cadence®

VXE User Guide

Product Version IXCOM20.12
June 2021

© 2021 All Rights Reserved.
Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Product VXE contains technology licensed from, and copyrighted by: Lucent Technologies, Inc, and is © 1991-1998, Lucent Technologies, Inc, All rights reserved; Carnegie Mellon University, and is © 1988, 1991 by Carnegie Mellon University, All rights reserved; loi Kim Lam, and is © 1993-1996 by loi Kim Lam, All rights reserved; Qlogic Fibre Channel Driver, © 1989, 1991 Free Software Foundation, All rights reserved; Emulex Fibre Channel Driver, © 1989, 1991 Free Software Foundation, Inc, All rights reserved; Prentice Hall, © 1995 by Prentice Hall, All rights reserved; © Regents of the University of California © Sun Microsystem, Inc. © Scriptics Corporation, Novus Copyright (C) 1996 - 2007 by Novas Software, Inc., Source III (C) Copyright 2006 Source III, Inc., U.C Riverside COPYRIGHT (C) 1995, 1996 by Nexsyn Design Technology Inc. All rights reserved.

Lucent Technologies disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness.

Associated third-party license terms may be found at <ROOTDIR>/install/Copyrights

Open SystemC, Open SystemC Initiative, OSCI, SystemC, and SystemC Initiative are trademarks or registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission.

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 800.862.4522. All other trademarks are the property of their respective holders.

Restricted Permission: This publication is protected by copyright law and international treaties and contains trade secrets and proprietary information owned by Cadence. Unauthorized reproduction or distribution of this publication, or any portion of it, may result in civil and criminal penalties. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. Unless otherwise agreed to by Cadence in writing, this statement grants Cadence customers permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used only in accordance with a written agreement between Cadence and its customer.
2. The publication may not be modified in any way.
3. Any authorized copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement.
4. The information contained in this document cannot be used in the development of like products or software, whether for internal or external use, and shall not be used for the benefit of any other party, whether or not for consideration.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third-party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor

Contents

1
Overview 45
Compiling the Designs for ICE Mode 48
Compiling the Designs for SA Mode 50
Compiling the Designs for SA Mode Using *xrun* 51
Debugging the Designs 51
Debug Vision Modes 52
Debug Sessions 54
Debug Operations 54
Waveform Database 57
Viewing Global Error Log 59
Using the readerr Utility 59

2
Setting Up the Environment for Palladium Z1 63
Mandatory Variables for the Palladium Z1 System 63
Environment Variables for Xcelium 64
Environment Variables Required in Specific Cases 66

3
Invoking Palladium Z1 Compilers and Debugging Tools 75
Invoking xeCompile 75
Invoking xeDebug Tool 76
Invoking ***xrun*** Tool (for SA Mode) 77
Using the XEL Shell 78
Initializing the XEL Shell for Design Compilation 78
Initializing the XEL Shell for Design Debug 79
Verifying Platform Information 79

© 2021 All Rights Reserved.

Quitting the XEL Shell . . . 79
Using Help for XEL Command Syntax . . . 79
Using Special Characters . . . 80
Using XEL Scripts . . . 80

4
Files and File Formats . . . 83

Configuration Files . . . 83
Configuration File Information . . . 84
Basic Language Elements . . . 84
External Timing Files . . . 85
Design Files . . . 86
Feature File for Enabling and Disabling Various Tool Features . . . 86
Memory Pattern Files . . . 87
Pattern File Format . . . 87
Pattern File Rules . . . 88
Pattern File Values . . . 89
Memory File Format . . . 90
Binary Memory File Formats . . . 91
Raw Format . . . 91
memTran Format . . . 92
Cycle-By-Cycle (CBC) Format . . . 93
Clock Domain Crossing (CDC) Format . . . 95

5
Using Utilities . . . 97

cds2libmap . . . 97
Purpose . . . 97
Syntax . . . 97
Parameters . . . 97
vhdel . . . 98
Purpose . . . 98
Syntax . . . 98
vhls . . . 99
Purpose . . . 99

© 2021 All Rights Reserved.

Syntax 99
Printing Full Hierarchy of Paths with -dh and -dhf Options - Example 101
vhmake 103
Purpose 103
Syntax 103
Description 103
vhshell 104
Purpose 104
Syntax 104

6
Working with the Palladium Z1 Emulators 109

Configuring the Emulator 109
Working with Emulators at Compile Time 110
Identifying Design Host and Domains 110
Compiling the Designs for Best Performance 113
Compiling the Designs for Best Relocation 113
Specifying the Connection to Target System 114
Working with Emulators at Run Time 131
Specifying the Emulator 131
Identifying Possible Locations for Downloading Design 132
Relocating Designs on Palladium Z1 Systems 133
Relocation Examples 139
Relocating Designs to Domains with Known Defects 152
Reserving Emulator Resources 154
Reducing Emulator Initialization Time Using KeepHostAlive 163
Using Suspend and Resume to Manage Emulator Resources 166
Saving and Restoring Design States 172
Compute Farm Control Mechanism 174
Farm Configuration Settings 174
Additional Options Supported in Farm Configuration Settings 176
Additional Settings for NC Compute Farms 177
Using Custom Compute Farms 177
Balancing Load Using Load Sharing Facility 179
Handling Failures at Run Time Using LSF 179

© 2021 All Rights Reserved.

Tracking Emulator Jobs . . . 181
Checking Emulator Status . . . 184
Job Sessions and Host Details Tables . . . 185
Colors of LED Boxes in Emulator Status GUI . . . 188
System Log Files and Hardware Configuration Files . . . 189
Setting Email Alerts for Error Messages . . . 190
Board-Manager and Power Log Files . . . 198
Physical Status LEDs - SCD, CCD, and LD . . . 198
Viewing Diagnostics Information . . . 199
Viewing Palladium Z1 Rack—LCD Display . . . 201

7
Improving Emulation Speed . . . 203

Improving Oversampling . . . 204
Upgrading from 4X Mode to 2X Mode . . . 204
Upgrading from 2X Mode to 1X Mode . . . 205
Upgrading from 1X Mode to 0.5 Mode . . . 205
Improving Step Count . . . 205
Factors Contributing to Step Count . . . 206
General Techniques to Improve Step Count . . . 206
Critical Path Reports . . . 209
Using Critical Path Reports to Improve Step Count . . . 214
Viewing the Critical Path in the Schematic Viewer . . . 218
Reports of Resource Utilization . . . 219
Effects of Memory on Step Count . . . 221
Using breakNet and breakPin . . . 225
Effect of Vision Mode . . . 226
Use of compilerEffort, optimizationEffort and placementEffort . . . 226
Use of partitionerEffort . . . 226
Using Design Retiming . . . 226
Files to Send to Customer Support . . . 227
Improving Step Count for Designs with External Target Interfaces . . . 227
Determining Signals Included in Critical Timing Paths . . . 228
Determining Emulation Cables that Connect to Targets . . . 229
Synchronizing with Target Clock to Resolve FCLK Delays . . . 229

© 2021 All Rights Reserved.

Troubleshooting System Problems 229

8
Behavior-Restricted Compilation 231

Introduction to Behavior-Restricted Compilation 231
Recommended Usage of Behavior-Restricted Compilation 232
Details of Behavior-Restricted Compilation 235
Step 1: Run Characterization Tests 235
Step 2: Behavior-Restricted Compilation 237
Controlling Behavior-Restricted Compilation 239
Step 3: Design Verification 239

9
Compiling Designs for In-Circuit Emulation with xeCompile 241

Preparing for Design Import 243
Creating the Directory Structure 243
Setting the Absolute Path for Libraries 244
Specifying RTL Design Files for Import 246
Specifying Verilog Designs for Import 246
Specifying VHDL Designs for Import 249
Specifying Mixed-Language Libraries for Import 251
Synthesizing the RTL Design for Emulation 252
Synthesizing Verilog or VHDL Designs 253
Synthesizing Mixed-Language Designs 255
Incremental Synthesis 256
Preserving the RTL Names 257
Maintaining Log Files 258
Using Synthesis Directives for Emulation 258
Synthesizing Designs Using Old Synthesis Engine 262
Synthesizing Designs with Memories 263
Using Memory Synthesis 263
Modeling Verilog Memories 265
Modeling VHDL Memories 267
Synthesizing Designs with Acceleratable Testbench (ATB) Constructs 270
Use Model for ATB 271

© 2021 All Rights Reserved.

Using hdlConfig Command at Run Time . . . 272
Supported Testbench Constructs . . . 272
Synthesizing Initial Blocks with Synthesizable Constructs . . . 280
Initializing Registers and Memories . . . 283
Support for LA Mode . . . 284
Executing Distributed Synthesis during RTL Synthesis . . . 284
Distributing the Synthesis Process . . . 284
Scheduling the Tasks . . . 288
Limitations of Distributed Synthesis . . . 288
Using Conformal to Verify HDL-ICE Compiler-Generated Netlist . . . 288
Generating Script for Conformal . . . 289
Starting Conformal . . . 290
Analyzing Results Using Conformal . . . 290
Using UNIX Commands for RTL Import and Synthesis . . . 292
Library Mapping File . . . 292
vavhdl . . . 295
vavlog . . . 297
vaelab . . . 301
Importing the Design . . . 308
Importing a Netlist . . . 309
Using Parallel Netlist Structure for Parallel Processing in Compile Flow . . . 312
Compiling the Design . . . 314
Opening the Design Library . . . 315
Selecting the Compilation Mode . . . 315
Determining the Vision Mode . . . 315
Selecting a Vision Mode Based on Preferred Debug Strategy . . . 317
Connecting Nets Using iScopedNets Program . . . 318
Creating Nets File for iScopedNets . . . 319
Providing Vector (Bus) Support . . . 320
Breaking the Loops . . . 321
Synthesizing Flip-Flops and Latches . . . 323
Inserting the Instrumentation Logic . . . 323
Running the Precompile Process . . . 325
Optimizing the Precompile Process . . . 326
Preparing the Design Files for Compilation . . . 326
Setting Up the Design for Precompile . . . 327

© 2021 All Rights Reserved.

Incorporating Physical IP . . . 327
Running ICE Prepare . . . 329
Support for Long Module Names . . . 329
Support for Retaining Flip-Flop Signals . . . 330
Support for Encryption in HDL-ICE Compiler . . . 331
Debug Capability of Encrypted Source Code . . . 331
Compiling the Design Files . . . 333
Using Parallel Partition Compiler for Large Designs . . . 333
Modular Compilation in ICE Mode . . . 337
Viewing Compile Results . . . 342
Using Compiler Options for Better Compile-Time Performance . . . 342
Using Multithreading Feature for Compilation . . . 353
Synthesizing and Compiling ALUs . . . 355
Controlling ALU Utilization During Synthesis . . . 356
Controlling ALU-to-Gates Transformations at Compile Time . . . 358
Connecting Hierarchical Nets . . . 360
Finding Faults Using ModelChecker . . . 362
Invoking the ModelChecker . . . 363
Types of Checks Performed by ModelChecker . . . 363

10
Compiling and Running Designs with Memories . . . 367
Supported Memory File Formats . . . 368
Transforming Memories . . . 368
Transforming Shallow Memory Instances into Gate Arrays . . . 369
Merging Memory Instances . . . 370
Splitting Memory Read Ports . . . 371
Squeezing Memory Instances with Constant Address Pins . . . 372
Compacting Memories . . . 372
Handling Sparse Memories . . . 373
Handling Sparse Memories through the Page Model . . . 374
Conditions When Sparse Memory Implementation Does Not Improve Capacity . . . 374
Modeling Sparsely-Used Memories at Compile Time in ICE Mode . . . 374
Accessing Sparsely-Used Memories at Run Time in ICE Mode . . . 376
Support for Sparse Memories in SA Mode . . . 378

© 2021 All Rights Reserved.

Specifying Verilog Memory as Sparse Memory in SA Mode 379
Comparison of Sparse Memory Support between SA and ICE Modes 387
Memory Slicing 389
Enabling Memory Assignment and Memory Slicing 389
Default Setting of Memory Assignment and Memory Slicing 389
Considerations for Memory Slicing 390
Using Memory Primitives 391
Building Memory Primitives 391
Enforcing Design Rules for Memory Primitives 392
Running Designs with Memories 394
Global Memory Options 394
The Memory Address Register 395
Loading Memories 395
Loading/Dumping Memory 396
Dumping Memories 398
Using Memory Streaming 399
Initializing and Masking Memories 401
Translating Memory Format Using the memTran Utility 401
Options for Specifying Memory Dimensions 402
Emulating TCAM Memories with Palladium Z1 405
Representing TCAM to the Compiler 405
Defining and Instantiating TCAM 405
Using TCAM in SA Mode 407

11
Modeling Clocks in ICE Mode 411

Supported Clock Technologies 411
Using the 1X Mode 412
Using the CAKE 0.5X Mode 437
Changing CAKE Clock Frequencies from DUT 439
Inserting Delays between Cross-Domain Synchronizers 440

12
Loop-Breaking 447

Definition of Loop-Breaking 447

© 2021 All Rights Reserved.

Problems Caused by Loop-Breaking 447
Proactive Steps to Avoid Loop-Breaking Problems 448
Steps to Break Loops and Long Paths 448
Miscellaneous Steps for Clock Delays 449
What to Do if a Loop-Breaking Problem is Suspected 449
Delays in Palladium Z1 450
Avoiding the Need for Oversampling 451
Preventing a Particular Break 452
How precompile identifies (clk) nets 453
How Precompile Avoids Delaying Clocks 454
Solving Clock-Delay Problems Caused by Loop-Breaking 454
Designs Containing Latches 456
Clock Gating Cells 457
Classification of Loops 458
Using the Precompile -breakHalfCycleAll Command 458
Using the Precompile -dontMinimizeDataBreaks (or -dataLoops101) Command 459
The *.brk File 460
The *.loops File 461
Displaying a Loop in the Schematic Viewer 463
Loops through Memory 464
Loops Discovered after Precompile 464
Detecting Clock Delay 464
Rarely-Used Options 467
Loop-breaking in SA Mode 467
Making Loop-breaking Delays Visible in Waveforms 467
Loops in BFM Modules 468

13
Using FIFO 469
Why Do We Need FIFO? 469

14
Modeling Guide for Verilog Designs for ICE Mode 475
Creating Synthesizable Verilog Designs 475

© 2021 All Rights Reserved.

Using Verilog HDL 476
Identifying Simulation Directives 478
Identifying System Tasks 479
Identifying Keywords 479
Identifying Unsupported Statements 481
Considering Synthesis Assumptions 481
Modeling Sequential Logic with always Blocks 483
Register Inference 484
Latch Logic Inference 488
Synthesizing Tristate Logic 490
Examples of Verilog Tristate Logic 490
Enabling or Disabling Memory Synthesis 491
Out of Module Reference 491
Verilog -2001 Configuration 492
SystemVerilog Support 493
Supported SystemVerilog Constructs for IEEE 1800-2005 493
Supported SystemVerilog Design Constructs for IEEE 1800-2012 506
Supported SystemVerilog Testbench Constructs for IEEE 1800-2012 518
Deviations from IEEE 1800-2012 Standard 524

15
Modeling Guide for VHDL Designs for ICE Mode 527

Entities and Architectures 528
Objects 529
Data Types 532
Operators 537
Operands 540
Aggregates 543
Subprograms 557
Packages 559
Configuration Declaration 563
VHDL Synthesis Directives 566
Enabling or Disabling Memory Synthesis 570
Syntax and Semantic Restrictions 570
Modeling Styles 571

© 2021 All Rights Reserved.

Support for VHDL 2008 584
INTEGER_VECTOR [5.3.2.3 and 16.3] 585
External names [8.7] 586
C-Style Comments [15.9] 586
elsif/else generate and case generate [11.8] 588
Condition Operator [9.2.9] 589
Enhanced Bit String Literals [15.8 (versus section 13.7 in 1993 LRM)] 590
Sequential Conditional Assignment [10.5.3 and 10.5.4] 592
Slices in Aggregate Phase 2 [9.3.3] 593
Generic Function/ Procedure 594
Slices in Array Aggregates [9.3.3.3] 596
Context Declarations [13.3] 597
RISING_EDGE and FALLING_EDGE Functions (section 16.3 and 5.2.6) 599
MINIMUM and MAXIMUM Functions (section 16.3 and 5.3.2.4) 600
Unconstrained Element Types (section 5.1 and 6.3) 601
Alias Declarations (section 6.6) 603
Deviations from VHDL 2008 LRM 603

16
Compiling Designs for Simulation Acceleration Using IXCOM
605

Overview of the IXCOM Compilation Process 607
Analyzing Mixed-Language Designs 609
Instantiating Verilog in VHDL 609
Instantiating VHDL in Verilog 610
Associating Logical Libraries to Physical Directories Using libmaprc File 617
libmaprc File Syntax 618
Optimizing Library File Processing 622
Enabling Concurrent Execution of vlan Sessions 622
Parallel Invocation of HDL-ICE Analyzer 625
Compiling Mixed-Language Designs 626
Partitioning Designs into RTL (Hardware) and Behavioral (Software) 627
Compilation Log Files 630
Support for SystemVerilog XMR Referring to VHDL Objects 632
Compiling for Hardware Using IXCOM 634

© 2021 All Rights Reserved.

Directing Design Compilation for Hardware 634
Detection of Hardware Platform Based on xeCompile Environment 635
Messages and Information about Detected Platforms 636
Switching the Platforms 637
Modifying the IXCOM Compilation Flow 638
Automatic Selection of Protocol for SA Transfer 639
Compiling SystemC Designs 640
Supported Types 640
Developing and Simulating SystemC Designs 641
Instantiating SystemC under Verilog/VHDL 643
Instantiating Verilog/VHDL under SystemC 644
Support for Multiple SystemC Shared Libraries 645
Using Specman Elite with IXCOM 646
Compiling and Running a Verilog Design in the Specman Environment 646
Compiling and Running a Mixed-Language Design in the Specman Environment . 647
Compiling Testbenches Separately 649
Linking DUT with Multiple Testbenches 649
Compiling Testbenches in Parallel 651
Compiling Top-Level Testbenches Using +xcDesignTop 654
Support for Bi-Directional Ports (inout) at the DUT-Testbench Boundary for Two-Top-Level Modules 657
Determining File Type by Source Filename Extension 659
Options for Verilog Design Compilation Mode 659
Options for File-Extension-Based Compilation Mode 660
Compiling Behavioral Constructs 662
Supported Behavioral Constructs 663
Scheduling Behavioral Processes 674
Detection of Behavioral Process Events 676
Identification of Behavioral Delayed Processes 677
Support for External Module References (XMR) in Behavioral Compilation 682
Using $IXC_CTRL Directives to Guide Behavioral Compilation 683
Transforming `initial` Blocks to `always` Processes 688
Compiling Behavioral Interface Constructs 691
Enabling Interface Compilation in IXCOM 691
Fine-Tuning Behavioral Performance 760
Reporting Behavioral Evaluation 760

© 2021 All Rights Reserved.

Identifying the Source of Behavioral Evaluations 761
System Task Mapping in IXCOM flow 762
Mapping of $display-Family Tasks 763
Support for System Tasks at Elaboration Time 764
Using System Tasks and Procedures at Run time 765
Supported Verilog System Tasks and VHDL Procedures 765
Executing Non-Synthesizable Code in the Emulator 766
Activating Monitor Code Only in Software Simulation 766
Monitoring Signals and Displaying Messages 767
Compiling DUTs with Non-Synthesizable Code by Using Testbench Callback Technology 768
Writing to DUT Signals Using Testbench Callback Tasks 776
Controlling Palladium Z1 from within Testbench Callback Tasks 776
Compiling Designs in Stages using IXCOM Compiler 778
Elaboration and Non-Elaboration Contexts 780
Creation for IXCOM-generated HDL (-target ixvcg) 780
Software Compilation Stage (-target swmodel) 780
Hardware Compilation Using HDL-ICE Compiler Stage (-target hdlice) 781
Hardware Compilation Stage Using xeCompile (-target xeCompile) 782
Generating Pulses 786
Pragma Handling in IXCOM 788
Rules for Using translate_off and translate_on Pragma 789
Ignoring Pragmas in VHDL Designs 790
Using //ixc Pragmas 790
Using Pragmas to Enable xrun to Handle Sparse Memories 791
Pragmas to Mark Design Portions for Bypassing IXCOM Compilation 791
Reading and Writing Signals in DUT Modules 795
Using Plus Arguments 805
Support for Encryption in SA Mode 806
Reading Encrypted Files 806
Support for IEEE Encryption Standards 807
Encryption with Levels of Debug Capability 807
Application of Protection Rights 808
Support for HDL-ICE Compiler Pragmas in Side File 809
Format of the Side File 810
Support for Strengths in SA Mode 811

© 2021 All Rights Reserved.

Handling of UDPs by IXCOM . . . 811
Support for Verilog Configuration Blocks . . . 813
Examples . . . 814
Compiling Designs with Verilog Configuration Blocks . . . 814
Support for Overriding VHDL Generics from the IXCOM Command Line . . . 814
Supported Data Types for Generic Overrides . . . 816
Example for Generic Override . . . 818
Comparison with Xcelium Support . . . 819
Assigning a Common Value to All Parameters with Specified Name . . . 820
Assigning Value by Name . . . 821
Assigning Value by Instance and Name . . . 821
Assigning Value by Full Hierarchical Path . . . 821
Examples of the Value Assignment Using the -gpg Option . . . 822
Support for VHDL Packages to be Imported Into SystemVerilog Design Files . . . 823
Limitations in Support for Importing VHDL Packages in SystemVerilog Designs . . . 825
Support for specify Blocks in SA Mode . . . 827
Deviations from Verilog 2005 LRM for specify Blocks . . . 829
Support for Compressed Files . . . 829
Enabling Run-time Access of For-generate Blocks and Instance Arrays Using Original RTL Names . . . 831
Using the -rtlNameForGenerate option for Run-time Access . . . 831
Reporting Delays in Initial Blocks . . . 832
Direct Access to DUT Memory Arrays and Registers . . . 836
Using the marg Library . . . 836
Using marg Interface in Pure Simulation . . . 836
C and Verilog Interface marg Functions . . . 837
Tasks for Memory-Array Buffer . . . 838
Tasks for DUT Register Access . . . 843
Examples: Passing Memory Data from Testbench to DUT . . . 845
SystemVerilog DPI marg Functions . . . 846
Specman marg Functions . . . 850
Enabling and Disabling Features at Compilation Time . . . 852
Syntax of the <feature_file> File . . . 853
Instance-Based Configuration . . . 856
Design-Unit-Based Configuration . . . 857
File-Based Configuration . . . 858

© 2021 All Rights Reserved.

Precedence Rules . . . 859
Examples of <feature_spec_lines> . . . 860
Controlling Assertions and Covergroup Through <feature_file> File . . . 863
Using Conformal to Verify IXCOM-Generated Output . . . 875
Generating the Files for Comparison with the LEC Tool . . . 876
Comparing the Models Using the xeLecRun Command . . . 878
Comparing Models in the OrigElab Mode . . . 878
Comparing Models Using the LEC Tool in the modNet Mode . . . 878
Files Generated by the LEC Tool . . . 879
Customizing the LEC Tool Invocation . . . 879
Generating Light Weight Debug Database (LWD) from IXCOM . . . 884
Handling of Multi-Step IXCOM Compilation Commands . . . 885
Handling of Single-Step IXCOM Compilation Command . . . 886
Handling of +xcDesignTop and -xmcompile Options with -enableLWD . . . 887
Generating the Indago-Compatible Debug Database with LWD Feature . . . 887
Support for External Netlists with LWD . . . 889
Generating Lightweight Simulation Snapshot for Large Designs . . . 890
Introduction to the Lightweight Flow . . . 890
IXCOM Hardware-Only-RTL Compile Flow Specification . . . 892
IXCOM Hardware-Only-RTL Run-time Checks . . . 896
IXCOM Support for XMUTILS of Xcelium . . . 896
Importing External Netlists in the SA Mode . . . 898
Using the Configuration File for Specifying Netlist Information . . . 899
Netlist Compilation . . . 900
Importing Netlist Modules . . . 902
Exporting RTL Modules into the Netlist Hierarchy . . . 908
Importing Netlists Using the XEL Scripts for ICE-Mode Compilation . . . 915
Generating Skeleton Design Netlists for IXCOM . . . 917
Using Hot Swap with Imported Netlists . . . 917
Accessing Netlist Objects from RTL Hierarchy . . . 918
Generating Clocks and Binding Clock signals in the Netlist Hierarchy . . . 920
Resolving Design Unit Redefinition . . . 921
Generating 32-bit Simulation Snapshot for Xcelium Simulator . . . 922

© 2021 All Rights Reserved.

17
Using $ixc_ctrl Pragma 925
IXC_CTRL Pragmas for Elaboration and Mapping 925
IXC_CTRL Pragmas for Interface Compilation 927
IXC_CTRL Pragmas for Behavioral Compilation 928

18
Compiling Designs with SystemVerilog Constructs in SA Mode 931
SystemVerilog Design Constructs 931
Compiling a Pure SystemVerilog Design 931
Compiling a Mixed-Language Design 932
SystemVerilog Testbench Constructs (SVTB) 932
Compiling SVTB Constructs Supported by IXCOM 932
Support for Covergroups and Mailbox Constructs 933
Support for SystemVerilog Interface Constructs 934
Interface Declaration 935
Support for Modports 935
Support for Virtual Interfaces 936
Interface Ports 939
Accessing Objects Through Interface Ports in the Hardware Partition 939
XMR References to Interface objects 940
Limitations In Interface Support 941
Support for Interface Port Arrays 941
Support for SystemVerilog Packages in SA Mode 944
Support for Packages Referenced Across Multiple Logical Libraries 947
Support for Packages with Same Names in Multiple Logical Libraries 950
List of Supported SystemVerilog package Constructs 953
Support for SVA and PSL Assertions with SystemVerilog Packages 954
Support for Coverage Constructs with SystemVerilog Packages 955
Support for DPI Declarations with SystemVerilog Packages 955
Handling of Stale Package References 955
Partitioning of SystemVerilog Packages by the IXCOM Compiler 955
Support for Compilation Unit Scopes in SA Mode 956

© 2021 All Rights Reserved.

Support for SystemVerilog bind Directive for Instantiations in VHDL Designs 958

19
Support for SystemVerilog Constructs 959

Limitations in the Support for SystemVerilog Constructs 982
Difference in Behavior in Software and Hardware Runs for Sampled Value Functions Used in Continuous Assignment Statements 982
Handling of Concatenation Operators 983
Support for Bounded Queues 983

20
Coverage Support 997

Xcelium Versions Required for Coverage Features 997
Code Coverage 1001
Toggle Coverage Support with Palladium Z1 1003
Block Coverage Support with Palladium Z1 1013
Expression Coverage Support with Palladium Z1 1015
Verilog and VHDL Process Restrictions in Code Coverage Support 1028
Differences in Code Coverage Support on Palladium Z1 with Respect to Xcelium 1028
Functional Coverage Support with Palladium Z1 1031
Covergroup Support in IXCOM 1032
Support for Covergroups in Dynamic Target Mode 1048
Support for SystemVerilog Functional Coverage Constructs 1049
Reporting for Functional Coverage Constructs 1061
Error Handling for Functional Coverage Constructs 1061
Support for SystemVerilog Control Functions 1062
General Flow for Using Coverage 1065
Additional Limitations of Coverage Support 1076
Generating a Coverage Database at Run Time 1076
Using Run-time Use Models 1081
Using Integrated Metrics Center for Analyzing Coverage Data 1083
Merging Emulation Coverage with Simulation Coverage Database 1085
Examples of Different Kinds of Merging of Simulation and Emulation Data 1091
Using vManager for Coverage Analysis 1094
Performance Considerations for Coverage Support 1098

© 2021 All Rights Reserved.

Performance Considerations for Code Coverage 1098
Managing Coverage Data 1099
Considerations Related to Coverage Verification on Palladium Z1 Emulator 1100

Quick Reference Guide for Covergroup Synthesis 1103
Making Covergroups Palladium-Compatible 1103
Choosing Between Covergroups and Cover Directives 1105

22
Modeling Clocks in SA Mode 1107
Manually Modeling Clocks in SA Mode 1107
Using userData to Create a Clock-Generation Module for SA Mode 1116
Oversampling in SA Mode 1131

23
Design Profiling for Palladium Z1 Acceleration 1133
Design Profiling on Software Using Xcelium Profiler 1133
Commands for Design Profiling Using the Xcelium Profiler 1134
Analyzing and Interpreting the Xcelium Profiler Output in ncprof.out 1138
Improving Profiling Accuracy and Caveats 1145
Profiling Design Run-Time Performance on Hardware with XC Profiler 1147
Enabling the XC Profiler 1147
Generating Hardware-Run Profiling Information Using +xcprof Option 1147
Generating Hardware-Run Profiling Information Using xc profon Command 1157
Integration with Advanced Profiler - xprof 1159
Collecting Data for Software Simulation Mode 1159
Collecting Data for Hardware Mode 1160
Using the xprof GUI to View Profile Reports 1160
Using xprof in Batch Mode 1164

24
Guidelines for Writing Testbenches 1169
Following a Synthesizable Coding Style 1170

© 2021 All Rights Reserved.

Using Initial Blocks (Verilog Only) 1170
Initial Block Statements That Delay Swapping to Hardware 1171
Initial Block Statements That Prevent Swapping to Simulation Acceleration Mode 1171
Specifying Clocks 1172
Applying Stimulus to Your Design 1173
Synchronous Testbenches 1173
Applying Test Vectors Using Memory ($random) 1174
Avoiding Testbench Race Conditions (Verilog Only) 1175
Using Hierarchical References in Testbench (Verilog Only) 1176
Accessing Memories Using Hierarchical References 1177
Using Verilog PLI in the Testbench 1178
Task/Function Routines Instead of Access Routines 1178
Reading and Writing Signals in DUT Modules 1179
Dealing with Testbench Monitoring Code 1180
Activating Monitor Code Only in Software Simulation 1180
Monitoring Signals and Displaying Messages 1182
Exercising Caution While Forcing Signals 1183

25
Compiling and Running Simulation Acceleration Designs with *xrun* 1185
Overview 1185
Setting Up the Environment to Use *xrun* for Simulation Acceleration 1186
Compiling and Simulating Designs 1186
SA-Specific Options of the xrun Command 1187
Compilation Options to Enable SA Mode 1188
Run-Time Options for Using xeDebug 1189
Accessing Help Information for IXCOM Tools from xrun 1190
Compiling Designs 1190
Verilog-Only Designs 1191
Mixed-Language (Verilog and VHDL) Designs 1192
Designs with PLI/VPI and VHPI 1193
Designs with SystemVerilog DPI 1193
Designs with SystemC 1194
Designs Using Specman ***e*** Testbenches 1195

© 2021 All Rights Reserved.

Parsing IXCOM and xrun Command Options with the *xrun* Tool 1196
Xcelium-Compatible Support for Testbench Compilation 1197
User-Partitioned Testbench and DUT Compilation 1198
Designs Using TBA and UVCs 1199
Logical Library Support 1199
Simulating Designs 1202
Simulating Using xmsim 1202
Simulating Using xeDebug 1203
Compilation and Run-Time Logs 1203
Limitations 1204

26
xrun - IXCOM Option Mapping 1205
IXCOM Options with the Same Syntax and Behavior as xrun Options 1206
IXCOM Options with Different Syntax but the Same Behavior as xrun Options 1206

27
Compiling and Running Designs with Assertions 1209
Enabling the Use of SVA 1210
Enabling the Use of PSL Assertions 1210
Disabling PSL Assertions in Selected Modules 1211
Labeling PSL Assertions 1212
Command Options for Compiling Designs with Assertions 1213
Enabling and Disabling Assertions at Compile Time 1213
Functional Coverage Support 1215
Generating Assertion-Based Functional Coverage Using VXE 1215
Integration with Integrated Metrics Center Tool 1216
Concurrent Cover Statement 1216
SystemVerilog Bind Support 1217
Using Assertion Control Tasks (ACTs) in HDL 1217
Usage of Formal Arguments of Property and Sequence Declarations in Constant Expressions 1218
Assertion Coverage of Assertions with Constant Property 1218
Feature-File Configuration for Optimization of Static Assertions 1219
Specifying Varying Widths for Assertion Counters 1219

© 2021 All Rights Reserved.

Running and Debugging Designs with Assertions 1219
Debugging Assertion Failures 1222
Xcelium Simulator Required for Exercising Assertions 1223
Support for Assertions on Emulator 1224
Using Assertions with InfiniTrace 1224
SDL Support for Assertions 1225
End-of-Sim Assertion Checks 1226
Coverage Support 1231
Assertion Reporting 1231
Assertion Summary Report 1231
Debugging Assertions Using Waveforms 1232
Assertion States 1233
Assertions-Related Features Required at Compile Time 1234
Assertion Run-Time Commands 1234
Assertion Tcl Variables 1235
Examples of SVA Usage 1237
Example 1: Using Concurrent Assert Statements 1237
Example 2: Using Concurrent Cover Statements 1238
Example 3: Using SV Bind 1238
Example 4: Functional Coverage Information 1240
Examples of PSL Assertion Usage 1244
Example 1: Using Embedded PSL Assertions (VHDL Only) 1244
Example 2: Using PSL Assertions in vunits (VHDL Only) 1247
Example 3: Using Embedded PSL Assertions (Verilog Only) 1250
Example 4: Using PSL Assertions in vunits (Verilog Only) 1253
Debugging Designs with PSL Assertions 1256
Using Assertions in Dynamic Target Mode 1263
Running Assertions in Dynamic Target Mode 1263
Support for Assertions Commands when Using Dynamic Targets 1264
Support for Specific SVA Constructs 1267
Support for PSL Constructs 1279
Default Clock Implementation 1279
Extended Support for Built-in Methods (VHDL Only) 1281
Differences between Xcelium and IXCOM for PSL Assertions 1281
Support for Specific PSL Constructs in a Mixed-HDL Design 1282
Support for Specific PSL Constructs in a Verilog Design 1298

© 2021 All Rights Reserved.

Support for Open Verification Library Components 1309
Supported OVL Components 1309

28
Methodology Guidelines for Compiling Assertions with the IXCOM Compiler 1311
Inspecting IXCOM Log File for Assertion-Related Information 1311
Number of Assertions Synthesized by the IXCOM Compiler 1312
Errors Generated during Compilation of a Design with Assertions 1313
Assertions Using X or Z Values 1315
Reporting Weights (Estimates of Gate Counts) for Assertions 1316
Advanced Support for SVA and PSL Assertions with Behavioral Compilation 1318
Assertions with Multi-Clocked Consequent 1319
Local Variables and Sequence Match Items 1320
Handling of Large Delays 1320
Handling expect Statements 1321
Differences in Handling of Assertions 1321
Assertion-Handling Differences Between Xcelium Simulator and Emulator (Compiled Using the IXCOM Compiler) 1322
Assertion-Handling Differences Between the IXCOM Compiler and LRM 1322
Best Practices for Assertions 1323
Using ifdef, Bind, or Vunits to Write Assertions 1324
Using Immediate Assertions when the Sequence Expression is a Constant Expression 1324
Using Variables Instead of Constants in Sequence Expressions 1325
Labeling Assertions 1326

29
Implementing SCE-MI Pipes 1327
Overview of Transaction Pipes 1328
SCE-MI 2.2 Features 1328
C++ HVL-side Transaction Pipes API 1329
SystemVerilog HVL-side Transaction Pipes API 1330
Clocked SCE-MI Pipes 1332
Byte Array Access Functions 1333

© 2021 All Rights Reserved.

Direct Memory Interface 1334
SCE-MI 2.1 Pipes Interface Features 1335
Deferred and Immediate Visibility Modes 1336
Notification Threshold 1336
Callback Threshold 1337
Functions of SCE-MI 2.1 1337
Software Components of SCE-MI Pipes Interface 1339
Encrypted HDL and Pre-Compiled Shared Libraries 1339
Testbench-Side Thread-Neutral C Interfaces 1341
DUT-Side HDL Interface 1343
Executing the Simulator Thread 1344
Reset of SCE-MI Pipes 1346
SCE-MI Protocol-Reset Functionality 1347
Using Language-Based Adaptors 1348
SCE-MI Pipes Blocking C Interfaces 1348
SCE-MI Pipes C++ Interface on Simulator Thread 1348
SCE-MI Pipes SystemC Interface on SystemC Thread 1349
SCE-MI Pipes SVTB Interface on Simulator Thread 1350
SCE-MI Pipes Specman Interface on Specman Thread 1351
Generating DUT Clocks with SCE-MI Pipes 1352
Compiling and Running Designs with SCE-MI Pipes 1353
Compiling and Running Designs in 32-Bit and 64-Bit Modes 1353
Sample Makefiles 1354
Compiling Designs 1355
Examples: Implementing SCE-MI Pipes 1363
SCE-MI 2.0 Examples 1363
Other Examples 1370
SCE-MI 2.1 Examples 1370
SCE-MI 2.2 Examples 1372

30
PLI/VPI and VHPI Support 1375

PLI/VPI Support in IXCOM 1375
Creating a PLI Table File 1376
Using the ncpli2tab Utility 1378

© 2021 All Rights Reserved.

Restrictions on Using PLI/VPI Applications 1381
Using a PLI/VPI Application in a Mixed-HDL Design 1381
Using a Verilog PLI Application in a Testbench 1382
Using a PLI/VPI Application in a DUT Module 1383
Statically Linking PLI/VPI Applications 1383
Naming PLI/VPI Applications 1383
Example: Using PLI and VPI Applications 1384
VHPI Support in IXCOM 1387
Restrictions on Using VHPI Applications 1387
Example: Using VHPI Applications 1388

31
Compiling Designs with SystemVerilog DPI 1391
Overview of SystemVerilog DPI 1392
SystemVerilog Layer 1392
Foreign Language (C/C++) Layer 1392
Supported/Unsupported C/C++ Functions 1393
import Functions and Tasks 1396
Definition of Import Declaration 1396
Example: SystemVerilog Layer Calling C/C++ Function 1398
Accessing Two-Value Bit Values in C/C++ 1400
Accessing Four-Value Logic Values in C/C++ 1400
export Functions and Tasks 1401
Example: C/C++ Layer Calling SystemVerilog Function 1401
Using SystemVerilog DPI 1403
SystemVerilog DPI Keywords in Verilog 1403
Using SystemVerilog DPI with Emulator 1404
Generating C Header Files for Exported DPI 1404
Support for Open Arrays in DPI 1408
Example: Using Open Arrays in DPI 1408
Querying and Accessing Open Arrays from the C Code 1409
Limitations in Using Open Arrays in DPI 1411
Supported Set of DPI Functions 1412

© 2021 All Rights Reserved.

32
Analyzing Power Consumption of the Designs 1413
Considerations when Using DPA 1414
Identifying Activity-Based Power Analysis Components 1414
Using DPA Methods in Design Flow 1416
Performing DPA in Offline Mode 1416
Analyzing Power Consumption of RTL Designs 1417
RTL Toggle Count Analysis 1417
Analyzing Power Consumption of Gate-Level Netlists 1423
Normal Toggle Count (NTC) Analysis 1423
Weighted Toggle Count (WTC) Analysis 1423
Using Tcl Scripts to Specify Power Tables for WTC Analysis 1436
Using Joules for Power Analysis 1447
Generating Backward SAIF Files for Power Analysis by Third-Party Tools 1448
Loading the Library Forward SAIF File into Palladium Z1 1449
Specifying Signals in Backward SAIF File 1449
Defining the Scope of Design Hierarchy Search for Instances 1450
Specifying the Time Windows for Backward SAIF File Generation 1451
Generating the Backward SAIF File 1451
Merging Multiple SAIF Files 1451
Accelerating SAIF Generation Using FullVision Partitioning 1451
Partitioning the Design 1452
Enabling FV Partitioning for PPC Designs 1453
Generating *.phy File during Emulation 1454
Generating SAIF Files in Offline Mode 1454
Parallel Computing for DPA Using Multiple CPUs 1456
Running DPA with Multiple Hosts 1456
Requirement on Remote Hosts 1457
Specifying Multiple Hosts Using Commands 1457
Specifying Multiple Hosts with rsh 1457
Specifying Multiple Hosts with LSF 1458
Setting Limitations on Remote Hosts 1458
Analyzing Power Consumption Using xeDebug GUI 1459
Setup Tab 1460
TCA Preparation Tab 1462

© 2021 All Rights Reserved.

TCA Tab 1463
DPA Probe Tab 1473
TCF Tab 1480
SAIF Tab 1482
PowerDebug Tab 1489
Analyzing Power Consumption with CPF 1495
Working with Power Domain in DPA GUI 1495
Creating *.ppfdata File for Power Domain 1497
Using Sparse Sampling with DPA 1498
Enabling and Disabling Sparse Sampling 1498
Estimating Power Using TCF/SAIF Files with Sparse Sampling 1499
Improving Accuracy of Power Estimation 1499
Generating High Quality TCF Files 1499
Estimating Accurate Power Consumption of RTL Designs 1500
Estimating Accurate Power Consumption of Gate-Level Netlists 1500
DPA Licenses 1501
DPA Tutorial Examples 1502

33
Specifying CPF Low-Power Intent on Palladium Z1 1503

Support for CPF Commands on Palladium Z1 1503
CPF 2.0 Features Supported on Palladium Z1 1506
Design and Instance Semantic Differences in CPF 1.1 and CPF 2.0 1507
Processing CPF Information on Palladium Z1 1508
Creating CPF Files 1509
Adding CPF Information to a Design 1510
Compiling a Design with Low-Power Intent 1511
Running and Debugging a Compiled Design with Low-Power Intent 1512
Design Considerations for Low-Power Verification 1530
Creating Power Domains 1531
Using Verilog Registers as Power-Domain Instances 1532
Defining Power-Domain States 1533
Resolving Inconsistencies between the Designs 1536
Handling Power Shut-Off Designs 1537
Handling Designs with Multiple Supply Voltages (MSV) 1539

© 2021 All Rights Reserved.

Handling Dynamic Voltage Frequency Scaling (DVFS) 1539
Handling Hierarchical CPF Files 1539
Logging of CPF Events 1550
Types of CPF-Event Logs 1550
Format of CPF-Events Log Messages 1553
Show CPF-Events Log Messages from a Log File 1553
Using CPF in SA Mode 1555
Support for CPF Features in SA Mode 1555
Compilation - CPF Constraints - Partitioning 1556
Support for CPF at Compile Time by IXCOM Compiler 1556
Support for CPF at Run Time in SA Mode 1557
Power Commands Supported in SA Mode 1560
SimVision Support for CPF in SA Mode 1560
Differences between Xcelium and Palladium Z1 CPF Low-power Implementation 1561
Using Dynamic Power Analysis with Low-Power-Aware Designs 1563
Selecting Low-Power-Aware Verification Method Based on Design Type 1565
Emulating RTL Low-Power-Aware Design 1565
Emulating Gate-Level Low-Power-Aware Design 1566

34
Working with IEEE 1801 Low-Power Designs on Palladium Z1
1569

IEEE 1801 Support Architecture 1570
Specifying IEEE 1801 Versions and Semantics 1572
Defining the Power Supply Network 1572
Processing IEEE 1801 Information on Palladium Z1 1573
Controlling 1801 Instrumentation Logic at Run-Time 1574
Compiling Low-Power Designs 1574
Overriding Ports During Compilation 1575
Identifying Fake Modules in SA mode 1575
Running and Debugging Designs 1575
Probing Low-Power Objects and Viewing Waveforms 1576
Checking Content of IEEE 1801 Objects 1577
Getting Value of IEEE 1801 Objects 1578
Using IEEE 1801 Conditions in SDL Expressions 1578

© 2021 All Rights Reserved.

Running Designs in SA Mode 1586
Verifying Low Power Aware Designs using the xeDebug GUI 1588
Runtime Tab 1588
Illegal Events Tab 1589
Power Intent Tab 1590
Power Supply Tab 1592
Isolation Tab 1593
Viewing Power Information for Probes in Xcelium SimVision GUI 1600
Simulating with xrun 1602
Specifying Power Back Initialization Values 1603
Memory Power Shut-Off Randomization 1603
Isolation 1603
Using Wildcards in IEEE 1801 Commands 1603
Using 1801 Boolean Expressions 1605
Modeling Non-Standard State Retention Registers 1606
Modeling Special ARM Retention Cells 1606
Modeling TSMC Retention Cells 1608
Modeling Cells Using NOR Isolation Strategy 1611
Performing Dynamic Power Analysis for Designs with 1801 Files 1611
Controlling Assertions for Powered Down Domains 1611
Logging IEEE 1801 Events 1613
Format of Logging Messages 1613
Showing 1801 Event Messages 1614
Logging 1801 Event Messages in Streaming Log Mode 1614
Logging 1801 Event Messages in Buffered Log Mode 1615
Handling IEEE 1801 Illegal Events 1616
IEEE 1801 Illegal Condition Checks 1616
Unsupported Xcelium Assertions 1621
Messages 1621
Enabling UPF 1.0 Isolation Semantics 1622
Locating Encypted Design Instances 1623
Identifying Partial Power-Aware Modules and Handling Corruptions 1623
Identifying Partially Powered Aware Models and Handling Corruptions in Liberty Flow 1624
Identifying Partially Powered Aware Models and Handling Corruptions in UPF Files . . 1625

© 2021 All Rights Reserved.

Supported Design Attributes for Power-Aware Models 1626
Specifying Default Value Conversion Table (VCT) 1627
Supported Design Attributes for VCTs 1628
Supported VCTs in VXE 1628
Identifying Power and Ground Supply Nets 1629
Viewing Default VCTs at Run Time 1630
Viewing Low Power Information in Palladium Database 1631
IEEE 1801 Support in Xcelium and Palladium Z1 1631
Similarities between Xcelium and Palladium Z1 IEEE 1801 Low-Power Implementation 1632
Differences between Xcelium and Palladium Z1 IEEE 1801 Low-Power Implementation 1632
Importing IEEE Standard UPF Packages 1636
Changing the Power States of Supply Nets and Ports 1636
Supported System Tasks and Functions 1637
Examples 1638
Generating a Coverage Model for 1801 Low-Power Designs 1639
Generating the Coverage Report 1640
Viewing the Coverage Report Using IMC 1641
Unsupported Low-Power Coverage Options 1644
Unsupported Xcelium Options 1645
Coverage Metrics for Supported IEEE 1801 Objects 1645
Viewing Power Loss in Waveforms 1646
Tracing HDL and Supply Nets for Designs with UPF Objects 1649
Monitoring 1801 Objects with Dynamic RTL 1653
Tips for Debugging Low-Power Aware Designs 1656

35
Support for IEEE 1801 Commands on Palladium Z1 1657
Supported IEEE 1801 Commands 1659
Supported VCTs in set_design_attributes 1677
Supported VCTs in connect_supply_net 1679

© 2021 All Rights Reserved.

36
Using the Dynamic Target Mode 1683
Overview of Dynamic Target Mode 1683
Clocking Requirements for Dynamic Target Mode 1684
Running Designs in Dynamic Target Mode 1684
Stopping a Design 1685
Switching between the Different Modes 1686
Assertions Support in Dynamic Target Mode 1687
Limitations of Dynamic Target Mode 1687

37
Importing Netlists as Blackboxes in SA Mode 1689
Use Model for Importing a Netlist as Blackbox Using IXCOM 1690
Testbench Access to Netlist Objects 1692

38
Design Debugging Processes 1697
Basics of Starting a Debug Session 1698
Terminating a User Debug Session from UNIX command prompt 1698
Invoking the Debugger Tool(s) 1699
Running Commands Remotely through xeclient 1699
Command Sequence for ICE Mode 1700
Debugging Process for SA Mode with xeDebug 1710
Debugging Process for SA Mode with *xrun* 1711
Dynamic Swap between Hardware and Software in SA Mode 1712
Using Hot Swap Options 1712
Using Hot Swap Run Modes 1713
Software Simulation Mode 1714
Run Mode 1714
TBrun Mode 1715
NBrun Mode 1717
C-TestBench Run Mode (CTB Run Mode) 1722
Autorun Mode 1724
Match Mode 1726

© 2021 All Rights Reserved.

Accessing Signal Values 1727
Forcing Values 1727
Releasing Values 1729
Depositing Values 1730
Initializing Registers Randomly in SA Mode 1732
Using the Palladium Xref Browser of SimVision GUI in SA Mode 1734
Using the Source Browsers of the Palladium Xref Browser 1736
Files Generated during Debugging 1738
xeDebug.log 1738
xe.msg 1738
xeDebug.key 1738
Debugging Designs with Modular Compilation 1739
Specifying Buckets to Debug 1739
Controlling Side File Generation for Bit-Blasted Memories 1741
Debug Commands Support for Modular-Compiled Designs 1741
Improving Debug Turnaround Time with Debugging in Regression Mode 1750
Limitations of Debugging in Regression Mode 1751

39
Run-Time Commands Support 1757

40
Probing Design Signals and Generating Waveforms 1759
Tracing Signals Using Probes 1760
Setting Probes 1761
Uploading Probes 1763
Probing Modes 1764
Partitioning FullVision Database 1771
Enabling Creation of Default Partitions 1771
Reducing FullVision Samples 1773
Controlling Maximal Trace Depth 1778
Changing the Vision Mode at Run time 1779
Probing SystemVerilog Design Constructs for SST2 and FSDB Waveforms 1780
Viewing Two-Dimensional Arrays in SimVision 1789
Viewing Modports in SimVision 1790

© 2021 All Rights Reserved.

Tracing Signals Using InfiniTrace 1791
InfiniTrace Sessions 1792
Using InifiniTrace in ICE Mode 1793
Using infiniTrace Command Options in HWIT Mode 1793
Using InfiniTrace in SA Mode Using xeDebug 1795
Using InfiniTrace in SA Mode Using xrun 1795
Comparing the InfiniTrace Feature in SA and ICE Modes 1796
Bookmarking InfiniTrace Session 1797
Tracing and Uploading Waveforms 1798
Tracing and Uploading Waveforms in ICE Mode Using xeDebug Tool 1798
Tracing and Uploading Waveforms in SA Mode Using xeDebug Tool 1801
Tracing and Uploading Waveforms in SA Mode Using xrun 1811
Generating Waveforms in Batch Mode with Partitioned FullVision 1811
Enabling FV Partitioning 1813
Loading swfvDB Files in xeDebug Offline Sessions 1813
Generating Parallel FSDB and SHM Files in xeDebug Offline Sessions 1814
Using shm_merge Utility 1816
Using fsdb_merge Utility 1817
Generating FSDB Files for Specific Instances 1818
Debugging in Offline Session 1819
Generating Trace Data and Preparing for Offline Session 1820
Starting an Offline Session 1821
Debugging in Offline Session to Save Emulator Time 1822
Using Offline Debugging 1822
Additional Debugging Features in Offline Session 1823
Examples of Using Offline Session 1824
Generating Waveforms in Offline Batch Mode 1826
Viewing Multiple Waveform Files 1829
Limitations of Waveform Generation in Batch Mode 1830
Generating Waveforms in Offline DYNP Mode 1831
Compressing *.phy Files 1831
Streaming Waveform Data 1832

© 2021 All Rights Reserved.

41
Debugging Designs and Controlling the Run on Palladium Z1

1837

Using the Palladium Xref Browser of SimVision GUI in SA Mode 1838
Using the Source Browsers of the Palladium Xref Browser 1839
Running Designs with Uncontrolled Clocks Using xeDebug in SA Mode 1841
Capture and Display Control 1841
Stopping the Design Clocks and Data Capture 1843
Using Uncontrolled Clocks with Other Debugging Operations 1846
Debugging Apparent Emulator Misbehavior 1847
Multiple Drivers on the Net 1847
Net Behavior is Altered by userdata 1848
Net is Forced at Run Time 1848
Compiler Bug 1849
Hardware Misbehavior 1849
Controlling Emulation Hang in SA Mode 1849
Aborting the Emulator Hang Using Files 1850
Controlling Behavioral Cycle of Waveforms 1851
Forcing TBsync Events in TBrun Mode 1851
Creating Checkpoints to Save Emulator's State 1852
Restrictions and Errors 1854
Compressing Saved Design Snapshots in ICE Mode 1854
Stopping Simulation with Initial Blocks 1855
Restarting Simulator and Resetting Emulator in SA Mode 1855
Starting Simulator on Remote Workstation 1856
Executing TCL Commands from C Testbench 1856
Viewing Optimized Nets in Design Browser 1857
Converting Signal Names 1860
Reducing Compiled Database Size 1860
Performing Process-Based Save and Restart Operations 1861
Prerequisites 1861
Supported Modes 1861
Enabling Process-Based Operations 1861
Restarting Snapshots in xrun Mode 1862
Saving and Restarting Snapshots in xeDebug SA mode 1862

© 2021 All Rights Reserved.

Accelerating Download Speed 1862
Controlling Execution of GFIFO and DPI Functions at Run Time 1865
Switching On or Switching Off GFIFO Functions 1866
Saving GFIFO Call Information 1867
Viewing Settings for GFIFO Functions 1868
Specifying Save Directory for GFIFO Calls 1869
Replaying GFIFO Functions 1869
Selecting the Replay Mode 1871
Controlling Execution of DPI Functions at Run Time 1874
Restrictions in Invoking void DPI-C and C Functions 1874
Using RTL Names with Palladium Designs 1876

42
Using the xeDebug GUI 1877

Invoking xeDebug GUI 1877
Using xeDebug Menu Bar and Toolbar Options 1880
Menu Bar 1880
Toolbar 1891
Starting a Debug Session 1912
Using Setup Tab in SA Mode 1912
Using Setup Tab in ICE Mode 1918
Running the Design Using the xeDebug Tabs 1921
Control Tab 1923
DRTL Tab 1933
SDL Tab 1937
Probe Tab 1941
Memory Tab 1947
InfiniTrace Tab 1951
Force Tab 1954
RTL/Gate Tab 1956
Deposit Tab 1957
Break Tab 1958
Clock Tab 1962
PartialFV Tab 1963
Saving/Restoring Debug Settings 1966

© 2021 All Rights Reserved.

Using Expressions in the Filter Field 1966
Saving Data Entered on xeDebug GUI Tabs 1966
Customizing the xeDebug GUI 1969
Using xegui XEL Command 1969
Loading libtk.so Tk Library (for Advanced Users) 1975

43
Controlling the Run with SDL 1977

SDL Concepts 1977
Using DRTL Instances in SDL Expressions 1979
Multiple SDL Instances 1979
Enabling and Disabling Individual SDL Instances 1980
Enabling and Disabling Individual Trigger, Display, TX, or Exec Statements 1981
Conditional Acquisition Using DCC 1981
Conditional Acquisition for Streaming Probes 1982
State Machine versus Global trigger 1984
SDL General Purpose Counters 1984
COUNT() Operator 1985
Statement Usage and Execution Semantics of an SDL State Machine 1985
SDL Clocking 1986
File Inclusion 1987
Aliases 1987
Use of Tcl Variables 1987
Assertions and CPF Objects 1987
Auto Probing 1987
Fast versus Accurate 1987
Delayed execution of TRIGGER and EXEC under IXCOM 1989
SDL Instrumentation Effect on Critical Path in SA Mode 1989
Debugging SDL Operation with the SDL Trace Dump Facility 1989
Saving and Restoring the Internal State of SDL 1991
Detection of Events without Triggering 1991
Limitations when Running with 1x Sampling Ratio 1991
Limitations when Running with 0.5x Sampling Ratio 1992
Interaction between SDL and XEL 1992
DISPLAY and TX Facilities 1994

© 2021 All Rights Reserved.

Recording Transactions with the TX facility 1994
Printing Messages from SDL with the DISPLAY Facility 1995
Emulation Resources Consumed by SDL 1996
Forming SDL Conditions in IXCOM Mode 1997
Summary of Basic SDL features 1998
SDL Preparation, Compilation and Execution 1999
Setup during Design Compilation 1999
Preparing the SDL Program 1999
SDL Enabling, Activation, and Execution 2000
Formatting and General Syntax Rules 2001
Format 2001
Comments 2001
White Space 2001
Identifiers 2002
Case Sensitivity 2002
File Inclusion 2002
Text Macros 2003
Conditional Compilation 2003
Using Values of Tcl Variables Inside SDL 2004
SDL Language Constructs 2005
Expressions 2011
Operators 2020
General Definitions 2029
Global or Instance Definition Sections 2031
INSTANCE Statement 2038
STATE Statement 2038
IF - ELSE statement 2039
Actions 2041
Examples: SDL Program 2063
Example 1: A State-Less SDL Program 2063
Example 2: A Single SDL Instance 2063
Example 3: SDL Program with Two Instances 2064
Importing Breakpoints from the Xcelium Simulator into SDL 2065
Theory of Operation 2066
Expressions in Breakpoints 2066
Restrictions and Error Conditions 2066

© 2021 All Rights Reserved.

Limitations and Differences in Behavior 2068
Creating Time Markers and Bookmarks 2069
Additional Tips 2072
Getting Live Feed Of Display Messages 2072
Using An Additional HDL Module To Track Complex Design Behavior 2072

44
Controlling the Run with Dynamic RTL 2075
Supported Modes 2076
Defining DRTL Modules 2077
Requirements 2077
Using Input and Output Ports 2077
Creating DRTL Instances 2078
Using DRTL Instances 2080
Using DRTL Instances in SDL Programs 2081
Using DRTL Output as the Input of Another DRTL Instance 2081
Using System Tasks in DRTL Modules 2082
Accommodating Large DRTL Instances 2087
Using Out of Module Reference (OOMR) with DRTL Modules 2088
Re-Instantiating DRTL Instances 2089
Saving and Importing DRTL modules 2090
Resetting the DRTL Instances 2090
Restrictions and Errors 2091
Dynamic RTL Example 2092
Initiating DPI Calls from DRTL Modules 2094
Defining GFIFO Monitor Modules 2094
Instantiating GFIFO Monitor 2095
Initiating DPI Calls 2096
Defining C Functions 2097
Executing DPI Embedded DRTL Modules 2098
Analyzing Capacity of DRTL Modules 2098

45
Debugging Designs using vvmDebug 2101
Introduction 2102

© 2021 All Rights Reserved.

Creating VVM Sessions 2102
Invoking the vvmDebug GUI 2106
Opening VVM Sessions 2108
SDL Trigger and Waveform Generation 2112
Selecting Trigger Modes 2113
Running Designs 2114
Managing Captured Trace Data 2115
Creating Bookmarks 2116
Creating SDL Programs 2118
VVM SDL Support Matrix 2118
Creating Probes 2120
Controlling Memory Dump Operations 2121
Debugging RTL Files and Gate Level Netlists 2124
Defining Fast Probes 2125
Analyzing Power Consumption in VVM Sessions 2126
Run-Time Commands Support for VVM 2127

46
Embedded Software Debugging with Palladium Z1 2133

Introduction 2133
Advantages of Embedded Software Debugging 2134
Hardware and Software Co-Verification with ESWD 2134
Offline Debugging Preparation 2135
Using vvmDebug 2136
Using xeDebug in Offline Mode 2136
Using InfiniTrace Sessions 2137
Generating ESWD Database 2137
Offline Debugging With Indago Debug Analyzer 2137
Viewing Modports in Indago 2141

47
Using Novas Tools 2143

Supported FSDB Versions 2143
Integrating Novas Tools with xeDebug 2143
Setting the Environment Variables for Running Novas Tools 2144

© 2021 All Rights Reserved.

Tracing and Uploading Waveforms 2145
Functions Supported by Novas Tools 2148

Index 2155

© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

1

Overview

Cadence® Palladium® Z1 is a verification computing platform that integrates the Xcelium™ Simulator tool to work together with Palladium® Z1 emulator for accelerating your simulation session without changing the design methodology.

The software tools that enable you to use the Palladium Z1 emulator have been collectively termed as Verification Xccelerator Emulator (VXE).

Starting with VXE 20.05, the VXE design verification system consisting of IXCOM functionalities, HDL-ICE functionalities, and other VXE functionalities will be available as three separate software builds instead of one, unified software build. Each of these softwares need to be installed in its own separate installation directory and required environment variables set to enable VXE to work properly. Refer to Mandatory Variables for the Palladium Z1 System section in Setting Up the Environment for Palladium Z1 chapter for information about environment variables.

The VXE 18.6.0 release supports the Xcelium class simulators starting with the Xcelium version **18.09.s004**. Each VXE 18.6.0 subsequent ISR-Hotfix is validated against the latest available Xcelium 18.09 ISR-Hotfix (Also known as a validated pair).

The VXE 19.10 release supports the Xcelium class simulators starting with the Xcelium version **19.09.001**. Each VXE 19.10 subsequent ISR-Hotfix is validated against the latest available Xcelium 19.09 ISR-Hotfix (Also known as a validated pair).

To understand the validated pair, refer to the *Readme.pdf* per ISR-Hotfix on downloads.This release will not work with Incisive class simulators. For information on the Xcelium simulator and Xcelium style commands, scripts, options, and environment variables, refer to the Xcelium documentation set.

This user guide describes the processes for compiling and running designs on the Palladium Z1 emulator. It assumes that the Palladium Z1 emulator has been installed and the required SA and host workstation setup is configured as per user requirements. For details about the installation of the Palladium Z1 emulator and workstation setup, refer to the *VXE Planning and Installation Guide* provided as a part of the VXE documentation set. A brief description to configure and work with Palladium Z1 emulator hosts at compile and run times is provided in the *Working with the Palladium Z1 Emulators* chapter.

© 2021 All Rights Reserved.

A design is compiled for different modes, namely In-Circuit Emulation (ICE) and Simulation Acceleration (SA) according to the requirements. After compilation, the compile result is imported into the run-time interface to run and debug the design on the selected Palladium Z1 emulator system.

Here is a brief description of the two modes:

- **In-Circuit Emulation (ICE) mode**

  In this mode, the design is completely synthesizable and runs entirely on the Palladium Z1 emulator. For ICE mode compilation, xeCompile tool is used, and the run-time and debug functionality on the emulator is handled by the xeDebug tool.

- **Simulation Acceleration (SA) mode**

  In this mode, only the synthesizable part of the design runs on the Palladium Z1 emulator. The rest of the design, that is, the non-synthesizable or behavioral part of the design, runs on the software simulator. For SA mode compilation, the SA compiler is primarily **I**ncisive **X**accelerator **COM**piler, that is, IXCOM. IXCOM compiles RTL, gate-level netlists, and some behavioral HDL constructs into the hardware emulator by internally invoking xeCompile, and keeps other behavioral elements and testbenches to run with the software simulator, *xrun*. At run time, you can select the VXE debug tool as xeDebug and/or *xrun* depending on the requirements. Even though xeDebug and *xrun* are two separate tools, both offer run-time and debug capabilities for SA mode, in the VXE design verification environment, these work in a fairly integrated fashion to ensure that you can access and manipulate the design objects in both software and hardware portions at run time.

When compiling the designs using VXE software tools, determine the appropriate compilation mode, ICE or SA, according to the design requirements. The VXE compile tool is xeCompile for ICE mode and IXCOM for SA mode. For SA mode, you can also choose to use *xrun*, which is the Xcelium software simulator tool to compile designs for Simulation Acceleration on Palladium Z1 emulator. *xrun*, then invokes IXCOM, internally to compile the design for emulation.

Consequently, following are the possible compile and run use flows offered on Palladium Z1 emulator:

- Compile the design for ICE mode using xeCompile and run using xeDebug

  Refer to Chapter 9, “Compiling Designs for In-Circuit Emulation with xeCompile” for details on the xeCompile tool and Chapter 38, “Design Debugging Processes” for details on the xeDebug tool.

- Compile the design for SA mode using IXCOM and run using xeDebug

© 2021 All Rights Reserved.

Refer to Chapter 16, “Compiling Designs for Simulation Acceleration Using IXCOM” for details on the IXCOM tool and Chapter 38, “Design Debugging Processes” for details on the xeDebug tool.

- Compile the design for SA mode using IXCOM and run using *xrun* (Software Simulation)
- Compile the design for SA mode using *xrun* and run using *xrun* (Simulation Acceleration)

  Note: This use flow helps you to easily migrate to SA mode, while continuing to use *xrun*, and accelerate the designs on the Palladium Z1 emulator. In this mode, too many changes in compilation and simulation command scripts are not required. This mode is documented in Chapter 25, “Compiling and Running Simulation Acceleration Designs with *xrun.”*

The following diagram illustrates the basic use flows available on Palladium Z1 emulator.

© 2021 All Rights Reserved.

Figure 1-1 Palladium Z1 Use Flows

Compiling the Designs for ICE Mode

In ICE mode, the xeCompile tool, which is the VXE Compiler, enables you to import the register-transfer-level (RTL) designs or gate-level netlist, prepare the design for compilation,

© 2021 All Rights Reserved.

and compile the design for emulation. The RTL should mostly follow synthesis guidelines when using ICE mode.

When you import RTL designs written in Verilog and/or VHDL, xeCompile works as an HDL-ICE Compiler and enables conversion of the RTL designs to a gate-level netlist or a database used to process the compilation result.

The Palladium Z1 emulator can be connected to a target system. A target system (or just a target) is an external board-level electronic system that you provide. It communicates with the emulated design through special connectors on an external panel of the Palladium system. A static target is a target system with clocks that can be stopped and restarted without corrupting its state. A dynamic target is a target with clocks that cannot be stopped and restarted.

Refer to Chapter 9, “Compiling Designs for In-Circuit Emulation with xeCompile” for a detailed description of using xeCompile.

In ICE mode, VXE supports Synthesizable TestBench (STB) and Logic Analyzer (LA) operation modes, as well as Vector-based Acceleration, also known as Vector Debug (VD).

Static Target Testbench (STB)

STB mode is used for emulating the DUT and the testbench, which is completely synthesizable, with a static target (that enables stopping and restarting the clock), or without a target system connection (targetless mode). You can start and stop the design clocks. All the design clocks must be generated inside the emulator. This is true whether a target system is connected or not. If a target system is connected to the emulator, stopping the emulator clocks will also stop the target system clocks.

STB is not suited for dynamic targets (whose clock cannot be stopped). If you try to use STB with a dynamic target, the emulator’s clocks can get suspended by some features, and this can cause the target to malfunction.

Logic Analyzer (LA)

Logic Analyzer (LA) mode uses a dynamic target. Although it can also work successfully with a static target or without a target at all, such configurations are not very useful because Logic Analyzer mode has only a subset of the debugging features of STB mode. With Logic Analyzer mode, a design can either have the Palladium Z1 system drive the target clock, or have the target system drive the design clock in the Palladium Z1 system. The design clock is normally always running. In this mode, data is continuously collected into a circular trace buffer inside the Palladium Z1 emulator. The probed data at that point can be uploaded to the workstation’s disk for viewing. Unlike STB mode, a trigger in LA mode only stops data

© 2021 All Rights Reserved.

collection, but not the design clocks. (It is possible to stop the design clock, but not to restart it.)

LA mode supports FV on Palladium Z1. However, be aware that you cannot append multiple triggers to one waveform. The previous waveform is cleared unless a different waveform file is specified before the next trigger and upload occurs.

Vector Debug (VD) Mode

Vector Debug (VD) applies test stimulus to the design from a vector file and obtains response vectors. The vector file contains a list of vectors (one per cycle) where each vector defines the stimulus and expected results for that particular cycle. This mode is most suited for regression tests created from simulation results or ATPG (Automatic Test Pattern Generation) software.

You can either call PTRAN as an independent executable to translate from a variety of stimulus file formats into Cadence's standard file, which Vector Debug uses directly, or you can use the Graphical Debug Tools to select the stimulus file format and the Vector Debug tool that will call PTRAN to automatically perform this translation.

Vector Debug mode cannot use a target connection.

Compiling the Designs for SA Mode

SA mode integrates workstation simulation with the hardware emulator for increased speed. SA mode enables you to enhance performance in a hardware-assisted environment. It enables you to verify and debug designs in which part of the design is accelerated by Palladium, while another part (typically the testbench) is run using *xrun*, while using the debugging capabilities of SimVision. SA mode accepts a wide range of language constructs and is not limited to a synthesizable subset.

In SA mode, the Palladium SA mode compiler, IXCOM, tightly integrates the Xcelium simulation kernel with the emulator, thereby giving the performance of an accelerator or emulator.

IXCOM provides an easy-to-use, simulator-style interface and a single-step compile command for both modes: software simulation mode and simulation acceleration mode with the emulator. The source files and command-line options provided with the `ixcom` command are passed to the respective back-end compilation tools by the IXCOM tool. For the back-end compilation, IXCOM internally invokes `xeCompile`, the ICE compilation tool, passing on the options and settings that you specify to the IXCOM tool as compiler directives. IXCOM handles Verilog and VHDL designs and provides support for SystemVerilog Assertions (SVA)

© 2021 All Rights Reserved.

and Accellera Property Specification Language (PSL) assertions, SystemC components, and PLI integration using *xrun*. IXCOM compiles the design and generates the simulation snapshot that can be run by using both the xeDebug and *xrun* tools.

Refer to Chapter 16, "Compiling Designs for Simulation Acceleration Using IXCOM" for a detailed description of using IXCOM.

Compiling the Designs for SA Mode Using *xrun*

This mode enables you to use *xrun* for compiling and running the designs in SA mode on the Palladium Z1 emulator. The original `xrun` commands can be modified by adding just a couple of command options to compile the designs for hardware. The *xrun* use model for simulation has been enhanced to invoke IXCOM tools to compile a design for simulation acceleration and simulate the hardware-accelerated snapshot. Thus, you can transition from the pure Xcelium-based software simulation mode to simulation acceleration mode using Palladium Z1. Details of this mode are described in Chapter 25, "Compiling and Running Simulation Acceleration Designs with *xrun.*" Details about the *xrun* tool are provided in the *xrun* User Guide as a part of the Xcelium documentation set.

Debugging the Designs

After compilation, the compiled database is downloaded to the Palladium Z1 emulator, and can be run in the ICE and SA modes, as per user requirements:

- For debugging designs in ICE mode, use the xeDebug tool to run and debug the design on hardware. Refer to Chapter 38, "Design Debugging Processes" for details on using the xeDebug tool.
- For debugging designs in SA mode, choose the debugging tool according to the portion of the design you need to access. When running in SA mode, the two portions of the design are distinguished as follows:
  - Device Under Test (DUT) is the portion of the design that runs on the emulator hardware.

    To access this portion, use the xeDebug tool.
  - TestBench (TB) refers to the portion of the design that runs on the software simulator.

    To access this portion, use *xrun*, the front-end tool for the Xcelium software simulator, which is integrated with Palladium Z1 to provide the complete Simulation Acceleration solution. In VXE, *xrun* can be run in:

© 2021 All Rights Reserved.

- Software Simulation Mode - In Software Simulation mode, the entire design runs in software on the host workstation; the emulator is not used at all. Running simulation in this mode is very similar, but not identical, to software simulation for designs that were not compiled for emulator modes.
- Simulation Acceleration mode - In the Simulation Acceleration mode, those parts of your design (the Device Under Test or DUT) that meet certain requirements are downloaded to the emulator and accelerated. The remaining parts not meeting these requirements are run in software simulation. In a typical scenario, the synthesizable ASICs would run in the emulator, while the behavioral testbench would run in software simulation.

  xrun usage is analogous to that of Xcelium with some exceptions. For detailed information on using *xrun*, refer to Chapter 25, “Compiling and Running Simulation Acceleration Designs with *xrun.”* More details on the *xrun* tool are available in the *xrun* User Guide of the Xcelium documentation set.

Debug Vision Modes

The options and design information available at the time of running and debugging the designs are determined by the Vision Mode selected at the compile time.

At compile time, you must chose from one of the following Vision modes:

- Dynamic Probes (Vision mode: DYNP)
- FullVision (Vision mode: FV - the default mode)

Dynamic Probes

In the Dynamic Probes (DYNP) Vision mode, you can specify the signals before the design run, during debug, depending on available probe channels. For DYNP, the maximum trace depth is 134,217,728 cycles (that is, 128 M cycles — where, M refers to 1024*1024). Conditional acquisition is supported with DYNP.

Advantages of using DYNP:

- Fastest waveform upload
- Avoids emulation speed costs that occur in some designs when FullVision Vision mode is used

Disadvantage of using DYNP:

© 2021 All Rights Reserved.

Before running a test, you must specify the signals to be probed in that run. The number of signals that can be probed in a specified run is limited.

FullVision

In the FullVision (FV) mode, all internal signals are visible at debug or run time. FV can be used when the emulator is connected to a dynamic target. The trace depth in FV mode is typically 2M cycles for a normal design. If a design is compiled with more number of domains, it gets a larger trace depth. You can use the `emulatorConfiguration` command to specify the number of domains. The trace depth in FV mode for small designs could reach tens of million cycles. The exact number depends on the design size.

Important

When using FV mode, you need to use 64-bit mode, which can be set using the `xeDesignBit 64` command.

In STB mode, FV supports incremental upload. You can mix run and trigger operations, add more probes, and upload probes. In SA mode, FV mode requires to capture every fclk cycle in which DUT status changes. During simulation acceleration, multiple DUT evaluations might be needed for one simulation time. Thus, one simulation time might need to capture the DUT status multiple times. The waveform displays only the last capture for each simulation time. Since samples were captured at multiple times (which are to be discarded later), this leads to a smaller trace depth. Conditional acquisition is supported with FV.

Advantages of using FV:

- You do not need to choose the signals to be probed before running.
- An unlimited number of signals can be probed.
- You can add new signals even if you are not currently running on the emulator.
- Any signal in the design can be probed.

Disadvantage of using FV:

For most designs, FV has no cost. For small number of designs, FV might increase the step count, thus making emulation speed slower.

© 2021 All Rights Reserved.

Debug Sessions

To run and debug a design, start either an online debugging session or an offline debugging session. To switch from an online debugging session to an offline debugging session (or vice versa), you must first release the current session.

Note: From this point onwards in this guide, the online debugging session will be referred as online session and the offline debugging session will be referred as offline session.

Debug Operations

When debugging the designs, you can perform signal trace operations, like specifying the signals to be traced, uploading the trace results to the trace file, and for power-aware designs, adding power-domain probe-related functionality, and collecting toggle count data during run time for dynamic power analysis.

Following are the common debug operations supported by the VXE debug tools:

- Tracing Signals

  You can trace signals and nets by adding probes and viewing them in waveforms.

  - Setting Probes:

    Probes are the means by which you can get values of signals from the design being emulated. Refer to Tracing Signals Using Probes for more details on setting probes.

  - Tracing Signals using InfiniTrace:

    The InfiniTrace feature helps save the state of the design during the run along a segment of simulation time, so that, this state can be replayed from any time points later to observe the emulation and simulation acceleration status. During the replay, you can observe the simulation status, dump out memory content, get the value of any of the nets in the design and add probes to upload the waveform of any nets.This feature offers infinite trace depth similar to simulation, and it overcomes the disadvantages of limited trace memory.

- Accessing Signal Values

  Both the xeDebug and *xrun* tools enable you to access signals at run time and set the signal values as required by forcing and depositing the values. When required, you can also release the values of selected signals. Refer to Accessing Signal Values on page 1727.

- Accessing Design Memories

© 2021 All Rights Reserved.

Memories can be read out (dumped) or written to (loaded) by using the `memory` command.

Also, individual memory locations can be viewed with the value command, and written into by the `deposit` command.

In emulation, the following memory file formats are supported:

- **For ASCII:** PD ASCII (`bin`, `oct`, `dec`, `hex`), readmemb or readmemh, NC_SIM, `%dn` (Denali memory format)
- **For binary:** memtran, raw, or raw2

Note: Loading and dumping memories to the *xrun* format on Palladium Z1 is not identical to the behaviors on standard *xrun*.

Refer to Running Designs with Memories for details about accessing design memories on Palladium Z1 emulator at run time.

- Controlling the Run with SDL and Trigger Conditions

SDL is a run-time debugging feature that enables, based on design signal values, control over several aspects of the run, such as when to stop, when to capture probed data (tracing), and how to insert command execution into the emulation flow.

SDL is the trace control and break mechanism, which can control data capture In-Circuit Emulation (ICE) modes (both STB and LA modes) and Simulation Acceleration (SA) mode. It can stop the running design in Vector Debug (VD), STB and SA modes, and it can schedule execution of XEL commands during a run in STB and SA modes. SDL is specified as a program in human-readable text.

For detailed information on SDL and its command options, refer to the Chapter 43, "Controlling the Run with SDL," and for the description of `sdl` command, refer to the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

- Controlling Assertions

Assertions can be used to verify that specific components in the design behave as expected, and can help in debugging by isolating problem areas. You can use XEL commands for assertion tracing.

You can select the assertions compiled into the design, view the assertion state and status information, and perform various operations assertions like:

- Enabling assertions
- Disabling assertions
- Adding or removing probes to selected assertions

© 2021 All Rights Reserved.

For more information on controlling assertions, refer to Compiling and Running Designs with Assertions on page 1209.

- Saving and Restoring Debug Environment

  At times, you might choose to re-create a debugging session. For example, you might want to save a set of probes and breakpoints on a design under development, so that it can be re-created at any time. You might want to save the preferences, re-create a complex mnemonic map, or open the same set of browser windows every time you simulate a design. The debugging environment is made up of the preference settings and the windows, cursors, Forcing and Releasing Testbench and DUT objects, and more.

  In addition to saving and restoring debugging sessions, the entire state of the design (including both DUT and TB) can be saved on the disk and restored later.

- Dynamically Swapping between Hardware and Software (Hot-Swap)

  You can choose to run the designs either only on the software simulator or on both the software simulator and emulator. You can dynamically change between the software simulation and SA modes at run time; that is, the current simulation state of a design is instantaneously swapped from one mode to the other. This is called ***hot-swapping*** and provides major benefits, especially for debugging purposes. For example, you can swap the design to Software Simulation mode to access its extensive debugging capabilities, then swap to the Simulation Acceleration mode for higher performance. The Command-Line Interface (CLI) commands and special system tasks in your HDL source give you control over this.

  At the time of hot swap (swap-out) the simulator takes full control and the emulator is released. Later, the design can be swapped back to the emulator (as available) and the emulator takes control (swap-in).

  For more information on hot-swap feature, refer to Dynamic Swap between Hardware and Software in SA Mode on page 1712.

- Debugging in Offline Mode

  Offline debugging enables you to run the design and save the traced results for later debugging without needing to run an emulator session. This frees the emulator for the use of others. Refer to Debugging in Offline Session for details about debugging designs in offline session using xeDebug tool.

  A new tool, vvmDebug, has been introduced, which offers enhanced offline debugging features for Palladium Z1 generated databases. Before debugging the designs offline using vvmDebug, you need to create VVM sessions online. For information on how to create VVM sessions and the operations that can be performed through vvmDebug, see Debugging Designs using vvmDebug.

© 2021 All Rights Reserved.

- Verifying Low-Power Designs Using Common Power Format or IEEE 1801

  Common Power Format (CPF) or 1801 enables verification of low-power designs. The appropriate files are set up at compile time (as described in Chapter 33, “Specifying CPF Low-Power Intent on Palladium Z1” and Chapter 34, “Working with IEEE 1801 Low-Power Designs on Palladium Z1”) and are compiled along with the RTL into the Palladium Z1 database. This enables a number of low power debug features that are available at run time.

  At run time, the design along with the power information is downloaded into the emulator and operated upon as needed. The information viewed in the waveform viewer displays the power information for the specified time or time interval.

- Dynamic Power Analysis

  Using the Dynamic Power Analysis (DPA) feature, you can estimate the total power consumed by a design to identify the power peaks that can break the functionality of the design and segregate the highly active modules. Details of the DPA feature are documented in Chapter 32, “Analyzing Power Consumption of the Designs.”

Waveform Database

To perform debug operations, you need to set the format of the waveform database at the time of invoking the debugger tool.

The waveform database can be generated in the following formats:

- SHM (also known as SST2)
- FSDB
- Value Change Dump (VCD)

The waveform database includes trace files that contain the data captured during the design debug process. The traced data can be viewed as waveforms using an appropriate waveform viewer.

SHM

SHM (also known as SST2) is the default waveform database format. The waveform output of the SHM format is stored in the output directory labeled as `<database_name>.shm`, where `<database_name>` is the same as specified with the `database -open` XEL command. This output directory contains two files, `<database_name>.dsn` and `<database_name>.trn`.

© 2021 All Rights Reserved.

The waveform database files of SHM format can be imported and viewed in the SimVision waveform viewer.

The `<database_name>.dsn` file contains the static design data such as design hierarchy, whereas, the `<database_name>.trn` file contains all the transition data. The size of the `dsn` file is limited to 4 GB and the size of the `trn` file is limited only by the filesystem. For example, on a Linux ext4 filesystem the limit of the `trn` file is 16 TB.

Note: If you add too many probes, the `<database_name>.dsn` file might attain maximum size limit, and this will cause the process to terminate with an error message.

FSDB

You can also choose to create a waveform database of FSDB format. At run time, the following three output files of FSDB format are created, but the `<database_name>_sdl.fsdb` file and `<database_name>_sdl.rc` files are created only when requesting to upload SDL activity, when SDL is enabled:

- A file named `<database_name>.fsdb` that holds the FSDB database
- A file named `<database_name>_sdl.fsdb` that holds the SDL COUNTERs and STATE
- A file named `<database_name>_sdl.rc` that has definitions for improved viewing, the most important one being a mapping of SDL state values to state names

The waveform database files of FSDB format can be imported into the Novas waveform viewers. The `.rc` file can be read by the Novas waveform viewer using its `restore signals` option.

Note: Before using the Novas tools to view waveforms of FSDB format, you need to set the required environment variables. For details, refer to the Setting the Environment Variables for Running Novas Tools section in Chapter 47, “Using Novas Tools.”.

Value Change Dump (VCD)

The VCD database records the data signal changes that occur during the design run. It records changes only on the signals that you have specified using the `probe -add <signal>` XEL command. The data capture starts after the design is run and trace results are uploaded to the trace file using the `database -upload` XEL command.

The VCD format significantly increases the debugging productivity by providing access to node history values from any point during run time, without requiring a rerun. The VCD format

© 2021 All Rights Reserved.

file is an ASCII file that comprises of header information, variable definitions, and the value changes for a variable in scalar or vector format, based on the variable type.

The SimVision (*xrun*) and xeDebug waveform viewers provide support for viewing VCD files.

Viewing Global Error Log

VXE provides a feature to write a short summary of any error reported while using most of the VXE tools. The error log is created at a central location for subsequent analysis of all errors accumulated since the error logging feature was turned on. To activate the logging of errors, you must create a soft link from `<VXE_INSTALL_HOME>/share/vxe/install/allerrors.msg` to a convenient location to access it.

The error log contains information similar to that recorded in other log files like `xe.msg`, `xeCompile.log`, `xeDebug.log`, and so on. The error log has the following columns:

- date+time
- host name
- user name
- tool name
- error kind
- error ID
- error text (first 80 characters)

As the link originates from the VXE installation directory, every VXE version can have its own error log or all versions can share the same log.

Cadence recommends that an AE or an R&D manager, not an individual engineer, reviews the error log.

Using the readerr Utility

Use the `readerr` utility to view the error log in different formats, such as an error count or report errors grouped by the requested fields. This utility can accept a key to group the results recorded in the error log and criteria to select results.

The following is the syntax of this utility:

```
readerr [-file=<file>] [-key=<key>] [-sort=<0|1>] [-host=<host>] [-user=<user>] /
        [-program=<program>] [-id=<id>]
```


© 2021 All Rights Reserved.

Here,

- `key` is a list of columns to group error count: `host`, `user`, `program`, or `id`. The default key is `id`.
- `-host=<host>` gets the count for only the given host name.
- `-user=<user>` gets the count for only the given user name.
- `-sort=<0|1>` sorts the errors by *count* if set to `1` or by *key* if set to `0`. The default value for sorting is set to `1`.
- An asterisk (`*`) after the number of errors means that the error text differs in the database.

Examples:

```
readerr -key=host,id -user=joe
readerr -host=planet -sort=0
```

To use error analysis, you need to create a link from `<VXE_INSTALL_HOME>/share/vxe/install/allerrors.msg`.

Usage

Calling the `readerr` utility without any options shows a sorted list of errors with the number of incidents for each error. The list is sorted by the number of incidents. For example:

```
% readerr
43984  4268* 20 DATABASE | 267: sim_configuration_management() failed: An SA
connection to the specified host co-
25444  3842* The program failed to initialize the hardware on host 'xxxxxxx' at
'03/12/2012 14:56:42'.
...
```

The first column lists the unique error ID, the second lists the number of incidents, and the last displays the text of the error message. If the text begins with `*`, it means that the text in the log is different for different incidents.

To select errors by a specific criterion, you can specify the required value for each field. For example:

```
% readerr -user=abcUser1
50173              511* OpenSession: simulator exited.
43287              316* Error in keep alive thread: xeDebug died, exiting...
...
```

The list of acceptable fields is: `host`, `user`, `program`, and `ID`. You can view this list from the man help page for this utility invoked by giving the `readerr -help` command.

© 2021 All Rights Reserved.

To group errors by a key instead of error ID, you can specify it with the `-key` option. For example:

```
% readerr -key=host
xxxxxx    2635* 20 ET6_API* | 692: The physical configuration file did not load
properly (rc='0C000004'x)
yyyyyy    2317* Severe: File xcva_top.etconfgs is empty or contains no valid
configuration fil-
...
```

In this case, you do not see the error IDs, but you see information such as a particular host had the displayed number of errors. A key can have several fields separated by commas. For example, `-key=host,id`

You can combine a key and criteria as illustrated in the following example:

```
% readerr -key=host -user=abcUser1 -program=xeDebug.bin
xxxxxx  355* The program failed to initialize the hardware on host 'xxxxxx' at '04/
09/2012 14:14:35'.
yyyyyy  151* current directory is already locked:   abcUser1 running on yyyyyy
(PID=17049)
...
```

The script is written in Perl and can be viewed to answer some questions not covered in this document. The fields in the log are separated by tabs (that is, `\t`) to simplify parsing, and a third-party analysis tool can be used for reports if needed.

© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

2

Setting Up the Environment for Palladium Z1

To compile and run the designs, the environment variables discussed in the following sections must be set:

- Mandatory Variables for the Palladium Z1 System
  - Environment Variables in the C Shell Setup Script File
- Environment Variables for Xcelium
  - Additional xrun Setup Options in Working Directory (Optional)
- Environment Variables Required in Specific Cases

Mandatory Variables for the Palladium Z1 System

Specifying the following variables is mandatory to enable Palladium to work seamlessly.

- `PATH`

  Set the path variable so that it includes the installation path to the three different software tool directories:

  - `<IXCOM_INSTALL_HOME>/bin`: Specifies the path to the IXCOM installation directory
  - `<HDLICE_INSTALL_HOME>/bin`: Specifies the path to the HDLICE installation directory
  - `<VXE_INSTALL_HOME>/bin`: Specifies the path to the VXE installation directory

    The path variable must also include the following Xcelium installation root directories:

    `<IUSHOME>/bin`

    `<IUSHOME>/tools.lnx86/bin`

    The path variable should not include the following directory:

    `<IUS_HOME>/tools.lnx86/bin/64bit`

© 2021 All Rights Reserved.

- `LD_LIBRARY_PATH`

  Set the library path variable, `LD_LIBRARY_PATH`, to include the libraries for appropriate bit mode.

  The library path variable must include the following Xcelium directories:

  <*IUSHOME*>/tools.lnx86/lib/64bit
  <*IUSHOME*>/tools.lnx86/inca/lib/64bit

  For using simulation with `xrun`, the library path must also include three separate software installation directories, which must be placed before the Xcelium directories:

  <*IXCOM_INSTALL_HOME*>/tools.lnx86/lib/64bit

  <*HDLICE_INSTALL_HOME*>/tools.lnx86/lib/64bit

  <*VXE_INSTALL_HOME*>/tools.lnx86/lib/64bit

Environment Variables in the C Shell Setup Script File

Following is an example of environment variables setting in the C shell setup script file for SA mode:

```
set IXCOM_INSTALL_HOME = /lan/cva_rel/ixcom20h1/20.04.049.b003
set HDLICE_INSTALL_HOME = /lan/cva_rel/hdlice20h1/20.04.049.b003
set VXE_INSTALL_HOME = /lan/cva_rel/vxe20h1/20.04.049.b003
set IUSHOME = /grid/cva/test_ius/xlm.18.03.005
set path = ($VXE_INSTALL_HOME/bin $IXCOM_INSTALL_HOME/bin $HDLICE_INSTALL_HOME/bin
$IUSHOME/bin $IUSHOME/tools.lnx86/bin $path)
setenv LD_LIBRARY_PATH /lib:/usr/lib:/lib64:/usr/lib64:$LD_LIBRARY_PATH
setenv LD_LIBRARY_PATH $IXCOM_INSTALL_HOME/tools.lnx86/lib/64bit:
                       $HDLICE_INSTALL_HOME/tools.lnx86/lib/64bit:
                       $VXE_INSTALL_HOME/tools.lnx86/lib/64bit:
                       $IUSHOME/tools.lnx86/lib/64bit:
                       $IUSHOME/tools.lnx86/inca/lib/64bit:$LD_LIBRARY_PATH
```

Where, `IUSHOME` is the root directory of Xcelium installation and VXE installation is separated into IXCOM, HDLICE, and VXE back-end software installation directories.

Environment Variables for Xcelium

The IXCOM compiler requires Xcelium tools and utilities in addition to the VXE software run on the Palladium Z1 system.

The coverage features are tightly integrated with the Xcelium simulator. As a result, newer functionality might not be available if an older version of Xcelium is used in conjunction with the IXCOM compiler. For more information on coverage features, refer to Chapter 20, “Coverage Support.”

© 2021 All Rights Reserved.

At run time, the IXCOM compiler checks if the simulator version is compatible with the VXE software. If the simulator version is not compatible with the VXE software, an error message is issued. You can bypass the compatibility check by using the `+ignoreSimVerCheck` option with the `ixcom` command, if you know that the VXE and simulator versions are compatible with each other. Bypassing the check for an incompatible simulator version results in an error or a tool crash.

Identifying Compatible Xcelium Versions

To determine the Xcelium version compatible with a Palladium software release, refer to the `README.txt` file available at downloads.cadence.com for each ISR.

Xcelium-Specific Environment Variables

Following are the variables required for Xcelium simulation tools and utilities:

Variable	Description
`PATH`	Defines the appropriate bin directory for Xcelium tools.
`LD_LIBRARY_PATH`	Defines the library path for running Xcelium tools.

Note: Refer to the *Xcelium Installation and Configuration Guide* for details about setting up the Xcelium environment.

Additional xrun Setup Options in Working Directory (Optional)

IXCOM lets you use the *xrun* options to compile a design for the emulator. IXCOM automatically generates a file named `<working_dir>/xc_work/hdl.var`, which enables the use of some commonly used *xrun* options with the `ixcom` command. You can specify additional *xrun* options to be used with IXCOM by adding or creating the following files in the working directory.

```
uxc_cds.lib
uxc_hdl.var
```

Here is an example of the `uxc_hdl.var` file:

```
DEFINE XMELABOPTS -loadpli1 myplifile.so:boot
DEFINE XMSIMOPTS -i myxmsim.tcl
```


© 2021 All Rights Reserved.

Environment Variables Required in Specific Cases

In specific cases, to perform a certain action, you need to set certain environment variables in addition to the mandatory variables discussed in the sections above. The following table lists such extra environment variables and their requirement:

Table 2-1 Environment Variables Applicable for All Modes

Environment Variable Applicable for All Modes	Description
`CDN_MT_HOST`	Specifies multiple workstations to improve the performance of Dynamic Power Analysis (DPA) and waveform generation. For details, see Specifying Multiple Hosts Using Commands in Chapter 32, “Analyzing Power Consumption of the Designs.”
`CDN_FV_MAX_THREAD`	Sets the maximum number of CPUs on a host that can be used for DPA and waveform generation. For details, see Parallel Computing for DPA Using Multiple CPUs in Chapter 32, “Analyzing Power Consumption of the Designs.” **Note:** The `CDN_FV_MAX_THREAD` environment variable is supported for both FullVision and DYNP probing modes.
`SMDB_SAVE_MC`	Required for viewing readable net names similar to the original net names in the `criticalPathReport.msg` file. For details, see Critical Path Reports in Chapter 7, “Improving Emulation Speed.”

© 2021 All Rights Reserved.

Environment Variable Applicable for All Modes	Description
`SWFV_FREEZE_PHY`	Restricts the growing size of the `*.phy` file by not saving the FullVision data to the file. Use this variable when you are interested in only generating the TCF/SAIF files and do not care about the waveforms. This will save valuable disk space and CPU time. **Note:** This will slow down the process of interactive waveform generation.
`SWFV_TMPDIR`	Specifies path of the directory that should hold the temporary files generated when dumping waveform or doing DPA. The specified path should be valid for all hosts. You should consider the impact on performance as the local directory is generally faster than the path on NFS.
`XE_SECURE_CONNECTION`	Uses `SSH` to establish a secure connection with the remote network host. Must be set before the software startup.
`CDS_LSF_OPTIONS`	Specifies complete path to the file that contains the Load Sharing Facility (LSF) options. For details about the content of the LSF options file, see Using LSF to Distribute Compiles with Different Seeds among Different Computers in Chapter 9, "Compiling Designs for In-Circuit Emulation with xeCompile."

© 2021 All Rights Reserved.

Environment Variable Applicable for All Modes	Description
CDS_MAX_PLACEMENTS	Specifies the maximum number of location subsets for job placements. Set the CDS_MAX_PLACEMENTS environment variable before executing the test_server -location command. ConfigMgr determines the location list by the *Static Weighted Scoring* method. The lower the score, the better the fit. The resultant locations are listed in the increasing order of scores. If the CDS_MAX_PLACEMENTS variable is not set, the test_server -location command might list up to 256 subsets. For more information on how to determine the best locations for downloading your design, refer to the description of the test_server command in the *Programs and Utilities* chapter of the *Command Reference Manual*.
CDN_FV_MIN_CPU	Sets the minimum number of CPUs on a host that can be used for LSF. For details, see Setting Limitations on Remote Hosts in Chapter 32, “Analyzing Power Consumption of the Designs.”
CDN_FV_MIN_MEM	Sets the minimum physical memory on a host that can be used for LSF. For details, see Setting Limitations on Remote Hosts in Chapter 32, “Analyzing Power Consumption of the Designs.”
CDN_FV_MIN_FREE_MEM	Sets the minimum free physical memory on a host that can be used for LSF. For details, see Setting Limitations on Remote Hosts in Chapter 32, “Analyzing Power Consumption of the Designs.”
CDN_FV_MIN_TMP_SPACE	Sets the minimum free disk space in the /tmp directory on a host that can be used for LSF. For details, see Setting Limitations on Remote Hosts in Chapter 32, “Analyzing Power Consumption of the Designs.”

© 2021 All Rights Reserved.

Environment Variable Applicable for All Modes	Description
IMPORT_MAX_PARALLEL	Changes, up or down, the maximum number of parallel invocations, either on same-host threads or multi-host LSF jobs. For details, see Using Parallel Import in Chapter 9, "Compiling Designs for In-Circuit Emulation with xeCompile."
IMPORT_LSF_QUEUE	Changes the default LSF queue established by the LSF administrator. For details, see Using Parallel Import in Chapter 9, "Compiling Designs for In-Circuit Emulation with xeCompile."
KEEP_ALIVE_XEDEBUG_DONT_EXIT	Retains the debug session even after DBEngine ends. **Note:** By default, xeDebug exits within 120 seconds if DBEngine ends unexpectedly. Specifying the KEEP_ALIVE_XEDEBUG_DONT_EXIT environment variable avoids the unexpected shut down of xeDebug.
■ LM_LICENSE_FILE ■ NOVAS_EXE ■ NOVAS_HOME ■ NOVAS_LIBPATHS ■ NOVAS_VERDI3 ■ NOVAS_VERICOM ■ TURBO_ETC_DIR ■ TURBO_LIBPATHS ■ TURBO_LIBS	Invokes the Novas tools. For details, see Setting the Environment Variables for Running Novas Tools in Chapter 47, "Using Novas Tools."
XE_RESERVE_KEY	Provides a resource reservation key when downloading the design. For details, see Reserving Emulator Resources section in Chapter 6, "Working with the Palladium Z1 Emulators."

© 2021 All Rights Reserved.

Environment Variable Applicable for All Modes	Description
XE_CREATE_TARGET_BP	If the design has a target, creates a .bp file, if one does not already exist, based on the domains specified during the compilation process. For details, see Creating BP File for Designs with Target section in Chapter 6, "Working with the Palladium Z1 Emulators."
XE_DB_SVR_DISP_MSG	Prints detailed information when the design is running. **Note:** The XE_DB_SVR_DISP_MSG environment variable must be set in the DBEngine shell before launching DBServer. For details, see Setting the DBServer section in Chapter 6, "Working with the Palladium Z1 Emulators."
XE_DB_SVR_RPC_NUM	Specifies a number that is unique for each user. **Note:** The XE_DB_SVR_RPC_NUM environment variable must be set in the DBEngine shell before launching DBServer. For details, see Setting the DBServer section in Chapter 6, "Working with the Palladium Z1 Emulators."
XE_KEEP_HOST_ALIVE	Retains the connection between the host and the emulator using the KeepHostAlive feature. For details, see Enabling the KeepHostAlive Feature section in Chapter 6, "Working with the Palladium Z1 Emulators."
XE_KEEP_HOST_ALIVE_TIMEOUT	Controls the time-out period value for the KeepHostAlive feature. For details, see Enabling the KeepHostAlive Feature section in Chapter 6, "Working with the Palladium Z1 Emulators."
XE_KEEP_PATH	Uses the PATH and LD_LIBRARY_PATH variables as-is when starting Xcelium from inside xeDebug.

© 2021 All Rights Reserved.

Environment Variable Applicable for All Modes	Description
`COMPRESSION_METHOD`	Specifies the compression method to compress the `*.phy` files for offline debugging. The supported compression methods are as follows: ■ `fast` ■ `slow` For details, refer to Compressing *.phy Files section.
`XE_SERVER_PORT`	Specifies the server port to be used by the `xeclient` command and the `xeserver.tcl` script. For more information, refer to Running Commands Remotely through xeclient section.
`XE_ICE_FAST_DOWNLOAD`	Enables fast downloading for designs in ICE mode. The valid values are: ■ `off` ■ `on` ■ `0` ■ `1`
`XE_SA_FAST_DOWNLOAD`	Enables fast downloading for designs in SA mode. The valid values are: ■ `off` ■ `on` ■ `0` ■ `1`
`CDN_FV_NO_LOCAL_HOST`	Specifies that the computation tasks for FullVision and DPA are only run on the remote hosts to reduce the load and memory usage of the local host. For more information on FullVision computation, refer to FullVision Mode section.

© 2021 All Rights Reserved.

Environment Variable Applicable for All Modes	Description
CDN_MT_HOST_TIME_OUT	Specifies the time-out value (in seconds) for the fvCompute process. The default value is 180 seconds. For details on the fvCompute process, refer to Optimizing Trace Data for Faster Interactive Offline Debugging section.
XE_DBGP_PARALLEL	Following are the valid values: ■ FALSE: The pass1 and pass2 processes are run one after the other. ■ LSF: pass1 is run by LSF ■ FORK: pass1 runs in another process generated by xeCompile. ■ NC: pass1 is run by the Network Computer (NC).
dbgPre_ST	Turns on the single thread processing for dbgPrepare pass1 process.
NO_CONSTRUCT	This variable turns off the construct support in the waveforms to accelerate the probe and upload process for SHM and FSDB waveforms. However, the individual bits of constructs are shown in the waveform window.
MAX_MDA_SIZE <*num*>	Specifies the maximum element size for multiple dimensional arrays for SystemVerilog constructs. The default value for <*num*> is 1048576. For more information, refer to the Viewing FSDB Waveforms section.
MAX_VECTOR_SIZE <*num*>	Specifies the maximum size for the vector elements The default value for <*num*> is 1048576. For more information, refer to the Viewing FSDB Waveforms section.
XE_POSTPONE_HW_START	Enables you to run the designs on machines even without the hardware. You can run on machines with the hardware, but postpone occupying the hardware. The XE_POSTPONE_HW_START environment variable disables starting DBEngine until the xc on command is issued.

© 2021 All Rights Reserved.

Environment Variable Applicable for All Modes	Description
XE_DEBUG_ARGS	Replaces the debug command arguments. This variable is supported in all modes. The XE_DEBUG_ARGS environment variable is significant in xrun mode. For example, in modular compilation, in which a design partitioned into smaller units called *buckets*. When a design compiled using modular compilation is run using the xrun tool (that is, with the xrun -R -xeDebug command), the debug command is executed implicitly and the arguments specified with the XE_DEBUG_ARGS variable are used as buckets. Example: setenv XE_DEBUG_ARGS=". -bucket 1" For details on modular compilation, refer to the following sections: ■ Modular Compilation in ICE Mode ■ Debugging Designs with Modular Compilation
XE_FSDB_VERSION	Specifies the FSDB format version for Verdi waveforms. VXE supports FSDB versions 5.1, 5.7, and 5.8 in both SA and ICE online and offline mode. The default FSDB file format is 5.1. For more information, refer to Using Novas Tools section.
NOVAS_VERDI3	Enables the xeDebug GUI to launch an older version of Verdi3 released in 2014: setenv NOVAS_VERDI3
NOVAS_VERDI3x	Enables the xeDebug GUI to launch newer versions of Verdi3 released after 2016: setenv NOVAS_VERDI3x
XCELIUM_INCISIVE_COMPATIBILITY_MODE [deprecated]	Forces use of INCISIVE naming conventions for all generated files and directories and also changes xcelium.d to INCA_libs. All content of xcelium.d including subdirectories of xcelium.d will be renamed to their Incisive names.

© 2021 All Rights Reserved.

The [deprecated] environment variables are deprecated from the VXE 18.5 release.

Table 2-2 Environment Variables Specific to SA Mode

Environment Variable Specific to SA Mode	Description
`XE_BOARDS`	Specifies the logic drawers and domains of the emulator for compilation. For details, see Identifying Design Host and Domains section in Chapter 6, “Working with the Palladium Z1 Emulators.”
`XE_DYNP`	Specifies the DYNP compilation mode instead of the default FullVision mode to compile the design. For details, see Identifying Design Host and Domains section in Chapter 6, “Working with the Palladium Z1 Emulators.”
`XE_HOST_FILE`	Specifies the full path to the `*.et3confg` file instead of an emulator. `setenv XE_HOST_FILE <et3confg_file_path>` For details, see Identifying Design Host and Domains section in Chapter 6, “Working with the Palladium Z1 Emulators.”
`XE_HOST`	In SA mode, specifies the name of an emulator which can be used to identify the emulator to compile for. For details, see Specifying the Emulator Name section in Chapter 6, “Working with the Palladium Z1 Emulators.”
`DBE_HOST`	Specifies the DBEngine host. For details, see Specifying the DBEngine Host section in Chapter 6, “Working with the Palladium Z1 Emulators.”
`XE_XMSIM_EXIT_TIME`	Specifies the time (in seconds) to extend the timeout for the Xcelium simulator. If simulator becomes unresponsive, xeDebug forces the simulator to exit leaving a running process. Use this environment variable to specify a wait time for the simulator.
`VMANAGER_WRITE_METRICS`	Generates the VSOF file. For details, refer to the Using Palladium-Generated VSOF Files section.

© 2021 All Rights Reserved.

3

Invoking Palladium Z1 Compilers and Debugging Tools

This appendix describes how the various compiler and debugging tools for Palladium Z1 emulator are invoked.

Invoking xeCompile

xeCompile is invoked by executing the `xeCompile` command on the UNIX command prompt.

Important

xeCompile must be invoked from the design directory.

The `xeCompile` command can be executed with or without parameters.

When specified without a parameter, the `xeCompile` command invokes xeCompile in an interactive command-line mode.

Note: After the xeCompile tool initialization is complete, the command prompt changes to `XEC>`.

To compile a design in batch mode using a Tcl script, execute the following `xeCompile` command on the UNIX command prompt:

```
xeCompile <script_name>
```

XEL Compile Commands

VXE supports its own set of XEL commands, variables, and parameters. XEL is an interpreted language that allows you to use the VXE verification functions either via a script or directly on the command line. XEL is based on the Tool Command Language (Tcl).

The XEL command prompt is available both in the xeCompile and xeDebug windows, as well as through a dedicated XEL shell that can be invoked from a UNIX window.

© 2021 All Rights Reserved.

The XEL commands are documented in the *VXE Command Reference Manual,* and are also available as XEL-level `help` commands. Type `help <XEL_command_name>` to view the documentation for a specific XEL command.

Note: You should not run the debug XEL commands within xeCompile and the compile XEL commands within the VXE Debugger tool called xeDebug. Doing so will generate an error because the XEL shell for the compiler and debugger tools call different sets of XEL commands based on which tool you invoke.

Invoking xeDebug Tool

xeDebug is invoked by executing the `xeDebug` command, with or without parameters, on the UNIX or Linux command prompt.

Following are the `xeDebug` command options that allow you to invoke the xeDebug tool in six alternative ways:

- ```
  xeDebug [-gui|-noecho][-shm|-fsdb|-vcd] \
          [-init <tcl_file>] ... \
          [-log <name>] [-append_log] [-append_key] \
          [--xmsim <option> ... [--]] [script [<arg> ...]]
  ```
- ```
  xeDebug [-shm|-fsdb] -wave <tracefile> \
          [script|probelist|rcfile]
  ```
- ```
  xeDebug [-shm|-fsdb] -offline <session> <tracefile> \
          [script|probelist|rcfile]
  ```
- `xeDebug -clean | -cleanall`
- `xeDebug -version`
- `xeDebug -help`

Note: These are alternative methods, and options specific to one method cannot be mixed with another. When specified without a parameter, the `xeDebug` command invokes the xeDebug tool in an interactive command-line mode. After the initialization of the xeDebug tool is complete, the command prompt changes to `XE>`.

For more information on the usage and the syntax, refer to the *Programs and Utilities* chapter of the *VXE Command Reference Manual*.

The xeDebug tool can be launched in the following modes:

- **GUI mode**

© 2021 All Rights Reserved.
  ```

```
xeDebug -gui [-shm|-fsdb] [-init <tcl_file>] \
        [--xmsim <xmsim_options> ... [--]] \
        [<script_name> [<script_options> ...]]
```

Using the xeDebug GUI is described in Appendix 42, "Using the xeDebug GUI."

- **Script mode**

```
xeDebug  [-noecho] [-shm|-fsdb] [-init <tcl_file>] \
         [--xmsim <xmsim_options> ... [--]] \
         [<script_name> [<script_options> ...]]
```

Note: The `[--xmsim <xmsim_options> ... [--]]` is applicable only to SA mode.

- **Offline mode**

  - For viewing FullVision probes:

```
xeDebug [-shm | -fsdb | -vcd] [-offline <session_name> <phyfile>] \
        [<pscript> | <plist> | <rcfile>]
```

  - For uploading probes and viewing the waveform with only trace files specified:

```
xeDebug [-shm | -fsdb | -vcd] [-wave <phyfile> ] \
        [<pscript> | <plist> | <rcfile>]
```

For details, refer to Debugging in Offline Session section.

During a debug session, if you need to verify the mode in which xeDebug was started, execute the `xeset waveType` command. This command returns either `shm` or `fsdb` depending on the mode in which xeDebug was started.

Invoking *xrun* Tool (for SA Mode)

To invoke *xrun* in script mode, use the following command:

```
xrun -R -tcl
```

To invoke *xrun* in GUI mode, use the following command:

```
xrun -R -gui
```

xrun provides the native debug environment of Xcelium simulator, but you do not have full debugging access to the DUT in the emulator. The xeDebug tool provides full debugging access to both the testbench in the simulator and the DUT in the emulator, though xeDebug debugging environment is different from *xrun*. If you invoke *xrun* in its default mode, the design is run in pure software-simulation mode.

To use *xrun* for Simulation Acceleration mode, and run the designs on the Palladium Z1 emulator with *xrun* as the Debugger, you can internally invoke the xeDebug tool from *xrun*. For this, first use the `-hw` and `-ua` options of the `xrun` command to compile the design for

© 2021 All Rights Reserved.

hardware and then use the `-xeDebug` option with the `xrun` command. Refer to Chapter 25, "Compiling and Running Simulation Acceleration Designs with *xrun.*"

The IXCOM compiler sets the default access to read and write (`-access +rw`) for the design objects. For performance reasons some designs do not require this access level. You can use the `-access <file>` option to modify the default value as follows:

1. Create an access file with the desired visibility access for the design objects

   For example:

   ```
   // File: ./access.file
   DEFAULT +r-wc
   ```

   For details on writing and using an access file, see *Using an Access File* in the Xcelium documentation.

2. Specify an access specification in the `./uxc_hdl.var` file

   For example:

   ```
   DEFINE XMELABOPTS -afile ./access.file
   ```

3. Compile and simulate the design

Note: *xrun* can be launched from both UNIX and Linux shells. However, the SimVision snapshot is platform dependent, that is, if the snapshot was generated on a Linux platform, the xeDebug and *xrun* tools need to be run on Linux platform too.

Using the XEL Shell

The XEL shell is a text-only XEL interpreter. Based on whether you are compiling the design or debugging it, the XEL shell is populated with the corresponding set of commands.

Initializing the XEL Shell for Design Compilation

Execute the `xeCompile` command on the UNIX prompt to start the XEL shell for compiling a design. This command initializes the XEL shell, prints out messages, and returns the `XEC>` prompt for executing the XEL compile commands. For details about the `xeCompile` command, refer to *Programs and Utilities* chapter of the *VXE Command Reference Manual.*

© 2021 All Rights Reserved.

Initializing the XEL Shell for Design Debug

Execute the `xeDebug` command on the UNIX prompt to start the XEL shell for debugging a design. This command initializes the XEL shell, prints out messages, and returns the `XE>` prompt for executing the XEL debug commands. For details about using the `xeDebug` command, refer to the *Programs and Utilities* chapter of the *VXE Command Reference Manual*.

Verifying Platform Information

You can verify the product name, model name, release name, and so on for the currently running program. Execute the `xeset platform` command to get information about the program that is running.

For more information, refer to the description of the `xeset platform` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Quitting the XEL Shell

Typing either `quit` or `exit` on the `XEC>` or `XE>` command prompt enables you to log out of the XEL shell. Consequently, all active subsystems are closed and the UNIX prompt returns.

Important

Cadence recommends that you do not use the `exit` command because the `exit` command immediately stops any post-simulation-Xcelium-processing and exits the shell. When the `exit` command is used, the information (related to covergroups) collected while swapped into hardware will not be processed. But this processing is essential in certain cases, such as handling covergroups while running xeDebug in SA mode. Therefore, it is best to use the `quit` command to quit the XEL shell.

Using Help for XEL Command Syntax

For help on a specific XEL command, from within the XEL shell, type `help`, followed by the name of the XEL command. The help page contains information on syntax of the specified XEL command.

For example, to view help about creating a User Data file, enter the following command on the xeCompile command prompt:

```
XEC> help userData
```


© 2021 All Rights Reserved.

For a list of topics, enter the following command on the xeCompile or xeDebug command prompt:

```
help *
```

For details on conventions used in describing XEL command syntax, enter the following command:

```
help Syntax
```

Using Special Characters

Use of wildcard characters in XEL varies by command, with some allowing use of wildcard, and some not allowing. For specific usage, refer to the *VXE Command Reference Manual*.

Table 3-1 Wildcard Characters

Character	Description
*	Matches any sequence of characters.
?	Matches any single character.
[char]	Matches any single character from the character list. If the character list contains a sequence of the form a-b, any character between a and b, inclusive, will match.
\x	Matches the single character represented by x. This provides a way to avoid special interpretation for any of the characters *?[]\ in the pattern.

Using XEL Scripts

When you are familiar with XEL, you can create XEL scripts to accomplish emulation tasks. Scripts can automate long or repetitive tasks. Sample XEL scripts are provided both in the *VXE Command Reference Manual* and in the `<install_dir>/share/vxe/gift` directory.

XEL scripting supports the entire command set of Tcl version 8.3.

Palladium Z1 records all emulation commands as XEL commands within a log file named `xeCompile.log`. After completing a series of emulation tasks, this file can be used as an example of how to construct an XEL script. You can also use this log as an input for creating an XEL script file.

© 2021 All Rights Reserved.

Some XEL commands or arguments for obtaining status information are purposely not logged to limit the size of the log file.

© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

4

Files and File Formats

This section describes files that the VXE Compiler either generates or reads.

Topics discussed in this chapter include:

- Configuration Files
- Design Files
- Feature File for Enabling and Disabling Various Tool Features
- Memory Pattern Files
- Memory File Format
- Binary Memory File Formats
  - Raw Format
  - memTran Format
- Cycle-By-Cycle (CBC) Format
- Clock Domain Crossing (CDC) Format

Configuration Files

The configuration file lists the available resources installed on a Palladium Z1 system or the resources allocated for a design. Configuration file extensions must be `et3confg`. This file uses the following conventions for defining resources:

- Any line starting with a plus (+) symbol lists an available resource.
- Any line starting with a minus (-) symbol lists a defective or excluded resource.
- Any line starting with an asterisk (*) is a comment line.

All resources are identified by name and resource specific configuration information.

© 2021 All Rights Reserved.

The initial configuration file is generated when the emulator is first started, through the automatic configuration utility when the system is first set up. (The automatic configuration script is run at the same time as the initial setup diagnostics.)

Configuration File Information

The configuration file contains:

- The target configuration table
- The maximum step count
- Optional constraints for signal assignments, such as I/O signals, trigger signals, and probe signals
- Clock phasing information

Basic Language Elements

The table below defines the variables used in configuration files.

Table 4-1 Configuration File Variables

Variable	Example
<alu_number>	0..1
<board_name>	48_CHIP_BOARD or 192_CHIP_BOARD
<board_number>	0..3|0..3|0..31
<board_Jconn>	0..4
<board_type>	ET6
<cable_type>	CABLE_*<length>* For example: CABLE_106, CABLE_154, CABLE_195, CABLE_230
<connector_location>	*<module_location> <connector_position>*
<connector_number>	*<board_number> <module_number> <board_Jconn>*
<connector_position>	T|B|L|R or J2|J4|J1|J3 (**T**op|**B**ot|**L**eft|**R**ight)

© 2021 All Rights Reserved.

Table 4-1 Configuration File Variables

Variable	Example
<domain_on_board>	0..3|0..3|0..15
<module_location>	*<board_number> <module_number>*
<module_number>	0..3|0..3|0..15

External Timing Files

The Compiler creates the *<design>*.xtime external timing file.

The *<design>*.xtime file shows the timing for all primary output signals (one line per primary output signal).

Syntax

<primary_output_signal_name> <emulation_step_number> <pulse>

where:

<*primary_output_signal_name*> is a primary output signal name.

<emulation_step_number> is the emulation step number when the primary output signal changes.

<pulse> can be one of the following designations:

- +
- -
- +number
- -number

The + symbol specifies an initial value of zero for the primary output signal.

The - symbol specifies an initial value of one for the primary output signal.

Initial value means the value that must be set on that signal when the design is downloaded into the emulator (similar to time zero initialization in simulation). The net returns this value until it is changed by a stimulus applied to the design. The initial values are useful for protecting target systems that have tristate interfaces to the emulator.

© 2021 All Rights Reserved.

+number specifies a pulse width of the number of emulation steps for the primary output signal. The primary output signal is observable at the external pin from the time specified by the <*timing_constraint*> for at least the number of emulation steps. All other times, the external pin is forced to zero.

-number specifies a pulse width of number emulation steps for the primary output signal. The primary output signal is observable at the external pin from the time specified by the <*timing_constraint*> for at least the number of emulation steps. All other times, the external pin is forced to one.

Example

```
PWATL_IN       27      -
WFB_EXT_BCLK  111    +10
WFB_MP_CLK      6   -128
```

Design Files

When you successfully import an RTL Verilog or VHDL design, the xeCompile creates gate-level design files. The format of these files depends on the format specified with the hdlOutputFile XEL command. The file format options are Verilog and VHDL.

A separate file is created in each specified format, for each module of the design. This includes a separate file for the top-level module, each hierarchical module, and each of the lowest-level instances.

The subset of features supported in these files conform to standard Verilog HDL (IEEE 1364) and VHDL IEEE STD1076-1993.

Feature File for Enabling and Disabling Various Tool Features

When compiling the design with the IXCOM compiler, you can specify the various features to be enabled or disabled for a particular instance hierarchy, design unit, or HDL file names by using the -featureFile <*feature_file*> option of the ixcom command. The feature file is primarily used to enable or disable assertion and coverage-related features. Refer to the Enabling and Disabling Features at Compilation Time section for details of feature file format.

Using the -featureFile option, you can enable or disable assertion support in specific design units or instance:

© 2021 All Rights Reserved.

Using this option, you can enable or disable:

- Instance hierarchy - for example, `top.FOO1`, and optionally, the entire hierarchy under it.
- Module hierarchy - for example, module `boo`, and optionally, all the modules instantiated by it.
- HDL file - for example, all the modules in the `file2.v` file.

Note that the options mentioned in the feature file have higher priority than command-line options.

Memory Pattern Files

The Netlist Importer can import memory cell instances, described in Structural Verilog into the emulation design database. Typically, these instances are black boxes, and the Compiler generates cell definitions for them.

During an import session, the Netlist Importer attempts to automatically detect these cells as memory instances, by comparing their names against cell name patterns within the ASIC library pattern files. The Netlist Importer applies defaults for the target architecture and optimization parameters.

If the user maps the design to a custom ASIC library, then the user needs to create a *memory pattern* file, and select this file in the Netlist Importer. Otherwise, the Netlist Importer might not be able to detect the cell as a memory cell, and the import process might fail.

The next sections describe the pattern file format, pattern file rules, acceptable values, and syntax. Examples show how to define user-defined memory patterns, which can be provided to the Netlist Importer.

Pattern File Format

A pattern file consists of one or more cell name patterns. A pattern consists of a cell name definition line followed by one or more terminal or pin name definition lines. Cell, terminal, and pin name definition lines begin with a keyword `CELL`, `TERMINAL`, or `PIN`, and end with a new line. Importing ignores any line in a pattern file that does not begin with `CELL`, `TERMINAL`, or `PIN` (`TERMINAL` and `PIN` are synonymous).

© 2021 All Rights Reserved.

Pattern File Rules

The pattern file uses a regular-expression syntax. The pattern file syntax uses the percent sign (%) as an escape token. Wildcard fields match alternative forms of the cell, terminal, or pin names.

The pattern name rules are as follows:

- Any character except %, and except a character that follows a % character, matches itself.
- %d means any digits—for example, 512 or 128. This is a *numeric field*; its value is the numeric value of the number.
- % followed by a digit is a reference to the corresponding regular expression field in the name, on the same line.
- %. means one character; it matches any single character. It does not assume a value, and the user cannot reference it.
- %* means zero or more characters. It does not assume a value, and the user cannot reference it.
- %% means a literal percent sign; it matches the percent sign itself. It does not assume a value, and the user cannot reference it.
- %{...%} surround a *literal set* that means any single character in this set. Its value is the character that it matches.
- %[...%] surround a *positional set* that means any single character in this set. Its value is the *position* of the character that it matches, as opposed to its literal value.

Positional Sets

Use a positional (reference) set if the value of the field should to be expressed as the position number of the field, or if the field should return a value whenever no character matches. Within a positional set, the space character matches any character not matched by any other character in the set. Therefore, the space character can match the null character.

Literal Sets

Use a literal set if the value of the field should be expressed as the exact characters the user entered as the value.

© 2021 All Rights Reserved.

Using Wildcards

A user can write memory patterns explicitly, or can use wildcard expressions to describe parameters implicitly. The `%d` wildcard matches any integer, and the user can use it in the `CELL` and `TERMINAL` (or `PIN`) definition lines. When the user uses wildcard expressions, multiple terminals can be described on one terminal definition line. For example, the `ROM16X2` pattern name matches only a ROM named ROM16X2 (which might define a 2-bit wide by 16-word deep ROM), but the `ROM%dX%d` pattern matches any ROM name that starts with ROM and contains the letter X (such as ROM8X64).

The `%d` expression matches any integer. For example, for a cell named `ROM <integer1>X<integer2>`, using `%d` matches one or more digits. The Netlist Importer interprets all letters as a literals, so the letter `X` matches itself.

Defining Terminal Properties

For each terminal/pin on the cell, define its `Name`, `Type`, `Direction`, `Bit`, and `Port`. For example, the pattern file can describe pins named `A[integer1]` as `A[%d]`.

Supported terminal types (`TYPE=`) are: `ADDRESS`, `DATA`, `BYTESELECT`, `WE` (Write Enable), and `RE` (Read Enable). The Netlist Importer reads the first one or two letters, regardless of case.

Direction (`DIR=`) can be any one of the following: `I` (In), `O` (Out), `R` (Read), `W` (Write), or `RW` (Read/Write).

Bit numbering, `BIT=%1` sets the bit number of a terminal within a port. The `%1` expression refers to the number that matches `%d` in the previous expression (the expression that defined the bit on terminal A). Use this variable for each terminal of a multi-bit bus.

Port numbering (`PORT=1`) can begin with any number, but the Netlist Importer normalizes it to `1`. The `PORT` keyword is optional for single-port memory cells.

The following example defines multi-port memory cells with five ports:

```
CELL    RR%dX%d%*                ARCH=CMM OPTI=SMALL SEP=_
PIN     %[ABCDE%]ADR%d PORT=%1 BIT=%2 TYPE=Addr DIR=RW
PIN     D%[IO%]%[ABCDE%]%d PORT=%2 BIT=%3 TYPE=Data DIR=%1
PIN     WE%[ABCDE%]%[ B%] PORT=%1 TYPE=W POLARITY=%2
```

Pattern File Values

`CELL`, `TERMINAL`, and `PIN` name definition lines (`TERMINAL` and `PIN` are synonymous) can include parameters that accept the following values:

- literal (matches words or numbers)

© 2021 All Rights Reserved.

- reference, or positional (extracts the actual value from the corresponding field in the name pattern)

When using a literal value in the pattern file, notify the Netlist Importer to pass that value immediately.

When a reference value is used, the Netlist Importer interprets the actual value from the expression for the name. It interprets the name starting from the left-most field proceeding to the right.

The Netlist Importer reads the first one or two letters of a word. Therefore, the user can abbreviate values, such as the `ADDRESS` value for the parameter `TYPE` as follows:

```
TYPE=a
```

Available optimization trade-offs are `FAST` or `SLOW`, `SPEED` or `SIZE`, and `SMALL` or `LARGE`.

The Netlist Importer respects case—that is, it interprets `NAME`, `Name`, and `name` differently for cell names. However, the Netlist Importer ignores case for parameters or parameter values, so `NAME`, `Name`, and `name` are the same parameters.

The syntax for a cell name definition is very specific, and is follows:

```
CELL <cellname> (<parameters>)
```

Memory File Format

The file format for memories is as follows.

Format:

```
$INSTANCE <memory_instance_name>
$RADIX BIN|HEX|OCT|DEC
$ADDRESS <start_address> <end_address>
    <address_x> <data>
    <address_x+1> <data>
    .
    .
    .
$END
```

where

<memory_instance_name> specifies the hierarchical name of the memory instance.

<start_address> and *<end_address>* are the hexadecimal starting and ending addresses for initialization.

© 2021 All Rights Reserved.

<address_x> and *<data>* are the address-data pairs. Addresses are hexadecimal and do not have to be contiguous. Palladium fills unspecified addresses with 0s.

The radix options BIN, HEX, OCT, and DEC apply to the data fields only. The address range and the address fields are hexadecimal values.

For example:

```
$INSTANCE U1__RAMBLK0
$RADIX BIN
$ADDRESS 0 5
    0    11011011
    1    00001111
    2    11011011
    3    00100010
    4    11001110
    5    00101010
```

Note: Lines in the memory content file starting with a semicolon (;), dash (-), or hatchmark (#) are treated as comments. All characters following these characters on the line will be ignored.

Note: Memory content files in real binary (raw) format must have a `.bin` extension. Do not use this extension when specifying the memory content filename for a memory read/write operation. Palladium will append the `.bin` extension to the specified filename before executing the read/write operation.

Binary Memory File Formats

A binary memory file format can speed up the `memory -load` and `memory -dump` commands significantly as compared to text-based formats.

Two types of binary memory file formats are available: raw and memTran format. Both can significantly improve performance.

Raw Format

A raw format file can be generated through memory dump, using the following command:

`memory -dump [%pd_raw | %pd_raw2] <memory_name> -file <file_name>`

This raw format file cannot be viewed. It can only be used during next `memory -dump`.

The `memTran` utility can translate between raw format and ASCII format by adding the `-raw` option to the `memTran` command line.

Table 4-2 shows the raw format of the binary memory file.

© 2021 All Rights Reserved.

Table 4-2 Binary Memory File Format

Offset	Description	Value
0-5	signature	QTMEM\0
6	major version	1
7	minor version	0
8-11	reserved	0X12345678
12-47	reserved	0
48-51	depth of memory instance	Depends on the specific memory instance.
52-55	width of memory in number of bits	
56-59	number of bytes for each memory word. Should be calculated as (memory_width+7)/8	
60-63	start address	
64-67	end address	
68-71	instance name length	
72-xx	instance name	
yy-zz	memory data (left justified)	

The memory data are aligned to the byte boundary for each line of data. The bits of each line are left-justified in the bytes, that is, the most significant bit of the memory is the most significant bit of the first byte.

For example, if memory width is 50 bits, then each line will occupy 7 bytes. Bit 49 of the memory is bit 7 (that is, 0x80) of the first byte. Bit 48 of the memory is bit 6 (that is, 0x40) of the first byte. Bit 0 of the memory is bit 6 (i.3. 0x40) of the 7th byte.

memTran Format

By using the memTran utility, you can translate between ASCII format files and binary format files, as well as other file types. The generated binary format file can then be used during the `memory -dump` to improve performance. The memTran format has extra overhead, which could result in larger file sizes than using raw format.

© 2021 All Rights Reserved.

The memTran format is better than the raw format. Use memTran format in general cases.

Use of raw format is most desirable for backward compatibility and publishing the raw format. You can write a C program to generate/analyze raw format.

memTran format can also be used to encrypt memory files. The encrypted file can then be used to perform a `memory -dump`, but it cannot be dumped unless you use a correct password.

The UNIX `memTran` command translates to/from memTran format.

The following examples show use of the `memTran` command.

To convert to a binary format, use the command:

```
memTran <ascii_file> <bin_file> bits [-depth <n>][-p <password>]
```

Here:

`bits` specifies the width of memory.

`-depth <n>` specifies the depth of memory. By default, the depth is derived from the `$ADDRESS` line of the ASCII file.

To dump the contents of the binary file, use the command:

```
memTran -d <bin_file> [-p <password>]
```

You can also compare two binary format files in the memTran tool using the following command:

```
memTran -compare <binfile1> <binfile2> [-p <password>] [-summary]
```

Here:

- `-p <password>` specifies a password (if the binary file is encrypted and a password is required.) User can only specify one password. If either of the two files use the specified password, only one of them needs the password.
- `-summary` notifies the program to print out only a one-line message saying whether the files are same. If `-summary` is not specified, verbose messages are printed out for all the differences.

Cycle-By-Cycle (CBC) Format

CBC format is used in PTRAN vector support.

© 2021 All Rights Reserved.

Cycle-By-Cycle (CBC) vectors contain a time stamp and signal values at fixed time intervals, as opposed to Print-on-Change (POC) vectors that change at irregular increments. To support CBC-format vectors in PTRAN, use the SIMULATOR quest statement.

Use CBC vectors to validate the emulation model. When you run Vector Debug on a design, the CBC vectors stimulate an emulation model of that design. Vector Debug uses the emulator's instrumentation to provide the inputs and clocks to the model.

For design input terminals, the emulator's instrumentation drives the emulated design. The possible input signal values are:

- 0 - LOW
- 1 - HIGH
- X - don't care

For 0 and 1, the emulator's instrumentation monitors the design input terminals for a mismatch that can occur in case of a contention (although a contention does not guarantee a mismatch). For X, the emulator's instrumentation can drive either (0 or 1) value without checking for mismatches.

For design output terminals, the emulator's instrumentation listens and checks for mismatches. The possible output signal values are:

- L - LOW
- H - HIGH
- Z - don't care

L and H are expected values. For Z, the emulator's instrumentation is tristated and does not check the value.

The following is an example of a CBC file:

```
pieVector .0;
vectorFormat CBC;
vectorType stimulus;
numSignal 10;
numVector 8;

PINS CLOCK RESET DATA[7] DATA[6] DATA[5] DATA[4] DATA[3]
DATA[2] DATA[1] DATA[0] ;

// CRDDDDDDDD
// LEAAAAAAAA
// OSTTTTTTTT
// CEAAAAAAAA
// KT76543210
> 19 - XXXXXXXXXX;
> 199 - 10LLLLLLHH;
```


© 2021 All Rights Reserved.

```
> 239 - 10LLLLLHLL;
> 319 - 10LLLLLHLH;
> 419 - 10LLLLLHHL;
> 539 - 10LLLLLHHH;
> 679 - 10LLLLHLLL;
> 839 - 10LLLLHLLH;
<
```

The header information shows the version of the CBC format, the vector format (CBC), the vector type, the number of signals, and the number of vectors.

The signal names (PINS) are associated with the data value for each cycle. The comment field (//) is optional. In this example, the comment field contains the same signal names (PINS), read vertically.

The CBC values show the time stamp and vector data values, indicated by the greater than (>) symbol at the left side of the line. Each data section in the file ends with a semicolon (;).

This example has two inputs and eight outputs.

The two last lines at the end of the file, each containing a single less than (<) symbol, mark the end of the data file. The emulator ignores the time stamp (second token on a data line), which indicates the real time in nanoseconds.

Clock Domain Crossing (CDC) Format

Each line of the CDC file describes one synchronizer. Buses must be split into individual bits. Lines that start with a hash (#) are comments, which are ignored by the software along with empty lines.

Optionally, the CDC file can start with a line of the following format that specifies the CDC version number:

```
set cdc_version <number>
```

Valid version numbers are `0` (default) or `1`. Differences between the two versions are discussed below in the corresponding paragraphs.

Each line that is not a comment must contain the following four tokens:

```
<synchronizer_name> <hierarchical_path_to_synchronizing_reg> <clock>
<control>
```

<synchronizer_name>	User-defined name without spaces, given just for readability. It is ignored by the software.

© 2021 All Rights Reserved.

<hierarchical_path_to_synchronizing_reg>	Path defined according to Verilog standard. It can contain spaces for escaped names. It can start from the design top module (Verilog style), or start one level down (VXE style). The name of the synchronizer can be appended with a `_reg` suffix. However, this is not a must because the software accepts the names with and without this suffix. If the CDC version number is `0`, the synchronizing register must be a flip-flop. However, for version `1`, the synchronizing register must be either flip-flop or latch.
<clock>	Name of the clock for the synchronizer. If this name contains a dot, it is treated as hierarchical path to clock. If there's no dot, the name is treated as a local name in the same hierarchical node as the synchronizer. When CDC version number is `0` and the clock name contains a dot, it is treated as hierarchical path to the clock. On the contrary, if there is no dot, the name is treated as local name in the same hierarchical node as the synchronizer. When CDC version number is `1`, the clock must be a register or wire local to the `cdc_control` module (a so called virtual clock). If this register or wire is not found, the synchronizer is ignored (that is, delay is not inserted). Multiple delays can be controlled by the same clock.
<control>	Name of the delay control. This must be the name of the register or wire local to the `cdc_control` module. If this register or wire is not found, the synchronizer is ignored (that is, the delay is not inserted). Many delays can be controlled by the same control.

For an example of a CDC file illustrating the use of the four tokens, refer to Example CDC File: 1.cdc.

© 2021 All Rights Reserved.

5

Using Utilities

These utilities have been developed for mixed-HDL simulation:

- **cds2libmap**
- **vhdel**
- vhls
- **vhmake**
- **vhshell**

cds2libmap

Purpose

The `cds2libmap` utility converts a `cds.lib` file to an equivalent libmaprc file. This utility can also be used to convert IXE specific `va.lib` files to equivalent libmaprc files. See Converting cds.lib Files to libmaprc Format on page 621 for more information.

Syntax

```
cds2libmap [-h] <cdslib_file> [-o <output_file>]
```

Parameters

`-h`	Prints help for this utility.
<cdslib_file>	Specifies the `cds.lib` file to be converted to libmaprc file

© 2021 All Rights Reserved.

`-o <output_file>`	(Optional) Name of the output (libmaprc) file. If no output file is specified, the `cds.lib` file is converted to a libmaprc file with the default name `<cdslib_file>_libmaprc`

vhdel

Purpose

The `vhdel` utility lets you delete one or more VHDL or Verilog design units from an analyzed library.

Syntax

```
vhdel -work <library> -e <entity> -a <architecture> | -p <package> | \
      -b <package_body> |-c <configuration> -v <Verilog_module>
```

Parameters

`-work <library>`	Specifies the analyzed library (default **WORK**) from which to delete design units.
`-e <entity>` `-a <architecture>`	Specifies the entity and/or architecture to delete. If the architecture is not specified, the entire entity is deleted; otherwise, only the specified architecture for that entity is deleted.
`-p <package>`	Deletes the specified package and its package body.
-b `<package_body>`	Deletes only the specified package body for the specified package.
`-c <configuration>`	Deletes the specified configuration.
`-v <Verilog_module>`	Deletes the specified Verilog module from the library.
`-libmaprc <file>`	Specifies a file containing logical-library-to-physical-directory associations, as required by mixed-HDL tools. See Associating Logical Libraries to Physical Directories Using libmaprc File on page 617 for more information.

© 2021 All Rights Reserved.

vhls

Purpose

The `vhls` utility lists the design units (Verilog or VHDL) compiled into one or more libraries and provides information on these units.

Syntax

```
vhls [-h] [-work <library>] <design_unit> [-e|-a|-p|-b|-c] [-l] [-s] \
     [-dh] [-dhf] [-dhoutputfile <file>]
```

Parameters

`-h`	Prints help for this utility.
`-work` <*`library`*>	(Optional) Limits the scope of the utility to the logical library specified. Without this option, the utility operates on all libraries in the `libmaprc` file.
<*`design_unit`*>	(Optional) Lists information for the specified <*`design_unit`*> only in <*`library`*>. If <*`design_unit`*> is not specified, all design units in the library or libraries are listed.
`-e|-a|-p|-b|-c`	Lists all design units of the specified type in the compiled library(ies): ■ `-e` specifies all entities ■ `-a` specifies all architectures ■ `-p` specifies all packages ■ `-b` specifies all package bodies ■ `-c` specifies all configurations
`-l`	Provides additional information on listed design units, such as logical library, source-file path, line number, compilation status (up-to-date or not), and last-compiled date.

© 2021 All Rights Reserved.

`-s`	By default, the `vhls` utility does not provide information for the STD or IEEE system libraries. Use this option to list the packages and package bodies in these libraries. The physical directory paths for these libraries are listed in the `libmaprc` file. See Associating Logical Libraries to Physical Directories Using libmaprc File on page 617 for more information on the `libmaprc` file.
`-libmaprc` <*file*>	Specifies a file containing logical-library-to-physical-directory associations, as required by mixed-HDL tools. See Associating Logical Libraries to Physical Directories Using libmaprc File on page 617 for more information on the `libmaprc` file.
`-dh`	Prints the full hierarchical path of an instance in the following pattern: <*libName*>.<*moduleName*> <*top*>.<*instanceName*>.<*instanceName*>.<*instanceName*> Refer to Printing Full Hierarchy of Paths with -dh and -dhf Options - Example to see an illustration of the actual output generated with the use of the `-dh` option. The informative messages and error logs are sent to the stdout and the default file: `vhls.log`. The actual output is written in the <*file*> specified with the `-dhoutputfile` option.If the `-dhoutputfile` option is not specified, the default name of the file to store the actual output of the `-dh` option is `vhls_moduleInstance.list`.

© 2021 All Rights Reserved.

`-dhf`	Prints the full hierarchical path of an instance with bind information of a module from the specific library in the following pattern:
	
	`<libName>.<moduleName>`
	`<top>.<instanceName>(<libName>.<moduleName>).`
	`<instanceName>.(<libName>.<moduleName>).<instanceName>`
	
	Refer to Printing Full Hierarchy of Paths with -dh and -dhf Options - Example to see an illustration of the actual output generated with the use of the `-dhf` option.
	
	The informative messages and error logs are sent to the stdout and the default file: `vhls.log`. The actual output is written in the `<file>` specified with the `-dhoutputfile` option.If the `-dhoutputfile` option is not specified, the default name of the file to store the actual output of the `-dhf` option is `vhls_moduleInstance.list`.

`-dhoutputfile <file>`	Specifies the `<file>` in which the actual output of the `-dh` and `-dhf` options is stored. If this option is not specified, the default name of the file to store the actual output of the `-dh` and `-dfh` options is `vhls_moduleInstance.list`.

Printing Full Hierarchy of Paths with -dh and -dhf Options - Example

Consider the following example. The output generated with the use of `-dh` and `-dhf` options of the `vhls` command for the following example are given below.

```
module bot();
   reg a;
endmodule;

module dut();
   reg q;
   bot b();
endmodule

module top();
   dut d1();
   begin
      dut d2();
   end
   begin: block1
      dut d3();
   end

   genvar n;
   generate
      for (n=0; n<=7; n=n+1)
      begin
```


© 2021 All Rights Reserved.

```
         dut d4();
      end
   endgenerate
endmodule
```

Using the `-dh` option with the `vhls` command produces the following output:

```
WORK.dut_xcm2      top.d1
WORK.bot_xcm4      top.d1.b
WORK.dut_xcm3      top.d2
WORK.bot_xcm5      top.d2.b
WORK.dut_xcm3      top.block1.d3
WORK.bot_xcm5      top.block1.d3.b
WORK.dut_xcm3      top.genblk1_0_.d4
WORK.bot_xcm5      top.genblk1_0_.d4.b
WORK.dut_xcm3      top.genblk1_1_.d4
WORK.bot_xcm5      top.genblk1_1_.d4.b
WORK.dut_xcm3      top.genblk1_2_.d4
WORK.bot_xcm5      top.genblk1_2_.d4.b
WORK.dut_xcm3      top.genblk1_3_.d4
WORK.bot_xcm5      top.genblk1_3_.d4.b
WORK.dut_xcm3      top.genblk1_4_.d4
WORK.bot_xcm5      top.genblk1_4_.d4.b
WORK.dut_xcm3      top.genblk1_5_.d4
WORK.bot_xcm5      top.genblk1_5_.d4.b
WORK.dut_xcm3      top.genblk1_6_.d4
WORK.bot_xcm5      top.genblk1_6_.d4.b
WORK.dut_xcm3      top.genblk1_7_.d4
WORK.bot_xcm5      top.genblk1_7_.d4.b
```

Using the `-dhf` option with the `vhls` command produces the following output:

```
WORK.dut_xcm2      top.d1
WORK.bot_xcm4      top.d1(WORK.dut_xcm2).b
WORK.dut_xcm3      top.d2
WORK.bot_xcm5      top.d2(WORK.dut_xcm3).b
WORK.dut_xcm3      top.block1.d3
WORK.bot_xcm5      top.block1.d3(WORK.dut_xcm3).b
WORK.dut_xcm3      top.genblk1_0_.d4
WORK.bot_xcm5      top.genblk1_0_.d4(WORK.dut_xcm3).b
WORK.dut_xcm3      top.genblk1_1_.d4
WORK.bot_xcm5      top.genblk1_1_.d4(WORK.dut_xcm3).b
WORK.dut_xcm3      top.genblk1_2_.d4
WORK.bot_xcm5      top.genblk1_2_.d4(WORK.dut_xcm3).b
WORK.dut_xcm3      top.genblk1_3_.d4
WORK.bot_xcm5      top.genblk1_3_.d4(WORK.dut_xcm3).b
WORK.dut_xcm3      top.genblk1_4_.d4
WORK.bot_xcm5      top.genblk1_4_.d4(WORK.dut_xcm3).b
WORK.dut_xcm3      top.genblk1_5_.d4
WORK.bot_xcm5      top.genblk1_5_.d4(WORK.dut_xcm3).b
WORK.dut_xcm3      top.genblk1_6_.d4
WORK.bot_xcm5      top.genblk1_6_.d4(WORK.dut_xcm3).b
WORK.dut_xcm3      top.genblk1_7_.d4
WORK.bot_xcm5      top.genblk1_7_.d4(WORK.dut_xcm3).b
```


© 2021 All Rights Reserved.

vhmake

Purpose

To manage the compilation of VHDL, Verilog, or mixed-HDL designs, `vhmake` or generates a UNIX `Makefile` with:

- The paths to the original VHDL and Verilog sources and the proper order in which to compile them.
- An incremental compile feature—the `Makefile` recompiles only sources that have changed and other sources that have dependent design units.

Syntax

```
vhmake [-h] [-work <library>] <design_unit> [-d <design_unit>] > Makefile
```

Parameters

`-h`	Prints help for this utility.
`-work` <*`library`*>	The logical library of a top-level design unit if one is specified; otherwise, it is the library for which you want to generate a `Makefile`.
<*`design_unit`*>	The name of the design unit (VHDL entity or configuration or Verilog module) for which you want to generate a `Makefile`. The `Makefile` is generated for the entire hierarchy below the specified design unit.
`-d`	The dependency listing feature. You must specify <*`design_unit`*> with this option. For each design unit, the following information is given: dependent design units and their source files and line numbers.

Description

`vhmake` has a dependency listing feature (`vhmake -d`). Starting with the highest level entity or configuration, the design units of a particular design are listed in order of dependency:

- For each package body, the dependent package header is listed.

© 2021 All Rights Reserved.

- For each entity, the dependent packages are listed.
- For each architecture, the dependent entities are listed.
- For each configuration, the dependent entities and architectures are listed.
- For each design unit, the corresponding source file and line number where the unit is defined are listed.

Note

- To execute the generated `Makefile`, run the UNIX utility `make`.

vhshell

Purpose

The `vhshell` utility is used to generate a VHDL **shell** file and component declaration for a Verilog module.

Important

The `vhshell` utility is not required with this release. The DUT elaborates without the shell. The description of this utility has been provided for backward compatibility.

Syntax

```
vhshell [<module> | -all -o <output_file>| -multi_file] [-comp] \
        [-arch <architecture>] \ [-work <library>][-f <options_file>] \
        [-u] [-g] [-xmvhdl]
```

Parameters

`-comp`	Causes only the component declaration to be generated to either `<module>.inst` or as specified with the `-o` option.

© 2021 All Rights Reserved.

`-arch` <*architecture*>	Lets you change the default architecture name of the specified <*architecture*> in the generated shell file. This is useful if the shell file is used to replace a VHDL entity with a Verilog module, and you would like to use the original configuration for the VHDL entity. This switch cannot be used with the `-all` or `-f` options
module | `-all` `-o` <*output_file*> | *-multi_file*	Enables you to specify the Verilog module name(s) from which to extract shell files: To extract a single shell, specify just the <*Verilog_module*>. Use the `-f` <*options_file*> option to specify multiple module names to extract. Two files are generated for each module: <*module*>.vhd, which is the shell file, itself, and <*module*>`.inst`, which contains a VHDL component declaration for the Verilog module. To extract all modules from a library, use `-all` with either the `-o` <*output_file*> or `-multi_file` option. If `-o` <*output_file*> is specified, all shells go into a single file, <*output_file*>`.vhd`, and all component declarations go into the file, <*output_file*>`.inst`. If `-multi_file` is specified, `vhshell` generates a file, <*module*>`.vhd` (the shell file itself), and a file, <*module*>`.inst` (containing a VHDL component declaration), for each Verilog module.
`-work` <*library*>	Specifies the compiled Verilog library from which to extract shell file(s) (`WORK` is the default).
`-f` <*options_file*>	Specifies a file with additional options and module names to extract.
`-u`	Lets you specify `STD_ULOGIC` as the port type, instead of the default of port type `STD_LOGIC`.
`-g`	Suppresses generation of VHDL generics for Verilog parameters, instead of the default in which they are generated.

© 2021 All Rights Reserved.

`-xmvhdl`	Generates Cadence XMVHDL-compatible shell files, which is required if you want to use the shell files with the Xplore graphical user interface.

Notes

- In most cases, the **shell** is not required when instantiating Verilog in VHDL, except if you have a VHDL configuration that refers to a Verilog component.
- When analyzing VHDL shell files, do not pass the `-87` option to `vhan`. Shells require VHDL 1993 compatibility. But if you are using the `-mti` option, be sure to pass the `-93` option.

Example 5-1 VHDL shell File Generated for a Verilog Module

Example 5-2 Translating Verilog Parameters to VHDL Generics

Generics in the shell file are created based on the *initial* values of the Verilog parameters in the module:

- String initial values: Generates generics of type `string`. For example, for the following Verilog initial value:

```
parameter input_file="sram_input.dat";
```


© 2021 All Rights Reserved.

The shell generic is:

```
generic input_file: string := "sram_input.dat";
```

- Integer and real initial values: Generates generics of type `integer`, `real`. For example, for the following Verilog initial value:

  ```
  parameter RC_Const=1.1;
  ```

  The shell generic is:

  ```
  generic RC_Const: real := 1.1;
  ```

- Binary-, hex- or octal- formatted initial values: Generates generics of type `std_logic_vector` with the appropriate size. For example, for the following Verilog initial value:

  ```
  parameter init=8'hFF;
  ```

  The shell generic is:

  ```
  generic init: std_logic_vector(7 downto 0) := "11111111";
  ```

- For all other initial values, the default is a shell generic of type `integer`.

Example 5-3 Using Vector Ports Constrained by Parameters

The `vhshell`utility generates an unconstrained VHDL port in the shell file if the associated Verilog port has its size constrained by parameters. The VHDL port will be constrained appropriately when the component is instantiated in the VHDL design, for example:

```
parameter addr_width=8;
input [addr_width-1:0] ADDR;
```

The VHDL shell file will contain the following port:

```
port(ADDR: in std_logic_vector);
```


© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

6

Working with the Palladium Z1 Emulators

This chapter includes the following sections:

- Configuring the Emulator
- Working with Emulators at Compile Time
- Working with Emulators at Run Time
- Compute Farm Control Mechanism
- Balancing Load Using Load Sharing Facility
- Tracking Emulator Jobs
- Checking Emulator Status

Configuring the Emulator

Following are the mandatory requirements for an emulator:

- The Palladium Z1 software and supported operating system should be installed on the specified host.
- The design directory should be accessible to the specified host by using the same path as the workstation on which you run xeDebug or xeCompile.
- The specified host should be able to accept either `rsh` or `ssh` remote execution command.
- The directory `/et3mach` on the host needs to be readable by the workstation on which xeCompile and xeDebug are run. For Palladium Z1 the `/et3mach` directory is maintained on the System Control Drawer (SCD) and exported to enable various Linux autoimmunity capabilities in accessing the directory structure. It is available on NFS, such as `/net/<host_name>/et3mach`.

© 2021 All Rights Reserved.

Working with Emulators at Compile Time

At compile time, you can work with the emulator in the following ways:

- Identifying Design Host and Domains
- Compiling the Designs for Best Performance
- Compiling the Designs for Best Relocation
- Specifying the Connection to Target System
- Working with Emulators at Run Time

Identifying Design Host and Domains

After opening the design, specify the emulator and domains that must be used for compilation, as well as the compile mode with which the design is to be compiled. The supported compile modes are Logic Analyzer (LA), Static Target TestBench (STB), and Vector Debug (VD).

To set up the emulator for compiling the design, use the `emulatorConfiguration` XEL command. The `emulatorConfiguration` XEL command generates a configuration file named <*top_cell*>.et3confg in the `./dbFiles` directory, where the <*top_cell*> is the top cell name specified in the `design` XEL command. The <*top_cell*>.et3confg configuration file is used during the back-end compilation and run-time and debug sessions.

The syntax of the `emulatorConfiguration` XEL command is as follows:

```
emulatorConfiguration -add {host <hostname>} [{boards <sub_board>}]
```

There is no restriction on the specification of a <*hostname*>. The <*hostname*> can be any string. If you execute the `emulatorConfiguration -add {host ../myhost}` command, the string "`../myhost`" is used as a host name.

Note: When compiling a design in SA mode, you can alternatively use the environment variables — `XE_HOST`, `XE_BOARDS`, and `XE_DYNP` to set the emulator configuration. To specify the emulator for which you are compiling the design, set the `XE_HOST` environment variable. To specify the logic drawers and domains for compilation, set the `XE_BOARDS` environment variable. To specify the DYNP compilation mode, set the `XE_DYNP` environment variable.

- If `XE_BOARDS` is specified, but not `XE_HOST`, the current host is used.
- If `XE_HOST` is specified, but not `XE_BOARDS`, domain `0.0` is used.

© 2021 All Rights Reserved.

- If neither XE_HOST or XE_BOARDS is specified, a generic single domain configuration is used.

You can also use the XE_HOST_FILE environment variable to specify the full path to the *.et3confg file instead of the emulator.

For example,

```
setenv XE_HOST_FILE /home/user/scratch/emu.et3confg
```

To use XE_HOST_FILE you need to unset the XE_HOST environment variable if it is already set.

Specifying the Emulator Name

In SA mode, use the XE_HOST environment variable to specify the name of the emulator.

If you do not set the XE_HOST environment variable, it's value is set to the local workstation as the virtual host to derive the emulator to compile and run the designs. The local workstation must be connected to the emulator on the InfiniBand (IB) network and have the /et3mach installation.

The XE_HOST environment variable enables you to select the emulator for both compiling and running a design.

Note: The XE_HOST environment variable is also used while running a design in SA mode with *xrun*, whereas the host command is used while running a design in xeDebug.

Specifying the DBEngine Host

Use the DBE_HOST environment variable to specify the DBEngine host.

When DBE_HOST is set before running a design in SA mode with *xrun*, the Simulation Acceleration software automatically issues the following command:

```
host . -dbengine $DBE_HOST
```

If you do not set the DBE_HOST variable, it's value is the local workstation from where the xrun command is issued. The local workstation must be connected to the emulator on the InfiniBand network.

The DBE_HOST environment variable has no effect on a compilation process.

Setting both XE_HOST and DBE_HOST environment variables with *xrun* is same as issuing the following command in xeDebug:

```
host $XE_HOST -dbengine $DBE_HOST
```


© 2021 All Rights Reserved.

Conserving Logic Drawer Resources

Ideally, a design uses a full logic drawer. The following emulatorConfiguration command specifies the use of the entire emulator:

```
emulatorConfiguration -add {host <hostname>}
```

However, to allow multiple users and/or conserving logic drawer resources, smaller designs can be fitted into subsets based on logic drawer domains. A logic drawer domain is a subset of the emulation modules on a logic drawer.

If the design does not require the complete set of host resources, use the following command to specify the subset of the emulator to be used:

```
emulatorConfiguration -add {host <hostname>} {boards <sub_board>}
```

When the VXE system is first installed, diagnostics are run and all hardware resources are detected and verified. When diagnostics have run successfully, a `<hostname>.et3confg` file is created in the `/net/<hostname>/et3mach` directory of the host workstation. This file is used to specify the VXE resources for compiling and downloading. For example, if the host name is `ninja`, the configuration file will be named `ninja.et3confg`.

When compiling, you can specify a configuration file rather than a host name using the following command:

```
emulatorConfiguration -add {file <filename>} {boards <sub_board>}
```

This command is especially useful if the design needs to be compiled for an emulator that has not yet been installed.

On the same logic board, multiple designs can be compiled by using different domains on that logic board. During run time, compiled designs can be forced to use only specific domains by using a `<design>.bp` file, where the `<design>` is same as the `<top_cell>` name mentioned earlier with the `.et3confg` filename. The `*.bp` file can also be used to relocate a design into domains of different logic drawers based on the available space. For details, refer to Using a .bp File for Resource Reservation.

Note: The compiler compiles for ascending order of domain, irrespective of the order you specified to `emulatorConfiguration`. This may result in unexpected failures during download.

Use the `hwInfo` XEL command to display the detailed configuration information for a specific emulator, configuration file, or design. This information is useful in studying the hardware system or subsystem and its connectivity.

For more information on the syntax and usage of the `emulatorConfiguration` and `hwInfo` commands, refer to the *User Data Commands* chapter of the *VXE Command Reference Manual*.

© 2021 All Rights Reserved.

Creating BP File for Designs with Target

If a design has a target, you can use the `host -createTargetBp` command to create a `.bp` file based on the domains specified during the compilation process. If the `host -skipAllTargets` command is specified, the `host -createTargetBp` command is ignored and no `.bp` file is created.

Compiling the Designs for Best Performance

You can choose to compile the designs for improving relocation capability or for enhancing the performance. By default, designs are compiled for better relocation capability.

To compile the design for achieving best performance, use the `connectivityOptimizationGoal bestPerformance` option of the `compilerOption` command as follows:

```
compilerOption -add {connectivityOptimizationGoal bestPerformance}
```

Designs compiled for best performance yield a lower step count as compared to designs compiled for better relocation. A low step count improves the performance.

Note: Compiling the designs for best performance reduces the number of relocation options at run time. On the other hand, when you compile designs for best relocation, more relocation options are available at run time.

For example, consider a two-domain compile request. For 0.0+0.1 with delay D1, and actual wire delays on the emulator to be:

```
0.0 <-> 0.1 = D1;
0.1 <-> 0.2 = D2;
```

In the above specification, the delay values are not normalized. So, the design (`0.0+0.1`) can be placed only on `0.0+0.1`. The `bestPerformance` option contradicts the `bestRelocation` option. The `bestRelocation` option aims to find more relocation options to better utilize the emulator.

Compiling the Designs for Best Relocation

The `bestRelocation` option normalizes the hierarchical-level interconnect delays on the emulator when compiling a design. For example, all domain wires within an LD are normalized to have the same delay value. Similarly, all of the LD wires within a cluster and all of the LD wires outside the cluster within a rack are normalized. Thus, this normalization increases the relocation options for a particular design.

© 2021 All Rights Reserved.

Consider a two-domain compile request for `0.0+0.1` with wire delay D1 between these (`0.0 <-> 0.1 = D1`) . In addition, consider three domains in the emulator (`0.0, 0.1, and 0.2`) with wire delays as follows:

```
0.0 <-> 0.1 = D1;
0.1 <-> 0.2 = D2;
```

The `bestRelocation` option normalizes the wire delays to:

```
0.0 <-> 0.1 = DN1;
0.1 <-> 0.2 = DN1;
```

Here, `DN1 = D1`.

Now, based on the normalized delay values, the given design (`0.0+0.1`) can be either placed on `0.0+0.1` or `0.1+0.2`.

Specifying the Connection to Target System

The design-preparation step includes specification of the cable-connection parameters that need to be used with a target system. A target is an external circuit that connects to the emulator. The target parameters include cable types, terminal assignment, and terminal timing commands.

Specifying the Target Connection in ICE Mode

For ICE mode, you need to provide the following information to connect a target system to DUT (mapped onto the emulator), at compile time:

- RTL or netlist for the design
- Design signals that connect to a target interface
- Cable connections between the emulator and the target system (using the `cableConnection` userdata command).
- Interface pins used for the target connection (using the `terminalAssign` userdata command).
- Timing constraints for the terminal connected to the target system (using the `terminalTiming` userdata command).

Refer to Specifying Target Connection in SA Mode on page 122 for details about target specification in SA mode.

The following sub-sections provide details on the use model for connecting the target system to the design mapped onto the emulator by using different approaches (or methods).

© 2021 All Rights Reserved.

Assigning Target-System Terminal

Palladium Z1 supports a target cable carrying 192 logical signals (input, output, or bi-directional). Additionally, various adapters, such as TIB-192 and LTA, can be included. All I/Os from the top-level design unit (or cell) are available for the target connections.

Use the `terminalAssign` XEL command for associating specific pins with a terminal and cable name. Then, specify the terminal-timing constraints by using the `terminalTiming` XEL command. The information passed with the `terminalAssign` commands is used to determine the top-level I/O connection for establishing the target connections. Without additional User Data, the internal signals of the design are not available for connection to a target interface.

For TIB connections, the *<pin>* parameter is the combination of the TIB connector name and a valid pin number on the TIB connector. Valid TIB connector names are P2, P3, P4, P5, P6, P7, and P8.

Establishing Target-Cable Connection

When the target cable is first connected and powered on, both LEDs will glow amber. Once the FPGA is successfully programmed, the two LEDs function as described by their labels to indicate power status.

To program the cable connections, use the `cableConnection` command that has the following syntax:

```
cableConnection -add {<cable_label> <phy_loc> {I/O_technology}}
```

The *<cable_label>* is composed of a user-specified interface name and interface type in the format *<interface_name>*.*<interface_type>*. The type corresponds to the target cable type.

The <phy_*loc*> specifies the target connector location in the <*rack_number*>_T<*target_number*> format. The <*rack_number*> ranges from `0` to the maximum rack index in the system. The <*target_number*> is the index of the target physically connected to the target connector on the rack. The target number ranges from 0 to 71.

The *<I/O_technology>* is one of the supported Palladium Z1 technology names. The I/O technology also defines the drive strength of the cable. The default is `LVCMOS33_8`.

Example

The following example shows how to assign cables using the `cableConnection` command.

© 2021 All Rights Reserved.

```
cableConnection -add {<cable_label> <LTA_A|LTA_B> <rack>_T<target> LVCMOS33_16}
```

This adds a cable connection supporting two TIB-88 adaptor connections, one on the A connector of the Legacy Target Adapter (LTA) and one on the B LTA connector.

Assigning Terminal for the TIB-192 Interface

In the `terminalAssign` command, the *<pin>* parameter is the combination of the TIB connector name and a valid pin number on the TIB connector. For information about the various connectors and valid pin values for the TIB-192 interface, refer to the description of the `terminalAssign` command in the *VXE Command Reference manual*.

For example, to assign `DATA[0]` to `pin 17` of connector `P2` of the interface_type `TIB-192` with interface_name `mycable1`, use:

For example, a TIB-192 terminal assignment can be done as follows:

```
terminalAssign -add {myCable1 P2 17 DATA[0]}
```

If a TIB-192 interface is used, it must be explicitly specified in the *<cable_label>* field of the `cableConnection` command, as shown in the example below:

```
cableConnection -add {myCable3.TIB-192 0_T1 LVCMOS33_16}
```

Providing LTA Support

Use an LTA to adapt an MTCC cable to the target cable. In addition, an LTA can be used to turn a target cable into one or two Palladium Z1 cables of type TCC (interface type name: DATA_CABLE).

Terminal assignment for LTA support requires the following command:

```
terminalAssign -add {<cable_name> DATA_CABLE <pin> TERM | NET
                     <terminal_name>}
```

For LTAs, the *<cable_label>* in the `cableConnection` command should use the filename extension of `.DATA_CABLE`, followed by either LTA_A or LTA_B. These parameters indicate whether the A connector or B connector of the LTA is attached to the interface. The LTA_A or LTA_B parameter is required. For example:

```
cableConnection -add {pdCable1.DATA_CABLE LTA_A 0_T9 LVCMOs15_SERIES}
cableConnection -add {pdCable2.DATA_CABLE LTA_B 0_T9}
```

The hardware can be an extension cable, an adapter, or a connector.

Each of the two LTA connectors can be individually programmed using the `cableConnection` command.

© 2021 All Rights Reserved.

Note: The two connectors might share the same target cable. However, different cable names must be used.

For the complete syntax of the `cableConnection` command, refer to the *User Data Commands* chapter of the *VXE Command Reference Manual*.

Additionally, there are also two sets of `16` pins on either end of the LTA that can be directly connected to a target system using a user-supplied cable. Of these 16 pins, eight are always ground. Collectively, these pins are referred to as the extended cable pins (EXC) and are also programmed using the `cableConnection` command, where they are referred to as EXC LTA_C and EXC LTA_D.

Defining EXC Connection

16 pins are available as individual target interface pins on the LTA. These 16 individual pins are the result of 192 pins minus twice 88 pins allocated to the LTA A and LTA B connectors. These are separated into two groups, with eight pins per group. A group also corresponds to a connector. So, the connector location on the LTA is identified by either `LTA_C` or `LTA_D`. The interface type for these pins is called EXC.

Use the terminal assignment command as illustrated in the following example:

```
terminalAssign -add {myEXCcable EXC 7 Data[5]}
```

A cable connection statement for this special interface would be defined as illustrated in the following example:

```
cableConnection add {myIndividualPins.EXC  LTA_C  0_T2}
```

Multiple Assignments to the Same Terminal

Use the `terminalAssign` command for making multiple assignments to the same terminal. This is useful for handling multiplexed terms. For example, to assign more than one net (`netA`, `netB`, and `netC)` to the same target cable `pin 1` of connector `P2` that has the interface_type `TIB-192` and interface_name `myCable6`, use:

```
terminalAssign -add {myCable6 P2 1 netA}
terminalAssign -add {myCable6 P2 1 netB}
terminalAssign -add {myCable6 P2 1 netC}
```

Note: To make the above command work, it is necessary to include the `multiplexCable -add myCable6` directive.

© 2021 All Rights Reserved.

Synchronizing Output to a Target System Connected by ADC Cables

Groups of primary output signals to a target system connected via ADC cables can be synchronized for more precise timing. This can be useful for signals such as buses, where the same timing constraint can be applied to all signals.

Use the `terminalOutputSync` XEL command to specify one or more groups of primary output signals to be synchronized.

```
terminalOutputSync -add [<terminal> <group-number>}
```

The *<terminal>* signal is the primary output signal to be synchronized with the other primary output signals indicated by *<group-number>*. The *<terminal>* signal must be a valid terminal assignment. Only one statement can be used per command line; therefore, a statement must be issued for each signal for that group number. The actual group numbers are used only to identify the group to which they belong. They do not specify the exact step where the group is to be scheduled; they specify only that the signals having this number be scheduled at the same step. These signals will appear at the target at the same time.

Note: In a group, a signal can be a multi-sampled output only if all other signals of the group are also multi-sampled output.

For example, if the following statements are executed, the signals for `myterm_1` and `myterm_2` will be synchronized as a group, and `myterm_3` and `myterm_4` will be synchronized as another group:

```
terminalOutputSync -add {myterm_1 1}
terminalOutputSync -add {myterm_2 1}
terminalOutputSync -add {myterm_3 2}
terminalOutputSync -add {myterm_4 2}
```

The MakeCTL program will check for syntax correctness, and whether the signals are all multi-sampled or all non-multi-sampled. For each synchronization group, it creates a new section in the *<design>*`.ctl` file containing the lines of primary output signals to be synchronized. For example:

```
*ADC_OUTPUT_SYNC_GROUP
<signal_1>
<signal_2>
<signal_3>
...
```

For multi-sampled signals, a separate shadow synchronization group, consisting of all corresponding shadow signals, is also created. The `bitGen` function will create a *<design>*`.ttime` file, containing the final target TIF steps of all signals.

Every new `ADC_OUTPUT_SYNC_GROUP` will have a different ID number.

© 2021 All Rights Reserved.

Identifying Multisampled Input or Output (I/O)

Multisampled I/O enables you to use multiple asynchronous clock domains to identify portions of a design to implement so that they can be evaluated at a faster rate than other portions of the design.

Some designs have more than one clock domain. Usually, this is handled by operating the fast clock at the least common multiple of the various user clocks, or by establishing a (possibly irregular) clock pattern that accommodates all clock domains. However, in some designs, the combination of maximum path length and clock speed in different clock domains can result in the evaluation of all clock domains with respect to a common fast clock executing more slowly than the slowest clock domain.

By using multiple asynchronous clock domains you can modify clock domains to achieve better performance. These domains are restricted to those that interface to top-level input or output. These clock domains are identified by their top-level connections, and are mapped into the emulator in a modified form that allows them to be evaluated up to twice as fast as other clock domains.

You must recognize the design characteristics that indicate use of the multi-sampled feature. Usually, this is indicated by a combination of unsatisfactory evaluation speed and the design containing a top-level I/O-connected clock domain clocked faster than other clock domains. Keep in mind that the increase in the speed depends on the design.

This feature is intended for expert users. It requires deep knowledge of your design, and how that design is compiled and evaluated on the emulator.

The mechanism that effects acceleration where this feature is applied is not an increase in the frequency of evaluation. Instead, portions of the design are modified such that up to twice as much evaluation is compressed into one cycle. This modification can take as much as two times the capacity the unmodified circuitry would take.

As these portions of the design are effectively evaluated at up to twice the frequency of the rest of the design, the usual sampling mechanism of once per cycle is more complicated. In the sped-up portion of the design, signals are effectively sampled twice per cycle. However, you specify the sampling offset for normal sampling; the additional samples are derived from the normal sampling offset.

Note: This additional sampling complexity can result in anomalous-appearing waveforms. Exercise great care in setting the sampling offset, and understand the resultant data with respect to the specified offset.

Triggering is accomplished relative to normal (non-sped-up) portions of the design. Because of the compressed evaluation of the modified portion of the design, triggering conditions derived from signals in that portion might not conform to expectation.

© 2021 All Rights Reserved.

The `compilerOption -add {enableMultiSampledIO 0|1|2}` and `multiSampledTerminal -add|-rm <terminal_name>` XEL commands turn this feature on and off, and specify the top-level I/O connections to which this feature is to be applied. Multisampling can also be performed on inputs or both inputs and outputs. Refer to the *VXE Command Reference Manual* for more information on the command usage.

Handling Terminal-Timing Constraints

When emulating with a target system, terminal-timing constraints are handled using the `delayBox` and `terminalTiming` XEL commands.

For a top-level I/O terminal which is connected to a target system, terminal-timing constraints control the relative timing of clock and non-clock signals communicated between the emulator and the target, or specifies the behavior of the terminal as a pulse or multi-pulse output.

A delay box notifies the Palladium Z1 compiler to schedule the specified delay steps (TIFs or ns) on a particular output signal. If a delay box already exists, the command modifies the delay, and will add inputs from the specified inputs list to the existing list of inputs (that is, it will not replace the existing inputs list).

For more information on the command syntax and usage, refer to the *User Data Commands* chapter of the *VXE Command Reference Manual*.

Note: VXE 18.1 supports single bit range in timing compiler commands (`delayBox`, `delayBoxInput`, `terminalTiming` and `terminalOutputSync`) by replacing all the bits of the range.

Benefit of Advanced Capabilities of Terminal Timing Constraints

Multiplexed signals are communicated using HDDC cables. This is known as HDDC Muxing. HDDC Muxing takes advantage of the advanced capabilities of terminal timing constraints to use a virtual multiplexer (MUX) logic inside the HDDC cables. Using HDDC Muxing in your design can greatly increase the amount of inputs and outputs transferred across an emulation cable. Use `delayBox` and `terminalTiming` to define the timing relationship between input and output signals that go across the emulation cable to the target. While HDDC Muxing produces long timing chains of input and output signals at the target interface. These timing chains often become critical timing paths in the emulated design. So, designs that use HDDC Muxing might see significant performance improvement when using Pipelined I/O as Pipelined I/O can be used to shorten the Target I/O timing paths which is the longest timing path in the database, thereby increasing the overall emulation performance.

For more information on Pipelined I/O, see section "Improving Step Count for Designs with External Target Interfaces" in *VXE User Guide*.

© 2021 All Rights Reserved.

For more information on using HDDC Muxing, see "Adding FIFO When Using HDDC Muxing" in *VXE User Guide*.

Shaping the Pulse

The Palladium Z1 hardware can generate signals on external connector pins at a frequency higher than that of the emulation fast clock (FCLK). This capability is referred to as pulse shaping. Pulse shaping is useful when the clock of the external target system is controlled by the emulator and has divide-down circuitry as part of its clocking logic.

Any design to be emulated on Palladium Z1 must have one or more special signals you designate as clock signals. These signals are referred to as "design clocks" or "user's clocks"; these terms are synonymous.

A cycle of the design clock is carried out by one or more fast clock cycles of the emulator. For some designs, a full cycle of the design clock can be emulated in just one cycle of the fast clock. Other designs require two or more cycles of the fast clock to emulate one cycle of your design clock.

The cycle of a fast clock is subdivided into emulation steps. The emulation steps are controlled in the hardware by the step clock. The step clock has a cycle time of two nanoseconds.

xeCompile determines the number of steps required to emulate one full cycle of the fast clock. For most designs, the compiler can schedule the full cycle of a fast clock in fewer than 256 steps.

The following two options to the `terminalTiming` command generate pulses on external connectors:

- `-pulse`: generates a single pulse per user clock cycle
- `-edgeInfo`: generates multiple pulses during a single fast-clock cycle

For more information, refer to the `terminalTiming` command in the *VXE Command Reference Manual*.

Specifying Delay for the Palladium Z1 Emulator

Palladium Z1 emulators have enhanced timing accuracy. To specify the correct delay units on the Palladium Z1 emulator, use the `compilerOption -ioDelayUnit pd2tif | uatif | step | ns` command, which enables you to specify whether the delay should be in step, tif, or nanoseconds format.

© 2021 All Rights Reserved.

Specifying Target Connection in SA Mode

In SA mode, the task of establishing target connections is handled by the IXCOM compiler, and you need to do the following:

- Compile both the testbench and design files using the IXCOM compiler. Specify all options of the `ixcom` command, such as `-ua` and `+dut`, which are required for identifying the design mapping to the emulator.
- Provide information about the target to the design-interface connection.

  Specify the design unit for the target connection by using the `+targetTop` option of the `ixcom` command.

In SA mode, you need to specify the same settings to connect the target to the design mapped to the emulator as the ones listed for the ICE mode in the Specifying the Target Connection in ICE Mode section. However, for SA mode, the signal routed to the target interface cannot be shared with the connection to the host workstation (testbench). The IXCOM compiler generates an error under such condition. This means that the signal routed to the target interface cannot be directly read by the testbench. Further, for ICE mode, you can connect only the top-level design unit to the target interface, and the HDL-ICE Compiler determines the top-level I/O connections to the target system, based on the `cableConnection`, `terminalAssign`, and `terminalTiming` User Data commands.

For SA mode, on the other hand, you need to specify the `+targetTop+<top_level_designunit>` option of the `ixcom` command in addition to the `cableConnection`, `terminalAssign`, and `terminalTiming` User Data commands. Also, you need to specify the full hierarchical name (including the testbench hierarchy) for the `<terminal_name>` option with the `terminalAssign` and `terminalTiming` User Data commands.

You can specify multiple design units with the `+targetTop+` option. For each design unit specified as the target top, the compiler makes all ports of the module available for target connection. The use of the `+targetTop` option is the same for both multi-step and single-step IXCOM compilation.

The interface type for the target connection is determined as follows for each design unit specified as the target top:

Design Unit is Empty	Design Unit has at least One Port Connected	Target Interface
No	No	All I/Os are made available for target connection.

© 2021 All Rights Reserved.

Design Unit is Empty	Design Unit has at least One Port Connected	Target Interface
No	Yes	Error: all ports of a target top design unit must be unconnected for use by target interface.
Yes	No	Error: target top design unit specified is empty (or has no logic).
Yes	Yes	Design unit is made available for target connection as component adapter (caCell).

Example: Target Connection in SA Mode

The following example illustrates the specification of target interface connection as described in later sections.

```
//-----------------------------------------------------------------
// File : tb.sv
// Description:
//             Testbench module for target connection unit testing
//-----------------------------------------------------------------
module Tb;
     // All ports must be unconnected for target top design units
     Clk cg(.clk());
     Dut dut(, , );
     // ...
endmodule // Tb
//-------------------------------------------------------------------------
// File : dut.sv
// Description:
//              DUT design units for target connection unit testing
//-------------------------------------------------------------------------
module Dut(clk, din, dout);
     input clk;
     input [1:0] din;
     output [1:0] dout;
     Flop flop(clk, din, dout);
     Cell cell(clk,, );
endmodule // Dut
module Clk(output reg clk);
     initial clk = 0;
     always
          #1 clk = ~clk;
endmodule // Clk
module Flop(clk, din, dout);
     input clk;
     input [1:0] din;
     output reg [1:0] dout;
     // non-empty module with logic
     always @(posedge clk)
          dout = din;
endmodule // Flop
```


© 2021 All Rights Reserved.

```
module Cell(clk, din, dout);
     input clk;
     input [1:0] din;
     output reg [1:0] dout;
     // cell module must be empty (no logic)
endmodule // Cell
//-----------------------------------------------------------------------
```

Connecting Design Unit I/Os to the Target in SA Mode

This section illustrates the various ways or commands for specifying the connection of the top-level I/Os to the target when using the IXCOM compiler.

While specifying the design units, you can specify a single `+targetTop` option with multiple values or multiple `+targetTop` options of the `ixcom` command with one design unit. The use of the `+targetTop` option works in the same manner for both single-step and multi-step IXCOM modes.

- Connecting top-level design unit I/Os to the target
  - To connect the top-level design unit, use the following command:

    ```
    ixcom -ua +dut+Dut +targetTop+Dut dut.sv
    ```

  - To connect multiple top-level design units, use the following command:

    ```
    ixcom -ua +dut+Dut +targetTop+Dut+Clk dut.sv
    ```

- Connecting instantiated (non-top-level) design unit I/Os to the target
  - To connect non-top-level design units, use the following command:

    ```
    ixcom -ua +dut+Dut+Clk +targetTop+Dut tb.sv dut.sv
    ```

    Note: In the above compilation, the testbench hierarchy is included with the DUT instantiated inside the testbench.

  - To connect non-top-level design units from different hierarchies, use the following command:

    ```
    ixcom -ua +dut+Dut+Clk +targetTop+Clk +targetTop+Flop tb.sv dut.sv
    ```

    Note: In the above compilation, the testbench hierarchy is included with the DUT instantiated inside the testbench.

Note: In addition, refer to the points listed in Points of Consideration for Specifying Target Connections in SA Mode on page 129.

For both top-level I/O and instantiated-design-unit connections to the target interface, the design unit specified as the target top must meet the following requirements:

- All non-top design units must have all ports unconnected.

© 2021 All Rights Reserved.

- Design units must contain logic (must not be empty).

To connect the top-level design unit to the target interface, use the following steps:

1. Specify the target interface connection details in the `compilerOptions.qel` file (in the current design directory) with commands like the following:

```
# ...
cableConnection -add { <cable_label> <phy_loc> { <I/O_technology> } }
terminalAssign  -add { <interface_name> [<interface_type>] <pin> Dut.clk } \
              { <interface_name> [<interface_type>] <pin> Dut.din[1] }  \
              { <interface_name> [<interface_type>] <pin> Dut.din[0] }  \
              { <interface_name> [<interface_type>] <pin> Dut.dout[1] } \
              { <interface_name> [<interface_type>] <pin> Dut.dout[0] }
terminalTiming  -add { Dut.clk <constraint> }
# ...
```

   Note: The `compilerOptions.qel` file can include other commands in the same order as required in ICE mode.

2. Compile the design files using the `ixcom` command:

```
ixcom +targetTop+Dut dut.sv -ua +dut+Dut
```

Using Component Adapter (CACELL) for Target Connection in SA Mode

In addition to connecting design units to the target interface, you can mark cell(s) as component adapter(s) for the target connection. Using the specified information, the precompiler ensures that the nets connected to these cell(s) are available for target interface connection. All internal signals of the design that need to be routed to the target connection are connected as actual to the instance of the component adapter.

In ICE mode, the component adapters are marked by using the `caCell` IP command, In SA mode, you use the `+targetTop` option of the `ixcom` command to specify the component adapter(s) for the target connection, in place of the `caCell` command used in ICE mode.

In SA mode, you need to create an empty (shell) module for the component adapter (referred in the `caCell` command) module. The IXCOM compiler compiles the design files along with the empty module as a part of the design hierarchy. The hierarchy under the design is partitioned to the hardware or software based on the content of the module. The external connections to the simulator for hardware or software interface signals are automatically created by the IXCOM compiler.

For all component adapter target connections, the design unit, which is specified as the target top, must meet the following requirements:

- The design unit must have at least one port connected.
- The design unit must be empty (shell module).

© 2021 All Rights Reserved.

- The port direction (reversed) requirements remain the same as the ICE use model.

This section specifies the various ways to connect a design unit to target as a component adapter (caCell). The following commands are based on the Example: Target Connection in SA Mode.

To connect a top-level DUT module as a component adapter in SA mode:

```
ixcom -ua +dut+Dut +targetTop+Dut dut.sv
```

Note: In the above command, the DUT content is assumed to be empty like the Cell module in the previous example (Example: Target Connection in SA Mode).

To connect a non-top level DUT module as a component adapter in SA mode:

```
ixcom -ua +dut+Dut +targetTop+Cell tb.sv dut.sv
```

Note: For multi-step IXCOM compilation, the `+targetTop` option is used in the same manner as specified above.

To connect the design unit (or cell) identified as component adapter to the target interface:

1. Specify the target interface connection details in the `compilerOptions.qel` file with details such as:

```
# ...
cableConnection -add { <cable_label> <phy_loc> { <I/O_technology> } }
terminalAssign  -add \
         { <interface_name> [<interface_type>] <pin> Tb.dut.cell.clk[1] }  \
         { <interface_name> [<interface_type>] <pin> Tb.dut.cell.din[0] }  \
         { <interface_name> [<interface_type>] <pin> Tb.dut.cell.dout[1] }
\
         { <interface_name> [<interface_type>] <pin> Tb.dut.cell.dout[0] }
terminalTiming  -add  { Tb.dut.cell.din[1] <constraint> }
# ...
```

2. Compile the design files as follows:

```
ixcom tb.sv dut.sv -ua +dut+Top +targetTop+Cell
```

When using the component adapter connection to target system, in ICE mode, the *<terminal_name>* option in the `terminalAssign` and `terminalTiming` User Data commands include hierarchical path starting from the first component instance within the top-level cell. However, in SA mode, you need to specify the full hierarchical name (including testbench hierarchy) for the *<terminal_name>* option referred in the `terminalAssign` and `terminalTiming` User Data commands.

Note: In addition, refer to the points listed in Points of Consideration for Specifying Target Connections in SA Mode on page 129.

© 2021 All Rights Reserved.

Connecting an Arbitrary Internal Signal to the Target Interface

In ICE mode, you can connect an arbitrary internal signal in the DUT to a target interface by using the `globalNet` XEL command or the `iScopedNets` program. For non-top design units, you must include the full hierarchical path to the signal.

The description of the `globalNet` command is available in the *User Data Commands* chapter of the *VXE Command Reference Manual*. Refer to "Connecting Nets Using iScopedNets Program" section in *VXE User Guide* for a description of the `iScopedNets` program.

This section provides details on these methods.

Example of Connecting Using Arbitrary Internal Signals to Target Interface

The following basic HDL example is used to explain the connection mechanism:

```
// File: Dut.sv
module Dut(input clk, din/*, output doutT */)
     input clk;
     input din;
// top-level dout terminal is created using various methods listed below
     Sub sub(clk, din, dout);
endmodule // Dut
// File Sub.sv
module Sub(input clk, din, output dout);
      always @(posedge clk)
       dout = din;
endmodule  // Sub
```

All use models described in this section to connect an internal signal using *global net* require an explicit precompile step in the order listed below:

1. Make internal signal(s) available as global net.
2. Assign the top-level or component adapter terminal to a target system using the `terminalAssign` User Data command.
3. Invoke the precompile step to make *global net* available for target connection.
4. Assign the global net terminal to target system using the `terminalAssign` User Data command.
5. Specify the terminal timing details.

These steps are explained below for the different ways to create a *global net* to connect to target interface.

Connecting the Arbitrary Signal as Global Net to the Target Interface

© 2021 All Rights Reserved.

In this method, an internal signal is connected to the top-level terminal as global net. The User Data specification syntax is:

```
globalNet -add { <net> <term> }
```

In the above User Data command, a global net is created for the hierarchical net *<net>* as top-level terminal *<term>*. The *<term>* must be unique, and must not be an existing top-level terminal. To connect top-level design unit and global net signals to the target interface, use the following compilation steps:

1. Specify the User Data commands to create global net and connect signals to target interface in the `compilerOptions.qel` file.

```
# ...
# netlist import
# designImport
# design hdldb xcva_top
# Create global net for Dut.sub.dout
# NOTE: Scope for this global net is set to top-level cell specified by design
command earlier
globalNet -add { Dut.sub.dout doutT }
# set up design for precompile
# compilerOption -add ...
# userData -load ...
cableConnection -add { <cable_label> <phy_loc> { I/O_technology } }
# Terminal assigns for top-level I/O (see earlier sections)
precompile
# set up design for compile
# Terminal assigns for new top-level terminals created using globalNet
# NOTE: Terminal name referred here belongs to top-level cell
terminalAssign -add { <interface_type> [<interface_name>] <pin> TERM doutT }
# Terminal timing (see earlier sections)
# ...
```

2. Compile the design files using the `ixcom` command as follows:

```
ixcom dut.sv sub.sv -ua +dut+Dut +targetTop+Dut
```

Note:

- An explicit precompile step is required prior to setting up design for compile.
- In addition, refer to the points listed in Points of Consideration for Specifying Target Connections in SA Mode on page 129.

Connecting User Global Net Using Pinprop Attribute

In this method, an internal signal is connected to top-level terminal as user global net using the specification in the Attribute File Format. User global net information is provided in the attribute file as follows:

- Library cell to which the internal terminal signal or net belongs
- Internal terminal signal name

© 2021 All Rights Reserved.

- Top-level terminal name

For the sample HDL example, the following attribute specification is used to create the global net:

```
<code>
LIBCELL Sub 0 0 0
PINPROP dout userGlobalNet doutT
```

To connect the top-level design unit to the target interface:

1. Specify global net using attribute specification in the `externalNetlists.qel` file.

   ```
   # ...
   # import attribute specification
   netlistFile { attribute <attr_file> }
   ```

2. Specify the target connection details for the top-level signals and global net created.

   ```
   # designImport
   # design hdldb xcva_top
   # set up design for precompile
   # compilerOption -add ...
   # userData -load ...
   # cableConnection -add ...
   # Terminal assigns for top-level I/O (see earlier sections)
   precompile
   # set up design for compile
   # Terminal assigns for new top-level terminals created using userGlobalNet
   # NOTE: Terminal name referred has implied top-level hierarchy
   terminalAssign -add { <interface_type> <interface_name> <pin> TERM doutT }
   # Terminal timing (see earlier sections)
   # ...
   ```

3. Compile the design files using IXCOM tools as follows:

   ```
   ixcom dut.sv sub.sv -ua +dut+Dut +targetTop+Dut
   ```

Note:

- An explicit precompile step is required prior to setting up design for compile.
- The attribute file is imported just like any other netlist file in ICE mode. For details about this process, refer to section “Importing a Netlist” in *VXE User Guide*.
- In addition, refer to Points of Consideration for Specifying Target Connections in SA Mode on page 129.

Points of Consideration for Specifying Target Connections in SA Mode

Keep in mind the following points when specifying the target connection in SA mode using design units (Connecting Design Unit I/Os to the Target in SA Mode on page 124), component adapters (Using Component Adapter (CACELL) for Target Connection in SA

© 2021 All Rights Reserved.

Mode on page 125), and/or arbitrary signals (Connecting an Arbitrary Internal Signal to the Target Interface on page 127):

- The `compilerOptions.qel` file in the IXCOM compile directory must include target-connection-specific commands similar to those used in ICE mode.
- As compared to ICE mode, connecting target connections in SA mode involves the following differences:
  - Terminal names must include the testbench top module name. For example, specify the name as `top.clk` rather than `clk`.
  - The `compilerOptions.qel` file must include an explicit `precompile` command before any `delayBox` and `terminalTiming` commands.

© 2021 All Rights Reserved.

Working with Emulators at Run Time

You can work with the emulator hosts at run time in the following ways:

- Specifying the Emulator
- Identifying Possible Locations for Downloading Design
- Relocating Designs on Palladium Z1 Systems
- Relocating Designs to Domains with Known Defects
- Reserving Emulator Resources
- Reducing Emulator Initialization Time Using KeepHostAlive
- Using Suspend and Resume to Manage Emulator Resources

Specifying the Emulator

The emulator can be specified at the following time points:

- During Design Compilation for ICE flow

  `emulatorConfiguration` command in the compilation script

- During Run

  By specifying the implicit `host .` command ('.' selects current host) or explicit `host` *<host_name>* command

During run time, use either the xeDebug GUI interface or the `host` XEL command to specify the emulation host to which the emulator is connected. For more information on the syntax and usage of the `host` command, refer to the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Important

Emulation host is different from SA host. In SA mode, xeDebug must run on the SA host, while ICE mode does not have this requirement. The SA host and emulation host can be two different workstations or the same workstation, depending on how the Palladium Z1 emulator was set up.

© 2021 All Rights Reserved.

Specifying the Design Directory

By default, the `host` command starts the DBEngine in the same directory in which xeDebug is running. However, the directory name is translated to a portable format, like `/net/<machine_name>/...`

If the automatic path translation generates any performance issues or access problems, use the following `host` command to specify the design directory to be used:

```
host <hostname> [-path <directory_name>]
```

<directory_name> must be the path to the debug directory as seen from the workstation *<hostname>*. The directory path must be an absolute path. Following is the most recommended format:

```
host <hostname> -path [pwd]
```

The above command suppresses automatic path translation without the need to use a hard-coded directory name.

Identifying Possible Locations for Downloading Design

Use the `-location` option of the `test_server` command as follows to find possible locations for downloading your design:

```
test_server <top_cell_name> -location [-all]
```

Note: This command can be executed only if you have a *<top_cell_name>*`.proto` file in the `./dbFiles` directory.

The `test_server <top_cell_name> -location` command returns a list of possible locations where a design can be downloaded and run. This list of locations is determined by static weighted scoring method, which indicates how well the domains and resources match the requested job configuration. A location is scored higher if it has physical resources that are not needed by the design. And the resultant locations are listed in increasing order of scores.

In Palladium Z1, the combination of possible places to relocate is huge. So, `configmgr` lists only possible locations where a design can be downloaded and run. At placement time during the design download, `configmgr` dynamically picks the best locations that match the requested job configuration based on the current emulator usage. These locations are not necessarily the best locations when the actual job placement happens.The `configmgr` can pick a location that is not listed by the `test_server <top_cell_name> -location` command.

The `test_server <top_cell_name> -location -all` command displays possible locations including the locations that are currently not available because of locked resources.

© 2021 All Rights Reserved.

Relocating Designs on Palladium Z1 Systems

You can compile a design for a certain set of emulator resources (racks, clusters, LDs, or domains), and run it on a different set. Running a design on a different set of emulator resources than it was compiled for is called *relocating* the design. For a description of various emulator resources, refer to the *Hardware Overview* section in the *Palladium Z1 System Overview* chapter of the *VXE Planning and Installation Guide*.

Palladium Z1 provides you a variety of relocation options to chose from as compared to the previous generation systems.

Therefore, the `test_server` command in Palladium Z1 returns more locations. However, the locations returned by the `test_server` command at run time might be different than the locations returned by the command during compile time. This happens for the following reasons:

- The previous available location might now be occupied by another user design.
- The source and the destination locations are not connected with each other. Relocation in Palladium Z1 requires proper connectivity between various hardware components. The *Checking Connectivity of Hardware Components* section provides information on how the various hardware resources should be connected with each other.
- The source and the destination locations do not maintain their hierarchical relationships. Two domains compiled with some hierarchical relationship (intra-LD, intra-cluster, intra-rack, or inter-rack) must be relocated with same hierarchical relationship. The following table lists various relocation rules:

Rule	Description	Portion of Design Database Fits Within		
		A Single LD	Multiple LDs Within a Rack	Multiple LDs Across Racks
Hierarchy	Domains must maintain hierarchical relationship	Y	Y	Y
Half-LD	Domains must maintain half-LD relationship		Y	
Position	Domains must maintain LD-position relationship			Y

© 2021 All Rights Reserved.

Figure 6-1 Job Relocation and Placement

Example: Shown here is a single five-domain job occupying domains A through E. The diagram illustrates sample relocations for the same job at different locations in a single-rack system, with and without altering the domain sets.

To view the available locations to run the jobs, issue the `test_server` command as follows:

```
test_server <design_name> -location -all
```

Palladium Z1 enables you to move the jobs in the following ways:

- Move the jobs to another domains on the same LD
- Move single-LD jobs freely within a rack
- Move a multi-LD job and single-cluster job within the same or a different cluster
- Move multi-LD and multi-cluster job within a rack
- Move single-LD jobs outside of a rack
- Move jobs spanning across multiple racks with multi-rack cabling

For more information, refer to the Relocation Examples section that discusses the different ways in which you can relocate the jobs in the Palladium Z1 systems.

© 2021 All Rights Reserved.

To reserve a location for running the design, issue the following command:

```
test_server <top_cell_name> -reserve
```

For details on how to reserve the emulator resources, refer to section Reserving Emulator Resources.

You can also compile the designs for best relocation to increase the relocation choices for the compiled designs. Compile the design using the `connectivityOptimizationGoal bestRelocation` option of the `compilerOption` command as follows:

```
compilerOption -add {connectivityOptimizationGoal bestRelocation}
```

For more information, refer to section Compiling the Designs for Best Relocation.

Relocation Advantages in Palladium Z1

More relocation options in Palladium Z1 improve resource utilization because the run-time software prefers free and non-contiguous locations. Therefore, the emulator resources are efficiently utilized. The run-time software uses a scoring mechanism where the resources that are partially used and are non-consecutive, get a better score than the ones that are fully available and consecutive.

For example, assume there are no known defects in both the requested and physical configuration, and a job request is for `0.0+0.1+0.2+0.3+0.4`. Following four LDs are available in the emulator for placement such that:

- LD 0: All 8 domains are available
- LD 1: Available domains are `1.0+1.1+1.2+1.5+1.6+1.7`
- LD 2: Available domains are `2.0+2.1+2.2+2.3+2.4+2.5`
- LD 3: Available domains are `3.0+3.2+3.4+3.5+3.7`

The above LDs are scored in order such that: LD 3 < LD 1 < LD 2 < LD 0, and LD 3 is picked as a fit location for this job because it has the best emulator utilization result.

The LD3 domains have been used partially and the following non-consecutive domains are free for use: 3.1, 3.3, and 3.6. So, relocating jobs to LD 3 would fill it. Similarly, LD 1 becomes the next preference for relocation followed by LD 2 and LD 0 is the last preference. The LD 1 and LD 2 solutions would leave a board with only a single domain free. The LD 1 solution scores better than the LD 2 solution since LD 1 is already non-contiguous, so it is preferable to consume those domains and leave the contiguous domains in LD 2 available. LD 0 has the least preference because all the eight domains are available for use.

© 2021 All Rights Reserved.

Checking Connectivity of Hardware Components

The Palladium Z1 emulation system has interconnects wired with different connectivity at each level of the hierarchy between domains within a Logic Drawer (LD), domains within a cluster, and domains within a rack. Relocation of jobs is done within the following resource hierarchical groups.

- Intra-LD: Every domain connects to every other domain on the same LD as shown in the following figure.

Figure 6-2 Connectivity between Domains

- Intra-cluster: All LD-to-LD domains within the same six-LD cluster. The LD-domain pairs exist within the same QUAD, where a QUAD is defined as domains within the same half-LD.

© 2021 All Rights Reserved.

Figure 6-3 Connectivity between LDs

- Intra-rack or inter-cluster: Any domain pair that crosses clusters within the same rack. Every domain connects to the corresponding domain in the other clusters. For example: 0.0 is connected with the following: 6.0, 7.0, 8.0, 9.0, 10.0, 11.0.

© 2021
All Rights Reserved.

Figure 6-4 Connectivity between Clusters

- Inter-rack: Any domain pair that crosses racks. Every domain on one LD connects to the corresponding domain on another LD in the other racks, provided there is a cable between the LDs. For example: 0.0 is connected with 23.0 and 0.1 is connected with 23.1.

Figure 6-5 Connectivity between Racks

© 2021
All Rights Reserved.

The following example shows domains with connections to domain 0.0. Position dependence is maintained with connections to the corresponding domains between half LDs, clusters, and racks.

```
0.0 <-> 1.0, 1.1, 1.2, 1.3
0.0 <-> 6.0
0.0 <-> 19.0
```

The following diagram shows the connectivity between domains, LDs, clusters, and racks. The connectivity shown between racks is for the standard cabling between XL racks.

Figure 6-6 Domains with Connections to Domain 0.0

Relocation Examples

The following sections discuss different ways in which you can relocate the jobs in the Palladium Z1 systems.

- Moving Jobs on the Same LD
- Moving Single-LD Jobs within a Rack
- Moving Multi-LD Single-Cluster Job within a Cluster

© 2021
All Rights Reserved.

- Moving Multi-LD, Multi-Cluster Jobs within a Rack
- Moving Single-LD Jobs outside of a Rack
- Moving Multi-Rack Jobs

Moving Jobs on the Same LD

The following figure illustrates an example of moving a single-LD job freely within an LD. This is an example of a single three-domain job occupying domains A through C. The following diagram illustrates sample placements for the same job on the same LD.

Figure 6-7 Moving Jobs on the Same LD

Moving Single-LD Jobs within a Rack

The following diagram shows an example of a single five-domain job occupying domains A through E.

© 2021
All Rights Reserved.

Figure 6-8 Moving Single-LD Jobs within a Rack

The following diagram illustrates sample placements for the same job in which LD hierarchical relationship is not maintained. Such job relocations are not supported.

© 2021
All Rights Reserved.

Figure 6-9 Illegal LD Hierarchy

Moving Multi-LD Single-Cluster Job within a Cluster

The following figure illustrates the sample placements for a single five-domain job occupying domains A through E.

© 2021
All Rights Reserved.

Figure 6-10 Moving Multi-LD Single-Cluster Job within a Cluster

The following diagram illustrates sample placements for the same job in which the half-LD hierarchical relationship is not maintained. The following example shows placements for a single five-domain job occupying domains A through E.

© 2021
All Rights Reserved.

Figure 6-11 Illegal Half-LD Hierarchy

Moving Multi-LD, Multi-Cluster Jobs within a Rack

The following figure illustrates the sample placements for a five domain job occupying domains A through E within the same rack, but on different LDs, and different clusters.

© 2021
All Rights Reserved.

Figure 6-12 Moving Multi-LD, Multi-Cluster Jobs within a Rack

The following diagram illustrates sample placements for the same job in which connections between the clusters and position dependence are not maintained.

© 2021
All Rights Reserved.

Figure 6-13 Invalid Cluster Connections and Position Dependence

Moving Single-LD Jobs outside of a Rack

The following figure illustrates the sample placements for a five-domain job occupying domains A through E in one rack to another rack.

© 2021
All Rights Reserved.

Figure 6-14 Moving Single-LD Jobs Outside of a Rack

Moving Multi-Rack Jobs

The following figure illustrates the sample placements for a five-domain job occupying domains A through E in multiple racks, with multi-rack cabling.

© 2021
All Rights Reserved.

Figure 6-15 Moving Multi-Rack Jobs

The following figure illustrates sample placements for a five-domain job occupying domains A through E in multiple racks, with multi-rack cabling, in which the position dependence or domain connection is not maintained.

© 2021
All Rights Reserved.

Figure 6-16 Illegal Domain Connections

Following are some more examples for valid and invalid relocation choices. The valid relocations are shown in blue and invalid relocations are shown in red.

Figure 6-17 illustrates multi-LD domain relocation within a cluster or rack hierarchy.

© 2021
All Rights Reserved.

In the diagram, relocation choices marked in red violate the hierarchical constraint, and are, therefore, considered illegal relocation. The job requesting multi-board intra-cluster first-half domains must be relocated to maintain the same hierarchical relationship.

For example,

- Broken board relocation choice violates the multi-board constraint
- Broken cluster choice violates the intra-cluster constraint
- Broken half-board choice violates the half-board constraint.

Relocation choices shown in blue maintain the hierarchical relationship where the first choice marked in blue follows the relocation mechanism from previous generation systems, maintaining the order and domain sets as per the job request, and the second choice marked in blue follows relocation in Palladium Z1 by changing the order and placement of D and E domains.

© 2021 All Rights Reserved.

Figure 6-17 Multi-Logic-Drawer-Domain Relocation within Cluster or Rack Hierarchy

In the diagram, three relocation choices shown in red color violate the hierarchical constraints for placement, and, hence, are invalid relocations:

```
2.0+2.1+2.2+2.3+2.4
5.0+5.1+5.2+6.0+6.1
8.0+8.1+8.2+9.4+9.5
```

The above examples show relocation possibilities assuming there are no lane defects in the compiled design. The patterns change when there are lane defects found in the compiled design. The following figure highlights one example in which there are no lanes from one LD to a certain domain on another LD.

© 2021 All Rights Reserved.

Figure 6-18 illustrates multi-logic drawer intra-cluster domain relocation with defects. markouts.

Figure 6-18 Multi-Logic Drawer Intra-Cluster Domain Relocation with Known Defects

In the above figure, domain D is marked out from board 0. The domains A, B, and C are connected with each other, and D is connected only to C. Therefore, relocation choices shown in blue color for domains A, B, and C in the same quad, and for domain D in a different quad are valid choices. However, the following relocation choices shown in red color are invalid relocations because domain D is marked out.

```
4.0+4.1+5.3+5.6
```

Relocating Designs to Domains with Known Defects

A design might be relocated to any domain set with known defects provided that the design was compiled with the known defects. This can be done explicitly with or without the `symmetricConfiguration` command. Wire parity check in Palladium Z1 is not required to relocate to known defective resources.

© 2021 All Rights Reserved.

For example,

Consider a single board (eight domains) configuration, where every other domain has a random defective P8. The configuration might look as follows, where every odd domain has a defective P8 number 0:

```
+ SCD  0  SCD006035000000
+ NCC  0  NCC006035000000
+  8_CHIP_BOARD  0 ET6 BRD00000000000
- P8 0 1 0 * BAD P8  number 0 on Board 0 Domain 1,
- P8 0 3 0 * BAD P8  number 0 on Board 0 Domain 3
- P8 0 5 0 * BAD P8  number 0 on Board 0 Domain 5
- P8 0 7 0 * BAD P8  number 0 on Board 0 Domain 7
```

Consider the following two scenarios:

Compile for 0.0.

Suppose you compile a single domain design and select `0.0` as the compile domain string. Here, the compiler does not know about the defective domains and uses every P8 in the domain. If the entire domain is not needed, the compiler still utilizes every resource in the requested compiled configuration to achieve the maximum performance. Since the entire domain was used by the compiler, when you download this design, the relocation candidates are only domains 0, 2, 4, and 6.

Compile for 0.1

Now suppose you recompile and select `0.1` as the compile domain string. In this case, the compiler knows there is a defective P8 and will not utilize that portion of the domain.

The overall compile performance (step count) might be reduced due to less resources in the domain. However, in many cases, this performance reduction is negligible.

When you download the design, the relocation candidates are all domains 0, 1, 2, 3, 4, 5, 6, and 7 because the domains 0, 2, 4, and 6 are still candidates as a section of the domain is not used by the design and therefore, a fully functional domain can still accommodate a design that does not use the entire domain. Configmgr, will however, give these fully functional domains a reduced score and will not pick them for placement until all the defective domains are already taken by other jobs.

The benefit of compiling with known defects is that you achieve more relocation possibilities when the emulator is more utilized by other jobs. The cost of compiling with known defects is the potential for reduced step count. The measure of step count performance reduction when compiling with known defects is case by case.

© 2021 All Rights Reserved.

Reserving Emulator Resources

Large designs can take a considerable amount of time to download on the emulator, and during this time, it is possible that the hardware resources might be taken by another user's design. To avoid this situation and guarantee the complete design to be downloaded, it is a good idea to reserve the emulator resources for some time before beginning the design download. The emulator resources can be reserved by using the `test_server` program with the compiled design.

Before reserving the emulator resources, you can execute the following option of the `test_server` program to identify the best hardware configuration for your design:

```
test_server <top_cell_name> -location [-all]
```

For more information, refer to the Identifying Possible Locations for Downloading Design section.

With respect to reserving hardware resources, the `test_server` program provides the options for:

- Reserving the Emulation Resources (if required, for a specific duration of time)
- Using the Reserved Resources for Downloading the Design
- Releasing the Reserved Resources

Reserving the Emulation Resources

Following is the syntax of the `test_server` program to reserve emulation resources:

```
test_server <top_cell_name> \
            -reserve [-relocateTarget -qel=<qelFile> [-qel=<qelFile2>] \
            -relocateTarget -txt=<txtFile> | -relocateTarget
"<inst1>=<loc1>,<inst2>=<loc2>.."]\
            [-key <key_value>] \
            [-timeout <t> [<unit>]] | -holdtime <t> [<unit>]] \
            [-retry [<num>]] \
            [-bp] \
            [-keyfile <reservationFilePath>] \
            [-firstfit]
```

Note:

- When you use the `-bp` option in `test_server`, targets are not reserved, only specified domains are reserved in `.bp` file.
- `-relocateTarget` option with `-bp` option is unsupported.

© 2021 All Rights Reserved.

For more details on the `test_server` program, refer to the *Programs and Utilities* chapter of the *VXE Commands Reference Manual*.

Note:

- In ICE mode, the emulator resources required by the design are automatically reserved for maximum 10 minutes (for one time use only) on the execution of the `host <hostname>` command. In SA mode, similar type of automatic emulator resource reservation is enabled when you specify the `host <hostname> -reserveFirst` command. For more information about the `host` command and its syntax, refer to the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.
- In ICE mode, to turn off the automatic emulator resource reservation done before design download, set the `DISABLE_RESERVE_FIRST` environment variable to `1` before invoking the xeDebug tool.

Reserve the emulator resources for a specific time period in either of the following modes:

- ***Single-session mode*** (`-timeout <t> <unit>` option) – In this mode, the emulator resources are reserved for the specified time limit `<t>` to enable a single download of the design. After the design has been downloaded, the reserved resources are immediately released, even if the time limit has not expired.
- ***Multi-session mode*** (`-holdtime <t> <unit>` option) – In this mode, the emulator resources are reserved to enable multiple downloads of the same or different designs until the time limit `<t>` expires. The reserved resources are released only after the time limit has expired.

When specifying the mode as single-session or multi-session, define the time limit `<t>` in terms of either seconds, minutes, or hours time `<unit>`. If the time limit is not specified, by default, the resources are reserved in the single-session mode for five minutes. Reservation time limit cannot exceed 30 days (that is, 2592000 seconds or 43200 minutes or 720 hours).

The resource reservation takes place only after the `test_server` program has confirmed that the resources are available. Otherwise, an error message displays that the resources cannot be reserved. In such situations, the `-retry` option of the `test_server` command can be used to check the availability of the resources at regular intervals until resource reservation succeeds or the process is manually aborted.

If more than one set of resources are available for reservation, by default, only one is reserved. For example, a user has a one-domain design and a Palladium Z1 system with four domains available. When the following resource reservation command is executed, one best fit domain out of the available four domains will be reserved:

```
test_server <top_cell_name> -reserve
```


© 2021 All Rights Reserved.

When resource reservation is successful, the `test_server` program displays the following confirmation:

```
Emulator domains reserved with key 999303234 for 300 seconds.
Reserved resources (logic drawers/domains) are: 1.10
Exit code: 0
```

To determine that the resources are reserved, specify the `test_server` command without any arguments. The information about the reserved domains is displayed as shown in the following sample `test_server` output:

```
test_server
Emulator Host: <host> Hardware: Palladium Z1 System Status: ONLINE
Cluster 3 has 6 logic drawers           CCD: ONLINE
Logic drawer 18 has  8 domains              Logic drawer: ONLINE
Domain   Owner       PID            T-Pod    Design         ElapTime    ReservedKey
0        asdsqa      <host>:39338   --       xcva_top        00:00:29    --
1        asdsqa      <host>:16679   --       xcva_top        00:19:49    --
2        asdsqa      <host>:9870    12       xcva_top        00:24:47    --
3        NONE        0               --       NONE           --          --
4        NONE        0               18       NONE           --          --
5        NONE        0              --       NONE           --          --
6        NONE        0              --       NONE           --          --
7        NONE        0              --       NONE           --          --
Logic drawer 19 has  8 domains              Logic drawer: ONLINE
...
Logic drawer 20 has  8 domains              Logic drawer: ONLINE
...
Logic drawer 21 has  8 domains              Logic drawer: ONLINE
...
Logic drawer 22 has  8 domains              Logic drawer: ONLINE
...
Logic drawer 23 has  8 domains              Logic drawer: ONLINE
...
Static Target Information:
T12 18.2
...
T18 18.4
Exit code: 0
```

In the `test_server` output, the *Owner* column displays the `<username>` adjacent to the reserved domains, the *Design* column displays `RESERVED*`, and the *ElapTime* column shows the reserved time left in `HH:MM:SS*` format. The `*` indicates the reservation expiration time left within which the design must be downloaded.

After design download, the `test_server` output displays the reserved domain resource as a regular domain with the actual `<design_name>` displayed under the *Design* column, and the *ElapTime* column displaying the elapsed time since the design is downloaded.

Mapping Reservation to a Key

Each reservation of the emulator resources is mapped to a specific key value. If required, specify a value of your choice for the key by using the `-key <key_value>` option of the

© 2021 All Rights Reserved.

`test_server` command. The key value is unique for each user, and is stored in ASCII format in the `ts.<username>.resv.key` file in the `./dbFiles` directory. But the same user can make multiple reservations for different or same design. The key file is not protected, and can be corrupted. Therefore, Cadence recommends that multiple reservations for the same design at the same time should not be made from different terminals.

Later, the xeDebug tool reads the key value from the file in automatic key-use mode. This means that the key value need not be specified manually using the `XE_RESERVE_KEY` environment variable or the `xeset reserveKey` run-time parameter. Refer to `xeset` command description in the *Run-Time Commands* chapter of the *VXE Command Reference Manual* for syntax usage details.

Cadence recommends that both `test_server` program and xeDebug tool are run from the same design directory. This enables xeDebug to automatically locate the key file during design download. If the `test_server` program and xeDebug tool cannot be run from the same design directory, before initiating design download, the key must be manually specified to xeDebug. Refer to Using the Reserved Resources for Downloading the Design.

Using a .bp File for Resource Reservation

When you compile a design, the compiled database includes a logical configuration starting with board 0. The logical configuration must correspond to a subset of the physical configuration of the emulator to which you download. Since this is a logical configuration, the program analyzes it and attempts to make a best fit onto the physical configuration by mapping logical boards to different physical boards, as necessary.

However, this best fit method can be overridden by using a board preference (`.bp`) file, located in the user session directory or the design directory. This `.bp` file will force the run-time software to place the design on designated physical logic boards specified in the `.bp` file.

You can select specific domains to prioritize job placement and specify the LDs and domains in a `.bp` file such that the domains specified in the `.bp` file are used for resource-based scoring. The domains specified are matched in the order the domains were compiled for, while any unspecified domains in `.bp` file are left to the best fit algorithm to place. Multiple preferences can also exist for a single job such that the list of preferences forms a priority list.

Creating a Board Preferences (.bp) file

Following is the syntax of the `.bp` File:

```
<brd_pref_list>::= <brd_pref_string> "\n" <brd_pref_list> | <brd_pref_string>
<brd_pref_string>::= <brd_pref> "+" <bd_pref_string> | <brd_pref>
<brd_pref>::= <phy_brd> | <phy_brd> "." <phy_domain>
```


© 2021 All Rights Reserved.

Each line entry contains a string of either just the physical domain (`phy_brd.phy_domain`) or physical LD (*`phy_brd`*) separated by either "+" or space. There can be more than one line of entries implying the order of preference. Any bad entries other than the permissible syntax are not ignored but error out during download unless they are commented out using # or *. Specifying *`phy_brd`* alone translates to using the first domain on the LD for the first compiled domain. The `phy_brd` in the `.bp` file is the absolute board or LD number.

For example, to specify the LD 0 for rack 0 and cluster 1, specify the absolute LD number. That is, 6 (as there are 6 LDs per cluster and 3 clusters per rack). Similarly, to specify the LD 0 for rack 1 and cluster 0, specify 18 as the LD number.

Creating a Priority List

You can also control the priority of the preference list by specifying the `PRIORITY_FIT` keyword in the `.bp` file. If `PRIORITY_FIT` is specified, the preference order is the order in the which the domains are listed in the `.bp` file as opposed to score based ordering.

For an example, consider the following:

```
PRIORITY_FIT
0.1
0.3
```

Here, first preference is the top priority and the next preference is the next priority, and so on. So, `0.1` gets precedence over `0.3` in placement.

Reserving Resources

To use only the `.bp` file for resource reservation without the compiled design database information, specify the `-bp` option with the `test_server` <*`top_cell_name`*> `-reserve` command.

When you use the `-bp` option, ensure that the <*`top_cell_name`*>`.bp` file contains the complete logic drawers or domains information. For example, if a design requires four domains to download, the `.bp` file should have the information about all the four domains.

Note: If the number of LD's domains in the `.bp` file is not the same as in the job request, an error message is issued at run time.

When the `-bp` option is not specified, but a <*`top_cell_name`*>`.bp` file exists in the current compiled design directory, the `test_server` program will still use it. Without the `-bp` option, the starting domain or logic drawer specified in the `.bp` file together with the emulator resource requirement extracted from the compiled database will be used to reserve the resource. Every logic drawer in the `.bp` file is mapped to each logic drawer in the job request domain-for-domain in order of the request. If there are too many domains or logic drawers in the `.bp` file, the unmatched ones are ignored. And if there are not enough domains or logic

© 2021 All Rights Reserved.

drawers to match the requested domains or logic drawers, the remaining requested domains or logic drawers are placed according to the resource availability, connectivity, and hierarchy.

The following examples show the syntax and expected behavior. These examples assume a single cluster configuration with only LDs 3 and 4 available.

Example-1: Logical Request – 0.0+0.1

In this example, the .bp file looks like the following:

```
3.1
4.2
```

The above lines in the .bp file indicate forcing 0.0 placement in either 3.1 or 4.2, where 3.1 gets precedence over 4.2, and 0.1 can be placed anywhere resource availability, connectivity and hierarchy. The relocation mechanism in Palladium Z1 gives the following relocation choices for the logical request of 0.0+0.1:

```
3.1+3.2, 3.1+3.3, 3.1+3.4, 3.1+3.5, 3.1+3.6, 3.1+3.7, 3.1+3.0,
4.2+4.0, 4.2+4.1, 4.2+4.3, 4.2+4.4, 4.2+4.5, 4.2+4.6, 4.2+4.7
```

However, if you compile your design to achieve the best performance using the bestPerformance option of the compilerOption command, the relocation choices would reduce to the following: 3.1+3.2, 4.2+4.3. For more information, refer to the Compiling the Designs for Best Performance section.

Example-2: Logical Request - 0.0+0.1+1.0+1.1

In this example, the .bp file looks like the following:

```
3.1
4.2
```

The above .bp file forces domain relocation only for domain 0.0 in the job request to be either 3.1 or 4.2. Therefore, the run-time software is free to choose the rest of the domains. Valid relocation choices are as follows:

3.1+3.2+4.0+4.1, 3.1+3.3+4.0+4.1, 3.1+3.0+4.1+4.3 and so on.

4.2+4.3+3.0+3.1, 4.2+4.3+3.0+3.2 and so on.

In the following example, the .bp file looks like:

```
3.1+4.2
3.2+4.2
```

Valid relocation choices are:

3.1+3.2+4.2+4.1, 3.1+3.3+4.2+4.1, 3.1+3.3+4.2+4.3 and so on.

© 2021 All Rights Reserved.

`3.2+3.0+4.2+4.1, 3.2+3.3+4.2+4.3` and so on.

Relocation in previous generation systems ignores the second LD's domain in the `.bp` file, and places the job in any of the following domains:

```
3.1+3.2+4.1+4.2
3.2+3.3+4.2+4.3
```

Restricting the Designs to Run on Specific Domains

You can restrict the designs to run on specific logic drawers and domains by using a Board Restrictions (`.br`) file.

Unlike the `.bp` file, which specifies LDs or domains into which to download the design, the `.br` file specifies the LDs and domains not to be used. When a `.br` file is present, the design will download onto any domains except those specified in the `.br` file.

When both `.bp` and `.br` files are present, the `.bp` file is ignored and only the `.br` file is considered. The `.br` file is manually created using a text editor. The LDs and domains in the <*design*>`.br` file should be listed in a single line separated by "+" or white space as specified in the following syntax:

```
<br_list>::= <phy_brd> " " <br_list> |<phy_brd> "." <phy_domain> " " <br_list>
```

For example:

0 3 4.5 4.6 5.2

Note: Both `.bp` and `.br` file contents must be in ascending order.

The run-time software searches for the `.bp` and `.br` files in the following order:

```
./<emulator>.<design>.<.bp|.br>
./<alias>.<design>.<.bp|.br>
./<workstation>.<design>.<.bp|.br>
./<design>.<.bp|.br>
```

Using Multiple .bp and .br Files

You can specify multiple `.bp` and `.br` files in different ways. The files can be named using any of the following formats and must be located in the design directory:

```
./<emulator>.<design>.<.bp|.br>
./<alias>.<design>.<.bp|.br>
./<workstation>.<design>.<.bp|.br>
./<design>.<.bp|.br>
```

where,

- <*emulator*> is the machine where the design runs

© 2021 All Rights Reserved.

- <*alias*> is an alternate name for the emulator as defined in the Emulator Workstation Mapfile.
- <*workstation*> is the workstation on which DBEngine runs.
- <*design*> is the name of the top cell in the design.

The run-time software searches for the `.bp` and `.br` files in the following order:

```
./<emulator>.<design>.<.bp|.br>
./<alias>.<design>.<.bp|.br>
./<workstation>.<design>.<.bp|.br>
./<design>.<.bp|.br>
```

Allowing Mixed Board Domain Syntax in .bp and .br

`.bp` and `.br` ranges with mixed from/to "`b`" and "`b.d`" syntax is now supported. So now there are three ways to specify the syntax of `.bp` and `.br` ranges as demonstrated in the examples below:

Example 1:

LONG: 1.0+1.1+1.2+1.3+1.4+1.5

SHORT: 1.0-1.5

COLLAPSED: 1-1.5 (New syntax)

Example 2:

0.4+0.5+0.6+0.7+1.0+1.2+1.3+.14+1.5+1.6+1.7

0.4-0.7+1

0.4-1 (New syntax)

Example 3:

0.0+0.1+0.2+0.3+0.4+0.5+0.6+0.7+1.0+1.1+1.2+1.3

0+1.0-1.3

0-1.3 (New syntax)

© 2021 All Rights Reserved.

Using the Reserved Resources for Downloading the Design

Before starting to download the design, to override the key value recorded in the `./dbFiles/ts.<username>.resv.key` file, or to manually specify the key value if it was not saved to the key file by `test_server` earlier, use either of the following methods:

- Specify the following run-time parameter after invoking xeDebug:

  ```
  xeset reserveKey <key_value>
  ```

- Set the following environment variable before invoking xeDebug:

  ```
  setenv XE_RESERVE_KEY <key_value>
  ```

Specifying the key value manually is useful when the `test_server` program and the xeDebug tool are run from different work directories. For example, when the `test_server` program needs to be run outside of the design directory.

When key value is not specified manually, it is assumed that the `test_server` program was run from the design directory. In this case, xeDebug automatically reads the key value from the key file, provided it exists and has at least one key value entry.

If the key value is not specified manually, and the key file either contains no values or does not exists, the design download proceeds as if no resources were reserved.

To use the reserved resources, perform the following actions before the specified time limit expires, or else the reserved resources will be released for other users:

- Download the design by using the:
  - *Download* button in the xeDebug GUI
  - `download` command
- Swap the design into hardware by using the `xc on -<hot_swap_mode>` command if design is running in SA mode.

If you reserve two domains using the same key, and then use that key to download two single domain designs, the two single domain designs will be downloaded to the two reserved domains, respectively.

If you reserve two domains using different keys, and then use one of the keys to download a design requiring both domains, the download will fail.

Releasing the Reserved Resources

The reserved emulator resources are automatically released when either of the following situations occur:

© 2021 All Rights Reserved.

- The time limit for resource reservation expires
- The design download process finishes in the single-session mode

However, to manually release the reserved emulator resources, execute the following command:

```
test_server -rmkey [-key <key_value>]
```

Note: If the key value is known, any user can remove the resource reservation made by another user.

The emulator resources corresponding to the specified key value are released. If the `./dbFiles/ts.<username>.resv.key` file is used, the key value is deleted from the file. If the key file has more than one key values, the resources corresponding to the first key value are released, and the first key value is deleted from the file.

If the `dbFiles/ts.<username>.resv.key` file does not exist, the `test_server` program exits with an error and displays a message that prompts you to input the key value.

If the specified key value is not associated with any emulator resource, nothing happens, and no warning is generated.

Reducing Emulator Initialization Time Using KeepHostAlive

When you need to run the same design multiple times (for example, while running regression tests), the KeepHostAlive feature can be used to speed up the initial xeDebug connection and design download time. By using the KeepHostAlive feature, you can retain the connection between the host and the emulator even after the design run is finished.

Enabling the KeepHostAlive Feature

Enable the KeepHostAlive feature by running the following steps:

1. Before starting xeDebug, manually run the DBServer executable on the host which will run DBEngine. For details, refer to Setting the DBServer.
2. Start xeDebug and execute the `debug` command.
3. Before downloading the design, use either of the following commands on the xeDebug command prompt:
   a. `xeset keepHostAlive [0 | 1 | 2 | 3 | 4]`
   b. `host <hostname> -keepAlive`

© 2021 All Rights Reserved.

Note: Alternatively, before invoking xeDebug, set the `XE_KEEP_HOST_ALIVE` environment variable to either `0`, `1`, `2`, `3`, or `4`. For example, in the C shell, set the environment variable as follows:

```
setenv XE_KEEP_HOST_ALIVE 1
```

The level of restrictions is the highest when `1` is specified and lowest when `4` is specified. For details about the restrictions associated with each value, refer to the `keepHostAlive` run-time parameter in the `xeset` command description, *Run-Time Commands* chapter, *VXE Command Reference Manual*.

To use the KeepHostAlive feature in IXCOM-compiled designs, set the `xeset keepHostAlive` run-time parameter to either `2`, `3`, or `4`. For designs compiled in ICE mode, it can be set to either `1`, `2`, `3`, or `4`.

When the `host` `<hostname> -keepAlive` command is used to enable this feature, the value of `xeset keepHostAlive` run-time parameter is set to `2` by default.

When KeepHostAlive feature is enabled, even when xeDebug exits, the hardware resources on the emulator, for the particular design, remain unavailable, for a certain amount of time-out period (default is 180 seconds). Start the new xeDebug process before this time-out period ends to reuse the existing hardware resources from the previous xeDebug session. This saves the initial connection and design download time. The design is reset to its initial state in the emulator hardware for each new xeDebug process.

The time-out period value can be controlled by using either of the following methods:

- Set the `keepHostAliveTimeout` run-time parameter at the xeDebug command prompt as,

  ```
  xeset keepHostAliveTimeout <value>
  ```

- Set the `XE_KEEP_HOST_ALIVE_TIMEOUT` environment variable before starting xeDebug. For example, in the C shell, set the environment variable as,

  ```
  setenv XE_KEEP_HOST_ALIVE_TIMEOUT <value>
  ```

The time-out value set using the `xeset keepHostAliveTimeout` run-time parameter is given priority if both the commands mentioned above are used with conflicting values. Also, the value of the run-time parameter can be updated anytime before exiting the xeDebug session.

Important

When the KeepHostAlive feature is enabled, executing the `test_server` program to check the host and domain availability displays the used domains as occupied by the design being run.

© 2021 All Rights Reserved.

Setting the DBServer

DBServer is a daemon process you can optionally run on DBEngine to get direct control on spawning the DBEngine process. The DBEngine process is the process that communicates directly with the emulator and the xeDebug process (which can run on a different workstation).

By default, if you do not start DBServer manually, xeDebug starts DBEngine on the host workstation using a remote shell. In some cases, when this operation fails — for example, due to file system mount point naming issues — you can start DBServer manually on the host workstation. Then, DBServer becomes responsible for spawning the DBEngine processes, as needed.

To manually start the DBServer executable, use the following process:

1. Set the `XE_DB_SVR_RPC_NUM` environment variable before starting the DBServer executable. Use the following command in the xeDebug and DBServer command shells to set this environment variable:

   ```
   setenv XE_DB_SVR_RPC_NUM <unique_number>
   ```

   The value of the `XE_DB_SVR_RPC_NUM` environment variable must be unique for each user. The DBServer can handle multiple xeDebug processes if the design directory path is different. Multiple users can also access the same design database, using a separate DBServer process for each user.

   Note: In the DBServer command shell, user can also set the following environment variable to print detailed information when the design is running:

   ```
   setenv XE_DB_SVR_DISP_MSG 1
   ```

2. Start DBServer using the following command at the UNIX or Linux command prompt:

   ```
   DBServer &
   ```

Note: To use the `keepHostAlive 4` run-time parameter of the `xeset` command, all users of the design must set the same value of the `XE_DB_SVR_RPC_NUM` environment variable.

Disabling the KeepHostAlive Feature

To disable the KeepHostAlive feature, either set the `keepHostAlive` run-time parameter to `0` or execute the `host -kill` command. Consequently, the emulator resources become available to be taken by another user's design at the termination of the current job.

© 2021 All Rights Reserved.

Terminating KeepHostAlive Reservation Preemptively

Multiple DBEngine processes running on a host emulator using the KeepHostAlive feature can prevent a new xeDebug session from starting for a different design. In such situations, the DBEngine processes can be terminated preemptively to release the emulator resources for optimized use.

When the KeepHostAlive feature is enabled, DBServer records each DBEngine process it spawned. Also, DBServer tracks each DBEngine status and the domains it downloaded.

When a new xeDebug session connects to DBServer, it checks all DBEngine processes that are kept alive, and executes the following steps:

1. DBServer checks whether there is a DBEngine process that can be reused according to the keepHostAlive run-time parameter value. If such DBEngine process exists, reuse it; otherwise, execute steps (2) and (3) discussed below.
2. If none of the existing DBEngine processes can be reused, DBServer identifies the DBEngine processes that occupy the domains requested by the new xeDebug session. If such processes are identified, DBServer kills them and releases the corresponding domains.
3. If the number of domains released is less than the number of domains requested by the design in the new xeDebug session, DBServer kills other DBEngine processes too to release the sufficient number of domains.

This preemptive termination of KeepHostAlive reservation improves the efficiency of using the emulator hardware in multi-design debug environment.

Using Suspend and Resume to Manage Emulator Resources

The suspend operation releases the design from the emulator, while resume restores the design state to the emulator. The following sections discuss the suspend-resume operations in the SA and ICE modes in detail.

Suspending and Resuming Runs in SA Mode

In the SA mode, you can suspend and resume emulation using one of the following methods:

- File-based (asynchronous) suspend-resume of an SA run-time job with **xrun** and xeDebug tools
- Command-based (synchronous) suspend-resume with xeDebug tool

© 2021 All Rights Reserved.

File-based Suspend-Resume in the SA Mode

To enable suspend-resume asynchronously, the running job must periodically check for the existence and non-existence of a file. Note that this status check of a file introduces a small overhead in the run-time performance.

This method involves:

- ❑ Enabling File-based Suspend-Resume in the SA Mode
- ❑ File Indicating Suspending State

Note: When the hardware is in suspended state, the simulator enters into a loop to check for the existence of the suspend file. This software loop stops all simulation activity during the time of suspension. The simulator exits the loop only after the suspend file has been removed and the emulator is re-occupied.

Enabling File-based Suspend-Resume in the SA Mode

In the SA mode, you can enable the suspend-resume operation in two ways:

- Using the `+allowSuspend` option

  Add the `+allowSuspend` option to enable the suspend-resume operation when invoking the run as follows:

  When you use the **xrun** tool, specify the `+allowSuspend` option as follows:

  ```
  xrun +allowSuspend
  ```

  When you use the xeDebug tool, specify the `+allowSuspend` option as follows:

  ```
  xeDebug --xmsim +allowSuspend --
  ```

- Using the `xeset suspendEnable 1` run-time command

During the run, the following message is printed on the stdout and in the run log to indicate that the job can be suspended:

```
INFO(xc): to suspend hardware, touch <run_dir_path>/tmp/suspend
```

Controlling File-based Suspend-Resume in the SA Mode

After a local user enables the suspend-resume operation by using the `+allowSuspend` option or using the `xeset suspendEnable 1 command`, a remote user can suspend the emulator run by using the suspend file. Create the suspend file by using the UNIX `touch` command:

© 2021 All Rights Reserved.

```
touch <run_dir_path>/tmp/suspend; [chmod og+w <run_dir_path>/tmp/suspend]
```

Note: The `chmod og+w <run_dir_path>/tmp/suspend` command is not needed if the remote user has set up the file-creation mask correctly by using the UNIX `umask` command. This permission is required only to ensure that after the job has been run, the local users can clean up their directory.

When the suspend file is detected, the running process releases the hardware, and enters into a loop to keep checking if the suspend file has been removed. Whenever it is detected that the suspend file has been removed, the hardware run is resumed.

Remove the suspend file using the UNIX `rm` command:

```
rm <run_dir_path>/tmp/suspend
```

Unlike the ICE mode, in the SA mode, the suspend file in the `tmp` directory is used because the `tmp` directory is a run-time directory while the `dbFiles` directory is a compile-time directory, which might only be shared, and might not be writable at run time.

Note: The `<run_dir_path>/tmp/suspend` file is removed at the start of simulation to prevent residual effect from previous runs in the same directory.

To identify the current status of the suspend-able processes, use the `test_server` command. In the `test_server` output, the suspend-able processes are marked with a suffix `:S` after the job name. The sessions running in the SA mode are marked in the `test_server` output with a prefix `SA:` before the job name.

File Indicating Suspending State

Touching the suspend file does not necessarily mean you can suspend the job. It is just a request. The job itself must be first enabled for suspension with the `+allowSuspend` option or `xeset suspendEnable 1` command.

If the job is identified as suspend-able, and a suspension request is accepted, a file `<run_dir_path>/tmp/suspending` is created at the start of suspending the hardware job. After the job is resumed, the file is removed. This file is useful for third-party job scheduler to detect whether a hardware job is indeed suspending.

The suspending file `<run_dir_path>/tmp/suspending` is removed at the start of simulation to prevent residual effect from previous runs in the same directory.

© 2021 All Rights Reserved.

Command-based Suspend-Resume with xeDebug Tool in the SA Mode

When a run-time job is suspended, the emulator is released. While the hardware is suspended, the software simulator is not supposed to advance the testbench time, and you cannot access the hardware.

With the xeDebug tool, suspend the running job by issuing the `suspend` command. After this, you cannot issue the `run` command or any other hardware-related commands during the time that the emulator is suspended. The only commands that are allowed after the `suspend` command are the `resume` and `help` commands. In the xeDebug GUI, the *Suspend* and *Resume* buttons provide the same functionality as the `suspend` and `resume` commands.

Note: The command-based and GUI-based suspend-resume operation is entered from the xeDebug prompt. To bring up the user prompt, you must do one of the following:

- Interrupt a run which stops the clocks and gives user prompt. OR
- Issue the `run -nowait` command that does not stop the clocks and displays the user prompt.

Caution

The command-based suspend-resume operation is supported only with the xeDebug tool. It is not supported with the xrun tool.

Important

Certain jobs cannot be suspended:

- If the emulator is controlling a dynamic target, the job cannot be suspended and resumed because the target requires constant clocking from the emulator. Therefore, the suspend-resume operation cannot be used with the following command:

  `configPM -dynamic`

- Designs having an uncontrolled clock specified using `ixc_hwclksrc`.
- Designs not running in the hardware mode.

  If you swap out the design to the software mode, and do not intend to swap in to hardware mode again, then, use the `xc free` command to release the hardware.

Suspending and Resuming Designs in the ICE Mode

Enable the suspend-resume functionality during the run by using the following command:

© 2021 All Rights Reserved.

```
xeset suspendEnable 1
```

You can suspend and resume an emulation run without stopping the clocks.

Note: The command-based and GUI-based suspend-resume operation is entered from the xeDebug prompt. To bring up the command prompt, you must do one of the following:

- Interrupt a run that stops the clocks and gives user prompt.

OR

- Issue the `run -nowait` command that does not stop the clocks and displays the user prompt.

The following sections describe the suspend and resume operations in the ICE mode.

Suspending the Design from the Emulator

Suspending a design run releases the emulator. The suspend operation saves the design snapshot and stores it in <*run_dir_path*>`/tmp/suspend` file. You can set the design prototype, memory content, and all the set or force signals in the file. The suspend operation also saves the SDL control signals. To prevent altering the hardware state, the hardware-related commands and xeDebug GUI tabs are disabled when a run is suspended. If a disabled command is executed, an error message is generated.

Note: In the ICE mode, the design snapshot is also saved in the `./dbFiles/suspendSession.qt`<*PID*> file for backward compatibility.

You can suspend a design running on an emulator by using one of the following methods:

- Executing the `suspend` command on the `xeDebug` command line in interactive mode.

  Note: Refer to the description of the `suspend` command in the *Run-time Commands* chapter of the *VXE Command Reference Manual*.

- Invoking the `xeSuspend` script when running the design in batch mode. This script can also be used in interactive mode; however, its use in batch mode is strongly recommended. For details, refer to Using xeSuspend Script to Suspend Emulator in the ICE Mode.

  Note: The `xeSuspend` script is used to suspend and resume the design remotely.

- Using a <*run_dir_path*>`/tmp/suspend` file to interrupt the emulation in batch mode during the same xeDebug session.

  Note: To use this file, the `suspendEnable` run-time parameter must be set to `1`.

  ```
  xeset suspendEnable 1
  ```


© 2021 All Rights Reserved.

- Clicking the *Suspend* button on the *Setup* tab if using the xeDebug GUI to run and debug the design.

Using xeSuspend Script to Suspend Emulator in the ICE Mode

The `xeSuspend` script can be used in both interactive mode and batch mode. In the batch mode, it primarily automates the use of the `suspend` and `resume` commands.

Depending on the configuration and platform, the `xeSuspend` script might need modification. The format of the script is:

```
xeSuspend <emulator> <PID> -suspend | -resume
```

The `xeSuspend` script is provided in the following directory:

```
/<VXE_INSTALL_HOME>/share/vxe/gift/misc/xeSuspend
```

For the `xeSuspend` script to successfully suspend the design, the following conditions must be true, otherwise the suspend request is ignored with an error message:

- Emulation clock must be running in either blocking or non-blocking mode.
- You must have permission to create files in the design directory.
- The design session to be suspended must have the `suspendEnable` run-time parameter set as `1`.

  Note: The default value of the `suspendEnable` run-time parameter is `0`, which prevents emulation from being suspended remotely. In this manner, the existing user can prevent the xeDebug session from being suspended by any other user using the `xeSuspend` script.

Resuming the Design State to the Emulator in the ICE Mode

Use the `resume` command to resume the design state to the emulator. Alternatively, if using the xeDebug GUI, click the *Resume* button on the *Setup* tab.

On resume, the emulation `run/share/vxe` continues from the same cycle, as the one before suspend. Resuming a session re-downloads the design and restores the last snapshot taken by the `suspend` command. Memory content, all SDL control signals, and all set and forced signals are restored.

© 2021 All Rights Reserved.

Important

The SDL state and all the triggers of the design are maintained during suspend or resume of emulation. This means, if `run` command is executed and the design gets suspended before the trigger, the trigger takes place when the design is resumed and the trigger condition holds true. In batch (or script) mode, once the design gets resumed, the remaining XEL commands following the `resume` command get executed normally.

Infinitrace session is also supported during suspend and resume, irrespective of emulation clock being run or stopped during a call to suspend.

On exiting a xeDebug session, all suspend or resume operations are cleared and the `suspendSession.<PID>` directory is deleted.

For more information on the `resume` command, refer to the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Examples of Suspending and Resuming the Design in the ICE Mode

The following example shows the use of the `suspend` and `resume` commands:

```
XE> debug .
XE> host emulator1
XE> download
XE> configPM -stb
XE> run -nowait 50000000        <<starts non-blocking run for 50M cycles
XE> suspend                <<suspend emulation and release emulator
XE> getCycleNum        <<suppose 20M cycles done when suspended
XE> resume  <<reload design from last stopping point, and continue remaining cycles
XE> getCycleNum      <<when previous run complete, getCycleNum would show 50000000
```

The following example shows the use of `xeSuspend` in interactive mode:

```
XE> design top top
XE> xeset suspendEnable 1        <<enables suspend or resume mode
XE> debug .
XE> host emulator1
XE> download
XE> configPM -stb
XE> run 1000000               <<blocking run for 1 million cycles
             <<xeSuspend emulator1 <PID> suspend issued from other xterm
XE> INFO(legacy-xxx): Design is suspended from hardware
             <<xeSuspend emulator1 <PID> resume issued from other xterm
XE> INFO(legacy-xxx): Design is resumed from hardware
```

Saving and Restoring Design States

First issue the `save` command to save the current state. Then, use the `restart` command to restore the previously saved state. Quite similarly, you can use the `suspend` command to

© 2021 All Rights Reserved.

temporarily *freeze* the run and release the emulator resource. Then, use the `resume` command to recapture the emulator and restart the run. However, the `suspend-resume` operation is different from the `save-restart` functionality.

- The `suspend-resume` operation is restricted to the hardware state. On the other hand, the `save-restart` operation works on both the hardware and software states.
- In the `suspend-resume` operation, the software processes must be alive during suspension. Therefore, the Xcelium, AVIP, xmsim licenses are in use while the hardware is suspended. However, MMP and Palladium Z1 domain licenses are checked out when jobs are suspended and checked in when jobs are resumed. Ensure you also check your IB host workstation memory when you suspend the jobs. On the other hand, in the `save-restart` operation, you can terminate the software processes after issuing the `save` command.

The save Command

The `save` command is used to save the current state of the simulator and the emulator. This command works in both ICE and SA modes.

Before issuing the `save` command, you need to stop the emulator.

Use the following command to save the simulator and emulator states:

```
save {<save_name> [-overwrite]}
```

This option specifies the basename of the files in which the information is saved in the ICE mode. When saving the design state, the state information is saved in a few files in the directory named <*save_name*>. For example, you can include the path information `mysave/ss1` as the <*save_name*>. Then, the state is saved in the `mysave/ss1` directory.

The design state will be saved in binary format. The text format to save a design state is obsolete.

The `-overwrite` option is used to overwrite an old state saved with the same name. If you do not specify this option, saving the state with a pre-existing name fails.

The restart Command

The `restart` command is used to reload an existing snapshot that was saved with the `save` command. It is available in both the ICE and SA modes.

Following is the command of the `restart` command:

© 2021 All Rights Reserved.

`restart <snapshot_name>`

Use the <*snapshot_name*> option to specify the checkpoint from which the emulation state needs to be restarted. This is the checkpoint created by the `save` command. Only emulation state is restored.

Compute Farm Control Mechanism

Palladium requires access to a compute farm, meaning a set of servers as well as a third-party software managing the farm. There are different types of farms, such as LSF, NC and so on. Depending on the type of farm, Palladium needs to issue different commands and options. This results in burdening the software layer that interacts with the third-party software. Hence, the need for a unique farm control mechanism that is not dependent on the type of farm being used. The usage of this farm control mechanism can also be extended through different configuration settings.

Important

This new farm control mechanism is in experimental phase. The future ISR builds will have complete support of the new, custom compute farm.

Farm Configuration Settings

To use a farm, you need some kind of configuration setting that is farm, user, tool or design specific and must be provided in a configuration file. The configuration file format supported by xeCompile tool is JSON based as it is easy for the machine to parse. The default name of this configuration file is `computeFarm.json`.

This JSON based farm control mechanism is currently only supported for compiling designs in Modular Compilation.

Refer to Modular Compilation in ICE Mode section to know more about Modular Compilation.

Following attribute-value pairs are recognized by the farm control mechanism, so use these attributes/options while creating a custom farm:

© 2021 All Rights Reserved.

Table 6-1 Global Attributes

Attribute Name	Value Type	Description
farmName	String (LSF, NC, custom)	The farm control mechanism processes the configuration and errors out if the farm name is not recognized
farmOptions	Object	See Farm Options Attributes table on page 175 for more information.

Table 6-2 Farm Options Attributes

Attribute Name	Value Type	Description
FullDesign	String	Used when a job processes the full design
Partitions	String	Used when a job processes a PPC partition
Buckets	String	Used when a job processes a bucket in Modular Compilation (MC)
Import	String	Used for designImport jobs
dbgPrepare	String	Used for dbgPrepare pass1 jobs

Example 6-1 Sample computeFarm.json file used in Modular Compilation:

```
{
  "farmName" : "LSF",
    "farmOptions" : {
      "Partitions" : "-M 100000 -W1:00",
      "Buckets" : "-M 500000 -W3:00",
      "Import" : "-M 20000 -W0:30",
      "dbgPrepare" : "-M 400000 -W2:00"
    }
}
```

Example 6-2 Sample computeFarm.json file for NC farm type:

```
{
  "farmName" : "NC",
    "farmOptions" : {
      "Partitions" : "-D",
```


© 2021 All Rights Reserved.

```
      "Buckets" : "-D",
      "Import" : "-D" ,
      "dbgPrepare" : "-D"
    }
}
```

Example 6-3 Sample computeFarm.json file for LSF farm type:

```
{
  "farmName" : "LSF",
    "farmOptions" : {
      "Partitions" : "-M 150000 -R \" select[OSNAME==Linux &&
(OSTYPE==CL&&(OSMJR==6))||(OSTYPE==RL&&(OSMJR==5||OSMJR==6||OSMJR==7))||(OSTYPE==
SL&&(OSMJR==11))] rusage[mem=40000]\"  -q secondary",
      "Buckets" : "-M 150000 -R \" select[OSNAME==Linux &&
(OSTYPE==CL&&(OSMJR==6))||(OSTYPE==RL&&(OSMJR==5||OSMJR==6||OSMJR==7))||(OSTYPE==
SL&&(OSMJR==11))] rusage[mem=40000]\"  -q secondary",
      "Import" : "-M 200000 -R \" select[OSNAME==Linux &&
(OSTYPE==CL&&(OSMJR==6))||(OSTYPE==RL&&(OSMJR==5||OSMJR==6||OSMJR==7))||(OSTYPE==
SL&&(OSMJR==11))] rusage[mem=50000]\"  -q secondary" ,
      "dbgPrepare" : "-M 150000 -R \" select[OSNAME==Linux &&
(OSTYPE==CL&&(OSMJR==6))||(OSTYPE==RL&&(OSMJR==5||OSMJR==6||OSMJR==7))||(OSTYPE==
SL&&(OSMJR==11))] rusage[mem=40000]\" -q secondary"
    }
}
```

Additional Options Supported in Farm Configuration Settings

In addition to user settings required by this farm control mechanism, attributes that are not used for farm operations but somewhat related are also enabled in this farm control mechanism. For example, usage of `maxCumulativeThreads` option to limit the maximum number of threads is supported.

Attributes are also grouped hierarchically to provide global values so they can be refined at lower levels. These groups are “objects” in JSON terminology and they have pre-defined names indicating a tool, such as xeCompile, a flow or a phase of the flow. See the following example to know its usage:

Example 6-4 Optional Farm Attributes

```
{
          “maxCumulativeThreads”: default,
          “tools”: {
               “xeCompile”: {
                   “maxCumulativeThreads”: 256,
```


© 2021 All Rights Reserved.

```
                    "Flows": {
                      "MC": {
                         "phases": {
                            "Import": {
                                 "maxCumulativeThreads": 32
                            },
                            …
                      }
                    },
                }
                "xeDebug": {
                …
                }
            }
}
```

Additional Settings for NC Compute Farms

To use NC (Network Computer) farm type with modular compilation, ensure you add the following additional settings:

- `set NC_QUEUE` variable to a valid NC farm queue name.
- `setenv NC_SNAPSHOTDIR=`"*homedir*"

Note: These settings are not mandatory and is dependent on customer's farm setup.

Using Custom Compute Farms

If you have no access to any supported farm, use reserved `farmName` value "custom". To use this custom farm, ensure the user configuration file contains the following attributes that enable proper mapping so that the custom farm functions properly.

Table 6-3 Custom Farm Mapping Options

Attributes	Value Type	Description
`submitCmd`	String	Name of the command to submit a job
`getJobInfoCmd`	String	Name of the command to get information about a job
`killJobCmd`	String	Name of the command to kill a job

© 2021 All Rights Reserved.

Note: Each of these commands needs to be provided as TCL procs.

© 2021 All Rights Reserved.

Balancing Load Using Load Sharing Facility

Load Sharing Facility (LSF) is a job scheduling and management product from IBM. LSF is very effective at distributing user jobs across a group of servers, managing software licenses, managing emulation hardware jobs, and more. Based on the job specifications, LSF starts execution of jobs when there are enough system resources available for a job to complete. The jobs remain pending in a queue until the specified resources become available.

In VXE, LSF is used for various purposes:

- Distribute compiles with different seeds among different computers: For large designs, use the `-lsf` option of the `compileFind` command, to distribute compiles with different seeds among different computers. For information on how the `-lsf` option works and for information about the command submitted to LSF, refer to the “Using LSF to Distribute Compiles with Different Seeds among Different Computers” section.
- Distribute RTL synthesis: The distributed synthesis feature of the HDL-ICE Compiler is useful for synthesizing VHDL or Verilog RTL designs containing tens of million gates. The feature enables distribution of the synthesis process into tasks at module level. The tasks are dynamically allocated during run time to balance the load across multiple workstations or processes. For details, refer to the “Using Load Sharing Facility (LSF) Queue” section.
- Perform DPA operations with multiple hosts: DPA operations need a huge amount of computation. The computation work can be distributed to multiple hosts to speed up the process. You can specify the set of hosts using LSF. For details, refer to the “Specifying Multiple Hosts with LSF” section.
- Parallel import: By default, the netlist import distributes itself in multiple threads on the same host, if it can. Optionally, it can distribute itself on multiple hosts through LSF. To enable multi-host parallel import with LSF, use the `designImport -useLSF` command. For more information, refer to the “Using Parallel Import” section.

Note: The LSF software must be properly installed and configured on the emulator host. For details, consult your LSF administrator.

Handling Failures at Run Time Using LSF

When LSF version 9.1.1.1 is used to manage the run-time execution, Cadence recommends that the system locked memory be set to `unlimited`. Otherwise, the design download might fail with the following error message:

```
ERROR (legacy-43984): 20 DATABASE | 122: PDXETA0122E Board 0(0), Chip 0 not loaded
due to error.
```


© 2021 All Rights Reserved.

To fix the issue:

1. Edit or verify the `/etc/security/limits.conf` file on the emulator host(s) to ensure it contains the following lines:

```
*     -     memlock     unlimited
*     softmemlock     unlimited
*     hardmemlock     unlimited
```

   Note: The asterisks are part of the required syntax in the `/etc/security/limits.conf` file.

2. If the `/etc/security/limits.conf` file did not contain the above lines and the lines have now been added, then reboot the emulator host(s).

3. If the issue persists after confirmation of the required lines in the `/etc/security/limits.conf` file and after the reboot, shut down the LSF daemons on the emulator host(s) and start them up again.

Caution

Do not simply restart the daemons as this will not fix the problem. They must be shut down completely, and then started again.

For example (as root):

```
# badmin hshutdown
# lsadmin resshutdown
# lsadmin limshutdown
```

Alternately, you can kill all the LSF daemon process ID's on the emulator host(s).

It might take a few seconds for the daemons to stop when they are shut down. Wait for 30-60 seconds or confirm all processes have exited before proceeding further.

Start up the LSF daemons on the emulator host(s). As root, do as follows:

```
# lsadmin limstartup
# lsfadmin resstartup
# badmin hstartup
```


© 2021 All Rights Reserved.

Tracking Emulator Jobs

The system enables you to track the emulator jobs using a unique identifier. The unique identifier remains constant through the life of the job and is easily mapped into the job information captured in the Configmgr logs and displayed in the test server output.

You define this identifier using the `XE_JOBID` environment variable, as follows, before invoking xeDebug or xrun:

```
sh> export XE_JOBID=<job_name>
```

or

```
csh> setenv XE_JOBID <job_name>
```

Caution

The value of `XE_JOBID` ***should not contain any whitespace or non-printable characters. Otherwise, the system displays the following warning message and the value is ignored.***

```
WARNING (legacy-131313): Value '<vals>' for XE_JOBID environment variable
contains contains illegal characters and will be ignored.
```

You can access `XE_JOBID` through the test server utility or `configmgr_activity_log.cmd` file.

Note: The *`XE_JOBID`* environment variable does not affect the output of the xeGenReport to report the Palladium XP weekly usage by designs. The xeGenReport only uses the real design name for the output.

Example 1: test_server Output

```
setenv XE_JOBID LSF13
```

The output of the `test_server` command would be as follows:

```
Domain Owner PID BCA J-con MemCard Design ElapTime ReservedKey
0 artj cva-xeon152:10732 -- -- -- top:LSF13 00:53:50 --
```

Here, `top:LSF13` is the design name with the `XE_JOBID` suffix. The design name remains the same when the job is running and is derived from the top level module name.

The format for the design name field for an SA job is:

```
[SA:]<designName>[:S][:<XE_JOBID>]
```

where,

© 2021 All Rights Reserved.

- `SA:` Indicates a simulation acceleration job
- `:S` Indicates a suspendable job
- *`XE_JOBID`* is the value of the *`XE_JOBID`* environmental variable

If *`<XE_JOBID>`* is not set or set to an empty string, the suffix will be omitted. The other design modifiers might still be present and will retain their relative position within the design string.

In the above example, for an SA job, the full design name would be `SA:top:LSF13.`

If the job is suspendable, the design name would be `SA:top:S:LSF13.`

You can override the design name using the following command:

```
xeset designName <name>
```

Example 2: Configmgr Logs

In STB Mode

The value of *`<XE_JOBID>`* will be saved in the Configmgr logs at `configmgrhost:/et3mach/log/*.joblog` files.

1. Set the `XE_JOBID` environment variable:

   ```
   setenv XE_JOBID 54321
   ```

2. Download the design

   ```
   host .
   download
   ```

The output of `configmgr_activity_log.cmd` includes the job name, `54321`, as shown below:

```
2014/08/06 18:15:51 | session=13188 | user=garyw | JOB_START | PID=20883 |
host=158.140.188.21:eday | job_name=counter:54321 | design=counter |
<etc>
2014/08/06 18:16:26 | session=13188 | user=garyw | JOB_END | PID=20883 |
host=158.140.188.21:eday | job_name=counter:54321 | design=counter |
<etc>
```

In SA mode

```
setenv XE_JOBID 12345
2014/07/28 06:47:34 | session=12029 | user=garyw | JOB_START | PID=5745 |
host=158.140.42.27:cva-xeon144.Cadence.COM | job_name=SA:counter:12345 |
```


© 2021 All Rights Reserved.

```
design=xcva_top |
```


© 2021 All Rights Reserved.

Checking Emulator Status

The emulator status GUI provides an overview of the status of the Palladium Z1 system and lists links to various logs that provide more detailed status.

To invoke the Emulator Status GUI, use the following link:

`http://<scdname>.<domain>.com`

Here, <*scdname*>.<*domain*>.com is the name of the emulator, for example `hsv-scd105.cadence.com`. If the emulator is on a local private network rather than your corporate network, then you must start a web browser on a host on that local network, and use `http://<scdname>` instead of `http://<scdname>.<domain>.com`. Point the browser at the name of the desired emulator to bring up the page. The web page displays the output of a program (`statusmon.rexx`) that collects information on the emulator status from various files and components. The `statusmon` program queries the `configmgr` and `apsm`, and checks the various board manager files and power monitor files. It also checks the InfiniBand network status and creates a file `/et3mach/status.txt` that provides the information for the emulator status web page. You can specify a different location or file to display the desired information via the GUI.

For example, the following command uses `/et3mach/statusxyz.tx`t:

```
http://hsv-scd105.cadence.com/pdz/mainjob.php?scd=/et3mach/statusxyz.txt
```

The Apache web server runs on the SCD. The `statusmon` program collects new data every minute. The web page also auto-refreshes every minute. A timestamp at the bottom of the page shows the time at which the data was collected. If you see any errors in the emulator status, such as incorrect status, note the time, date, and emulator system. You can also run `pdz-report` script to collect the log files. For details about running this script, refer to the *Running pdz-report Script* section in the *Installing the Software* chapter of the *VXE Planning and Installation Guide*.

Firefox or Chrome are recommended browsers for viewing the emulator status GUI. In narrow window displays, the racks and tables might appear in columns through which you can scroll to view the required information. Using the browser zoom-in and zoom-out features might cause the display to be distorted. The number of racks listed in the GUI corresponds to the number of racks in each system.

The following figure displays the status of a four-rack system.

© 2021 All Rights Reserved.

Figure 6-19 Emulator Status GUI

The racks are arranged in the order in which these were physically installed. In this case, it is RACK 2, RACK 0, RACK 1, and RACK 3. Within a rack, the clusters are listed from top to bottom with their details about power supplies, boards and CCD. Each board has eight squares on it; each square represents a domain. There are 24 squares on the CCD representing its targets. The squares, the circles that show the status of the boards, SCD, CCDs, Power, and ACU/WCU are referred to as *LEDs*.

Job Sessions and Host Details Tables

Next to the rack-representations on the web page, you can see the jobs table that lists the sessions and their details. This table lists all the jobs currently running on the emulator.

The jobs table shows the session ID, user name, number of domains, and the time that has elapsed since the job started running on the system.

© 2021 All Rights Reserved.

Position the mouse over the session ID on the job table to view more details about the job. Click on the session ID to open a separate menu with the latest information on the job.

Next to the jobs table is the hosts table that lists all the hosts that are connected to the SCD on the IB network. Note that other emulators might be on the same IB network; but these are not shown on this list. An entry in the host table shows the number of domains that are running diagnostics. Hover the mouse over any line in the jobs table or a host name in the hosts table to temporarily highlight the domains that are:

- Either occupied by that job session (domains are highlighted in purple color)
- Or associated with all sessions running from a specific host

You can sort these tables by clicking on the required column heading.

Figure 6-20 Job and Host Tables in the Emulator Status GUI

© 2021 All Rights Reserved.

Displaying Messages from Emulator Administrator

Between the red title bar and the rack display there is an area for status messages about the emulator. Emulator administrators can use this area to display important messages. The title of each message is limited to 80 characters. About four messages can be displayed at a time.

Figure 6-21 Messages from Emulator Administrator - MOTD

These messages originate from the files in the `/et3mach/motd` directory. The files must be named as `msg<N>.txt` where <*N*> is the message number, such as `/et3mach/motd/msg1.txt`. The first line of the file contains the title.

Click on a message line to open a new tab that shows additional message lines from the file.

The `motd` directory needs to be manually updated. The administrator must log on to the SCD to copy a file into the motd directory or delete a file.

Viewing Domain and Target Status

Each *Board* row of the status page shows eight boxes—one for each domain on that board. These are numbered 0 to 7, left to right.

Each CCD row of the emulator status page shows 24 LED boxes for targets. These boxes are numbered 0 to 11 in the first row, and 12 to 23 in the second row.

The LED boxes for the domains and targets appear in one of the following colors:

- Gray color to indicate uninstalled components

© 2021 All Rights Reserved.

- A shade of green to indicate installed components
- Other colors that are explained in the mouse-over-text, which displays when you position the mouse over a box

Colors of LED Boxes in Emulator Status GUI

In general, the green color indicates that the component is functioning well, gray color indicates that the component is not installed. Amber and yellow colors mean that the component is not fully operational. Red color means that the component is marked off or it is not functioning. Hovering the mouse over the LED displays the details of the status of the component.

Shades of Green for Domains

Bright Green means the domain is locked to a job. Dull green signifies the domain is online and idle and waiting for a job to be assigned to it. An in-between shade indicates that the domain is reserved, but the job has not locked the domain.

Shades of Amber

Amber means that something is wrong. For example, the CCD LED is amber if both the IB links are not up and operating with the right number of lanes and speed. The power LEDs are amber if a rectifier is offline or missing. Note that the amber light appears more like orange color, depending on your vision and display.

Shades of Yellow

The disabled domains that are not locked by a job or diagnostics are displayed as yellow. If the domains are locked, positioning the mouse over these displays the status as *Locked+Disabled*.

If a board is marked as Not Ready, its domains display a dull yellow.

Purple

Purple color is used in conjunction with mouse-over actions on the jobs and host tables. When you move the mouse over a job or a host, the domains assigned to that entry are temporarily highlighted in purple.

© 2021 All Rights Reserved.

System Log Files and Hardware Configuration Files

In the upper right corner of the emulator status GUI is the *Logs* menu. Use the pull-down menus in the *Logs* menu to see the various system-log files.

Figure 6-22 Logs Menu in Emulator Status GUI

Click on these logs to open a new tab with links that display the system logs. The *Hardware* menus display the `scd.et3confg` named emulator-configuration file, and a summary of offline resources as specified in that file.

© 2021 All Rights Reserved.

Setting Email Alerts for Error Messages

You can use this function to send Email alerts whenever certain error messages or a class of error messages are logged in the configmgr.alertlog file on the SCD.

To configure, click the envelope icon in the upper right corner of the emulator status GUI.

Figure 6-23 Email Alert Icon

Clicking the envelope icon opens the Alert Monitor page with help text and an entry area at the bottom of the page. Enter email addresses of persons to whom you would like to send the email alerts, in that blank entry area.

Figure 6-24 Alert Monitor Screen

© 2021 All Rights Reserved.

Email addresses entered in this file receive emails every time any error message of any type is written to `/et3mach/configmgr.alertlog` file, unless “selection rules” are included.

As shown in the two example lines at the bottom of the Alert Monitor page, you can limit the types of error messages which trigger emails for each person by coding these selection rules. The selection rules only involve the message IDs themselves, never any part of the message text.

First, examine the `/et3mach/configmgr.alertlog` file on your Palladium Z1 system to determine the types of error messages you would want to receive emails about and the ones you want to ignore. The log can be viewed from the **Logs** menu item located in the upper right corner of the emulator status GUI.

Figure 6-25 Logs Menu

© 2021 All Rights Reserved.

The PDZ message IDs are of the following form:

`PDZcccnnns`

where,

- `ccc` is the component code
- `nnn` is the message number
- `s` is the message severity (S, E, W, I)

The following table describes the component codes, indicated as "`XXX`" in `pdzXXX`, and the message severity indicated as "`S`" in `pdzxxxnnnnS`, of an error message:

Table 6-4 Component Codes Used in Error Messages

Component Codes	Component
`bma`	ACU cooling unit
`bmw`	WCU cooling unit
`bmk`	AKU cooling unit
`bml`	Logic Drawers

© 2021
All Rights Reserved.

Component Codes	Component
bmt	Target Pods
bmc	CCD
bms	SCD
bmu	Board Manager
eta	Runtime - ETAPI
pwr	Power Monitoring
aps	APSM
dgs	Diagnostics
lnx	Linux
tpd	Target Pod Diagnostics
cfg	CONFIGMGR

Table 6-5 Message Severity Description

Message Severity	Definition
S	Severe
E	Error
W	Warning
I	Info

After deciding the types of alert messages you would want to receive or not receive emails about, code all the message IDs on one line after the user's email address on the Alert Monitor page. Following are a few examples:

This will ignore all "info" alert messages from all component types:

```
user@company.com -i
```

This will ignore all "warning" alert messages from all component types:

```
user@company.com -w
```

This will ignore all alert messages for ACU cooling units:

© 2021 All Rights Reserved.

```
user@company.com -pdzbma
```

To view the description and explanation of any Palladium Z1 message that appears in any log, click the message number when viewing the log using the Web page.

To see all the documented messages, click **PDZ message documentation** in the **Main** pull down menu as shown in the image below:

Figure 6-26 PDZ message documentation Main Menu Item

The Alert Monitoring Function

When Linux starts on the SCD, it automatically starts alert monitoring (alertmon). You can stop or start it as per your need by running either of the following commands as `root` on the SCD:

```
# stop cdn_alertmon
# start cdn_alert
```

To check whether `alertmon.rexx` is actually running, run the following command:

```
# ps -ef | grep alertmon.rexx
```

You should see output similar to the following:

```
root 11386 11384 0 Jul08 ? 00:00:00
/usr/bin/regina/et3mach/bin/alertmon.rexx
```

The alertmon function can be permanently turned off so that it does not restart when the SCD Linux reboots by creating the following dummy file:

```
# touch /et3mach/stop_alertmon.rexx
```


© 2021 All Rights Reserved.

Steps to Configure Email Alerts on the SCD

Note: For sending email alerts from Palladium Z1 SCD, ensure the SCD is not on a private network but on the company's intranet as alertmon uses the Linux `mailx` command to send emails using the company's SMTP mail server.

The FSE, with help from the company's IT department, must follow the steps below to configure the SCD for enabling it to send emails.

A number of files on the SCD must be configured properly so that email alerts can be sent from SCD.

Complete the following steps to enable the sending of alert emails:

1. Edit `/etc/hosts` file to ensure it has a line describing the SCD's hostname complete with the company's domain. For example,

   ```
   158.140.43.81 scd123.companyname.com scd123
   ```

2. Ensure `/etc/sysconfig/network` file has the complete SCD hostname as follows:

   ```
   HOSTNAME=scd123.companyname.com
   ```

3. Save the files and then test with the following two commands:

   ```
   # hostname -d
   companyname.com

   # hostname -f
   scd123.companyname.com
   ```

4. Edit `/etc/mail.rc` file and add a line in the end with the appropriate SMTP mail server for the company as shown below:

   ```
   set smtp=smtp://smtp.companyname.com:587
   ```

   Contact the company's IT department for exact mail server name and port number information.

5. Since you are using a name rather than a numerical IP address in `mail.rc` file, ensure you include the company's DNS server in `/etc/resolve.conf` file. For example:

   ```
   domain companyname.com
   search companyname.com
   nameserver xx.xx.xx.xx
   ```

6. Test whether you can send an email from the SCD using the following command:

   ```
   # mailx -s TEST some-user-name@customer.com < /etc/motd
   ```

7. Ensure the specified email addressee receives the contents of `/etc/motd` file in an email and the sender's email address shows up as `root@scd123.companyname.com`.

© 2021 All Rights Reserved.

Filtering to Reduce the Frequency of Duplicate Emails

Palladium Z1 hardware generates frequent, duplicate error messages that are logged in `apsm.log` file. If these messages qualify as an “Alert” message, then they are copied into the `configmgr.alertlog` file. This may result in a huge number of duplicate emails being sent.

To take care of this, an “apsm message filter” function has been added to Palladium Z1 system to control how often the duplicate error messages logged in `apsm.log` file are copied into `configmgr.alertlog` file.

Cadence recommends that only the FSE member should activate and deactivate any filtering, only if the error messages are understood to be redundant and not a valid cause for repetitious email alerts. For example, one pump in the cooling unit may have failed, and the FSE has ordered a replacement, so repetitious alert messages are not necessary.

Following is an example of filtered message displayed in `configmgr.alertlog` file:

```
PDZBMW0061W HARDWARE WARNING: type "WCU", loc "0.8110.0", IP
"10.0.4.30", dev "pressure" (id19): Low house water pressure (HXB
status 0x20500000). (35 duplicate messages suppressed)
```

Note that, `35 duplicate messages suppressed` at the end of the error message tells us the number of times the exact message has been suppressed.

Note that, all suppressed error messages are in `apsm.log` file. They are only being filtered from `configmgr.alertlog` file.

Configuring the apsm Message Filter

The configuration file for filtering apsm message is called `/et3mach/apsm_msg_filter.txt`.

There is no Web menu nor sample file for this, but following is an example:

```
# Set the default to 12 hours:
default 12:00
pdzbmw0061w 0.8110.0 -t 1:00  // Display this message at most once per hour
pdzbmw0002e 2.8110.0 -t 0:15  // Display this message at most every 15 minutes
pdzbmw0011w 4.8110.0  //  Display this message every “default” (12 hours)
```

The first token of a line is either a PDZ message number or the word “default” or a comment (*,//, or #).

© 2021 All Rights Reserved.

"default 12:00" means that subsequent messages in the file are displayed only once every 12 hours (unless otherwise specified).

Other lines start with the message-id, the R.C.B (rack.cluster.board) location, and optionally, the time period for this message (which overrides the default).

Ensuring the apsm Message Filter is Active

The running apsm program detects whenever the `apsm_msg_filter.txt` configuration file is created, modified, or deleted, it activates, modifies, or deletes the appropriate filters and logs it in `apsm.log` file. Following is an example of that log entry:

```
Reading in '/et3mach/apsm_msg_filter.txt' and building the configmgr.alertlog
filter map


configmgr.alertlog filter file '/et3mach/apsm_msg_filter.txt' (default msg delay
12:00) last modified 2019/06/30 11:58:38 PDT


The following configmgr.alertlog filters are now active
-------------------------------------------------------
MSGID=PDZBMW0061W RCB=0.8110.0 delay=1:00
MSGID=PDZBMW0002E RCB=2.8110.0 delay=0:15
MSGID=PDZBMW0011W RCB=4.a110.0 delay=12:00
```

If you do not see similar filter activation messages in `apsm.log` file, you must check whether apsm was started with `-alertlog_filter` parameter by running the following command:

```
# ps -ef | grep apsm
root 21472   1  0 Apr26 ?    00:00:00 /bin/ksh /et3mach/bin/run_apsm
root 21480 21472  0 Apr26 ?  00:38:35 /et3mach/bin/apsm -alertlog_filter
```

If you do not see the `-alertlog_filter` parameter, inform the Cadence FSE member so he can edit `/et3mach/bin/run_apsm` file and ensure the following line is active (uncommented):

```
exeparm="-no_xcu_shutdown -alertlog_filter"
```

After this file is edited, apsm must be restarted. Run the following commands as `root`:

```
# stop cdn_apsm
# start cdn_apsm
```

After running the commands, look into the `apsm.log` file at the bottom to verify that the apsm message filter startup messages are present and match it to what you included in `/et3mach/apsm_msg_filter.txt` configuration file.

© 2021 All Rights Reserved.

Board-Manager and Power Log Files

Hover the mouse over different areas of the rack, like a board domain, target, the board <*nn*> or CCD <*nn*> text on the rack, to see the corresponding status information. If you position the mouse on text that serves as a link, the mouse-pointer changes to a an arrow, and you can click here to view the corresponding board-manager (BM) log file.

Click on *Board*, *SCD*, *CCD*, *ACU*, *WCU*, or *Power* text strings to access the log files for the board managers and power supplies. The log files open in their respective tabs. Use the *Save log to file* option to download the log file to your local machine. Previous and next date options enable you to navigate through the log files for this board. To view the logs for other boards in the system, use the *board-1* and *board+1* options.

Board manager and power logs are available in the `/et3mach/log/<mm>-<dd>-<yyyy>` dated subdirectories. Typically, logs are maintained for a minimum of two weeks. The logs contain date and time stamps, enabling you to download, combine, and sort the logs by timestamp to correlate events. Board and CCD log pages also enable you to extract and save the temperature information as a `.csv` file that you can open in a spreadsheet tool, such as Microsoft Excel.

Figure 6-27 Palladium Z1 Log Files

Physical Status LEDs - SCD, CCD, and LD

The SCD and each CCD and LD has a physical multi-color LED on the right side of its front faceplate, which can be seen when the front door of the rack is opened. The following picture shows the faceplate LED display on the emulator cabinet.

© 2021 All Rights Reserved.

Figure 6-28 Faceplate LED Display

The following table lists the LED colors, state, which could be steady or blinking, and the state indications.

Table 6-6 Apollo Cabinet LED Status

Color	State	Indications	Faceplate legend
Green	3 Hz blink	Booting up	PWR
Amber	1 Hz blink	Awaiting system link	LNK
Amber	Steady state	Not in Use	RDY
Green	Steady state	Enabled	ENB
Green/Red	1 Hz alternate	Faulty	FLT
Blue	Steady state	Service – ready to remove	SRV
Blue	1 Hz blink	Locating	LOC

Viewing Diagnostics Information

Use the *Diags* option in the *Main* menu that appears in the upper right corner of the emulator status GUI to open the *Diagnostics* tab.

The following figure displays the Diags menu.

© 2021 All Rights Reserved.

Figure 6-29 Emulator Status GUI - Diags Menu

In this tab, you can run selected diagnostics from the SCD on specific boards or domains.

Figure 6-30 Palladium Z1 Diagnostics Tab

© 2021
All Rights Reserved.

Viewing Palladium Z1 Rack—LCD Display

The rack LCD display is a USB-attached display attached to the front of each rack, and driven by the CCD. This shows the clusters in a single rack—there is one display for each rack.

The message of the day is displayed as a ticker in the upper right hand of the status bar (not shown in the following picture). Host usage is displayed. Note that the jobs table is not displayed.

The colors in the LCD display have the same meanings as those used in the emulator status GUI.

Figure 6-31 Palladium Z1 Rack—LCD Display

© 2021
All Rights Reserved.

© 2021 All Rights Reserved.

7

Improving Emulation Speed

This chapter describes the factors that influence emulation speed in Palladium Z1 emulators, and the actions you can take to improve emulation speed. It covers in-circuit emulation and emulation with a synthesizable testbench in ICE mode.

The following topics are discussed in detail in this chapter:

- Improving Oversampling
  - Upgrading from 4X Mode to 2X Mode
  - Upgrading from 2X Mode to 1X Mode
  - Upgrading from 1X Mode to 0.5 Mode
- Improving Step Count
  - Factors Contributing to Step Count
  - General Techniques to Improve Step Count
  - Critical Path Reports
  - Using Critical Path Reports to Improve Step Count
  - Viewing the Critical Path in the Schematic Viewer
  - Reports of Resource Utilization
  - Effects of Memory on Step Count
  - Using breakNet and breakPin
  - Effect of Vision Mode
  - Use of compilerEffort, optimizationEffort and placementEffort
  - Use of partitionerEffort
  - Using Design Retiming
  - Files to Send to Customer Support

© 2021 All Rights Reserved.

- Improving Step Count for Designs with External Target Interfaces
- Troubleshooting System Problems

Improving Oversampling

A Palladium Z1 emulator has its own clock called FCLK. Designs are usually compiled with CAKE clock generation. With CAKE clock generation, at compile time you specify an oversampling ratio, *N*:

```
clockOption -add {technology CAKE N}
```

The oversampling ratio is the number of FCLK cycles per cycle of the fastest design clock. The oversampling ratio is usually 4, 2, or 1, but 0.5 can be used with certain designs. A design compiled with oversampling ratio 4, 2, 1 or 0.5 is said to be compiled in 4X, 2X, 1X or 0.5X mode.

The fastest design clock frequency is *N* times slower than the FCLK frequency. Therefore, the higher the oversampling, the slower the design runs. Higher oversampling also adversely affects the effective Logic Analyzer trace depth and waveform upload speed, since higher oversampling means that fewer design clock cycles are captured in a given Logic Analyzer trace depth.

To obtain the maximum possible emulation speed, trace depth, and waveform upload speed, you should use the smallest possible oversampling. Refer to the following sub-sections for more information.

Upgrading from 4X Mode to 2X Mode

If a design functions correctly in 4X mode but fails in 2X mode, the reason is usually loop-breaking. When a design contains combinational loops, the compiler's loop-breaking algorithm automatically inserts delays to break the loops. If these delays prevent signals from propagating from one register to another within one user clock cycle, the design will function incorrectly. The design might fail in 2X mode, but function correctly in 4X mode because 4X mode increases the period of the clock compared to the loopbreaking delay.

In Loop-Breaking, Proactive Steps to Avoid Loop-Breaking Problems and Avoiding the Need for Oversampling describe the steps you can take to improve oversampling from 4X to 2X.

© 2021 All Rights Reserved.

Upgrading from 2X Mode to 1X Mode

This section describes the steps to upgrade from 2X mode to 1X mode. There are three things to consider:

- For in-circuit emulation with a target system, you might need different compilation parameters related to the target interface in 1X mode than in 2X mode.
- For some designs, loop-breaking causes functional problems in 1X mode but not in 2X mode. To make loop-breaking in 1X mode functionally almost identical to loop-breaking in 2X mode, use `precompileOption -add breakHalfCycleAll`. The `breakHalfCycleAll` option might have too much of capacity cost. If so, the sections of Loop-Breaking mentioned earlier can help you improve oversampling from 2X to 1X.
- Some designs have a capacity cost for 1X mode. However, you might be able to eliminate the capacity cost using techniques described in the *Using 1X Mode* chapter.

Upgrading from 1X Mode to 0.5 Mode

The 0.5X mode is a special mode that enables the fastest design clocks to run at twice the frequency of FCLK.

Clocks running faster than FCLK are called double-speed clocks. The emulator capacity requirement is doubled for parts of the design clocked by double-speed clocks. Thus, 0.5X mode is useful primarily for designs where the fastest design clocks are used by only a small fraction of the design. 0.5X mode imposes additional restrictions on designs. For example, storage elements cannot change state on the falling edge of a double-speed clock.

For more information about 0.5X mode, refer to the Modeling Clocks in ICE Mode chapter.

Improving Step Count

FCLK frequency is design- and compile-dependent, but for most designs, is between 0.5Mhz and 1.5Mhz. Underlying FCLK is a much faster clock: the step clock. The step clock frequency is fixed at 500Mhz.

Each FCLK cycle takes a number of steps *M*, determined by the compiler for a given compile of a given design. Thus, FCLK frequency is inversely proportional to step count:

```
                          515.625 MHz
FCLK frequency = --------------------
                        [(step count rounded up to multiple of 8) + 8]
```


© 2021 All Rights Reserved.

The step count, *M*, is affected both by long combinational paths in the design, and by the availability of hardware resources in the emulator.

You can improve step count by eliminating unnecessary long paths in the design, or by using compilation options that reduce the need for a particular hardware resource. Refer to the following sub-sections for more information.

Factors Contributing to Step Count

High step count can be caused by several factors.

- Long combinational paths in the design can cause a high step count. Paths from flip-flop or latch clock, set or reset to output, or from latch D to Q are considered combinational in this context. Paths through memory are also combinational. A long path can cause high step count even when the path is not needed for functionality.
- Scheduling of accesses to memories can require a high step count. Sometimes, the compiler is forced to pack multiple memories, or memory slices, into a single physical memory with multiple accesses per FCLK. Successive accesses to an EXTRAM memory (the memory in a User Memory Card) must be at least 20 steps (or in some cases, 28 or 34 steps) apart.
- Designs compiled in 1X mode or 0.5X mode usually need I/O timing constraints on inputs from an external target. I/O timing constraints (especially on inputs) can contribute to a high step count.
- Vision mode FV can increase the step count compared to DYNP.
- The fraction of the emulator's logic primitives (also called instructions) that are used by a design imposes a lower limit to the step count, and therefore, an upper limit to the FCLK frequency. The step count can be no better than `512 steps x P`, where `P` is the fraction of the logic primitives that are used. For example, if the fraction of primitives used is 50%, the step count must be at least 256 (that is, 50% of 512). However, most designs have a higher step count than the minimum imposed by utilization.
- A shortage of routing resources within the emulator can contribute to high step count.

General Techniques to Improve Step Count

When you compile a design repeatedly, the back-end compiler (et3compile, invoked by the `compile` XEL command) automatically uses a different *random seed* for each compile, unless you specify a specific seed. Usually, different compiles will show a variation in step count due to the different random seeds used. You can automatically try multiple compiles with different seeds using the `compileFind` command with different options (such as

© 2021 All Rights Reserved.

`-first`, `-best`, `-opt`, `-all`, and `-lsf`). For a complete list of options, refer to the description of `compileFind` command in *Compile-time Commands in ICE Mode* chapter of the *VXE Command Reference Manual*.

When studying the effects of your actions on step count, do enough trials to avoid being misled by random variations. For example, in one design it was found that most trials had FCLK frequency in the range of 500-600 kHz, but about one in 10 trials had the frequency 969 kHz. For this design, even 10 trials was not enough to ascertain the effect of an action. Other designs might have a frequency variation of only 10% over many trials, and just a few trials would be sufficient.

When `compileFind -best` option is used, if the compiler determines that the step count for a given trial will be inferior to the best trial so far; it aborts the trial before completion. When studying the effects of your actions on step count, use the `-all` option so that every trial runs to completion. With the `-all` option, you can see the distribution of speeds in a set of trials.

Use the `compileFind -lsf` option to quickly run a large number of trials. With eight workstations, each having sufficient CPUs and memory to run three trials in parallel, you can run 24 trials in the same time that it takes to run a single trial. Use `compileFind -lsf` both for experiments, and in production to get a good step count with minimum compile time.

For best results, always use the latest software release.

Remove Modules to Reduce Design Size

Reducing the size of the design can improve emulation speed. Suppose your design contains just two large modules: A and B. If most tests require only one of the two modules, do three compiles: A-only, B-only, and the full design. If the compiles of A-only and B-only have better speed than that of the full design, you can reduce the overall testing time by running each test on the minimum design required for that test. Use the `moduleStub` and `instanceStub` commands to stub out modules or instances.

Tie Test-mode Signals

If all your testing will be done with a signal such as test-mode (or scan-mode, or bist-mode) set to zero, tie the test-mode signal to zero at compile time using the `tieNet` command. The compiler will then optimize the design logic based on the knowledge that test-mode is always low. Such optimization can greatly improve emulation speed.

© 2021 All Rights Reserved.

Identify and Tie Dead Clocks

With `precompileOption -add instrumentEdges`, you can run a set of tests, and obtain a list of ***dead clocks*** that never toggled during the tests. If you tie such clocks high or low, the compiler will optimize the design logic based on the known values of these clocks, and this might improve emulation speed. Tying dead clocks must be done with caution, since a clock that is dead in one test might not be dead in another test.

Remodel ASIC Library Cells

If your design instantiates ASIC library cells, examine the flip-flops and latch cell models carefully to make sure the models are efficient. For example, a flip-flop cell should be modeled using a Palladium (qtref) flipflop, not using a master-slave pair of latches. If some inputs of the cell (for example, test inputs) are not needed for tests that will be run in emulation, remove any logic enabled by these inputs inside the cell.

Adjust the emulator configuration

If a design fully uses the emulation resources available, usually the emulation speed can be improved by increasing the number of domains or boards. If the design uses nearly all of the memory resources of the emulator, you might improve emulation speed by adding User Memory Cards.

When you increase the number of domains or boards from the minimum possible number for the design, usually emulation speed increases at first. However, if you increase the number of domains too much, eventually the speed decreases because more signals need to travel from one domain or board to another. Therefore, when emulator utilization is low, you might improve emulation speed by *reducing* the number of domains or boards.

Do many compile trials

For some designs, when multiple trials are done, most trials have a slow speed, but a small percentage of trials have good speed. If you have such a design, do a sufficient number of trials to get good speed. To avoid long compile time, use `compileFind -lsf` to do multiple trials in parallel. Each workstation can do as many trials in parallel as its CPUs and RAM allow. In the `LSF_options` file, use the following:

```
-R "rusage[mem=N] select[OSNAME==Linux && mem>N]" [other_options]
```

The `rusage` command tells LSF the expected process size of jobs, so that it does not assign too many jobs to one workstation. For a 4-board design, `N=256000` should be sufficient and `N=150000` might be sufficient.

© 2021 All Rights Reserved.

Often, to improve step count significantly, you must first analyze the critical path report as described in the following sections, to find the reasons for a poor step count.

Critical Path Reports

After compiling, the et3compile message files (`tmp/e*.msg` and `tmp/criticalPathReport.msg`) include three different types of critical path report.

- Pre-schedule paths are paths with the greatest number of logic levels.
- Post-schedule by-levels paths are paths with the greatest number of weighted logic levels. The weight of each level considers how the design is partitioned (for example into emulator boards and chips) and also considers terminalTiming constraints specified by the user, and memory delay.
- Post-schedule by-steps paths are not longest paths. Rather, in the first path, the first net printed is the net in the design scheduled at the highest step number. This net is driven by some gate. The second net in the path is the input to that gate that is scheduled at the highest step number. And so on.

The reason for printing three types of path is historical.

- Originally, only pre-schedule paths were printed. When a design has very long paths, these long paths usually cause high step count, and the pre-schedule path report shows the user the long paths.
- Sometimes, the path with the most logic levels is partitioned into few chips and does not cause high step count. Another path with fewer logic levels, has more hops between chips, or needs resources (such as memory) that are scarce, and this fewer-level path causes high step count. Unfortunately, the pre-schedule path does not show this path because it has fewer levels. The post-schedule by-steps report was developed to show such paths.
- Unfortunately, sometimes the first by-steps path shows a short, easy path whose furthest-downstream net only has a high step number because the scheduler postponed scheduling this easy path to the end. The user would like the report to show more difficult paths which are the real reason for the high step count. The by-levels report was developed to show such more-difficult paths. It is hoped that the by-levels report will be useful, but this report is new in VXE and its usefulness is not well-established.

In a design where the first by-steps path is short (that is, it has few logic levels), there are two ways to see longer paths which more likely contribute to high step count:

- Look at the by-levels report.

  Or,

© 2021 All Rights Reserved.

- Use the `compileFind -crit_paths` option to print many paths, and look for by-steps paths with many logic levels.

A drawback of the by-steps reports is that you might need to print many paths, and it might be unclear which paths are important. A drawback of the by-levels reports is that the paths are calculated based on level number, and level number does not consider increases (jumps) in step count caused by lack of emulator resources.

To fully understand how critical paths are calculated, it is helpful to know how the Palladium compiler works. It works as follows:

Precompile, and the first stage of et3compile (called pass 1) optimize the logic of the design, and implement the design using ET primitives, which are combinational gates (up to 4 inputs), 1-FCLK delay cells, and memory port primitives. Each ET primitive is eventually implemented as a processor instruction, and et3compile's reports use the terms primitive, box, and instruction interchangeably. The pre-schedule critical path report is printed at the end of pass 1, and each logic level is an ET primitive.

The second stage of et3compile is called pass 2. The UPART module partitions the design into boards, chips, and clusters of (usually) 4 processors. Then, the SCHED module does the following:

1. It levelizes the design. Levelization assigns a "level number" to the output of each primitive. A simplified description of levelization is: a gate that drives an "ending net" (a net that has no logic depending on it) is at level 0. Primitives that drive the inputs of that gate are at level 1. And so on. However, the increase in level number at each level also considers net delay, memory delay and IO timing delay, and the level number for a given primitive is at least 1 higher than the maximum of the level numbers of the primitives its output net fans out to. At the end of levelization, the level number of each primitive is approximately the number of steps needed to compute the longest path downstream from that primitive, if there were no competition for emulator resources by other paths.

   The level calculation assumes the following numbers of steps net delay based on the type of fanout:

   ```
   within a processor cluster: 0
   different cluster within same chip: 1-3
   different chip within same chip-pair: 4-6
   different chip-pair within same domain: 10
   different domain: 25
   ```

2. After levelization, the scheduler schedules each primitive in the design, one by one, in decreasing order of level number. Scheduling a primitive means to decide which processor will implement the primitive, and at which step. The partitioner already assigned each primitive to a group of 4 processors, but if the scheduler sees that a given primitive could be computed at an earlier step by moving it to a different group of processors, it might do that. Often, the extra steps needed to transfer a value between

© 2021 All Rights Reserved.

groups of processors, and the extra routing resources that might be needed, make it unbeneficial to move a primitive.

To print pre-schedule paths, level numbers are assigned in a different way than described above. Each net's ET6CRIT level number is the number of logic levels in the longest path coming from upstream, leading to this net. To print N paths, the nets of the design are sorted by ET6CRIT level number, and the highest-level N nets are taken as starting points. One path is printed for each of these nets. It is the longest path coming from upstream, leading to this net. The path is printed starting from the downstream end.

By-levels paths are based on the level numbers assigned by SCHED levelization. The starting net for the first path is the net in the design with the highest level number. (By level number of a net, we mean the level number of the primitive that drives the net). A path downstream from this net is printed. To generate the path, if a net has multiple fanouts, the fanout with highest level number is taken. The path is generated starting from the upstream end, but printed starting from the downstream end.

To print N paths, the starting nets are chosen as follows. If 10 nets have level number 500, and 10 have level number 499, the starting net for the first path has level number 500, and the remaining 9 nets with level number 500 are skipped. The starting net for the second path has level number 499, and the remaining 9 nets with level number 499 are skipped.

By-steps paths are based on the step number at which each net is scheduled. (By step number for a net, we mean the step number for the primitive that drives the net.) The starting net for the first path is the net in the design scheduled at the highest step. This net is driven by some gate. The second net is the input of that gate that has been scheduled at the highest step. And so on. To print N paths, the starting nets are chosen as follows. If 10 nets are scheduled at step 500, and 10 are scheduled at step 499, a net scheduled at step 500 is the starting net for the first path, and the remaining 9 nets scheduled at step 500 are skipped. A net scheduled at step 499 is the starting net for the second path, and the remaining 9 nets scheduled at 499 are skipped.

Format of the Critical Path Reports

When et3compile runs, it generates critical path reports in the tmp/e*msg files. These critical path reports show encrypted signal names (such as _AAAA[3234]) rather than actual signal names from the design. To see actual signal names, compile the design using the `compileFind` command with the `-names_crit` option:

```
compileFind -best -max_trials <N> -names_crit -crit_paths <N> [other_options]
```

The `-best` option instructs the compiler to abandon any trial that results in slower performance compared to a base compile trial. The highest performance trial remains at the end of all trials.

© 2021 All Rights Reserved.

The `-max_trials <N>` option (where *<N>* is a number) defines the number of random trials the compiler tries as part of the process to find the best timing results. Each trial uses a unique random seed, and generates slightly different timing results.

If the `-names_crit` option is used, then when all trials have finished, the compiler calls the `getCriticalPathDesignNetNames` command to generate the file `tmp/criticalPathReport.msg`, containing critical path reports showing actual signal names.

Note: If you compile without `-names_crit`, you can manually run the XEL command `getCriticalPathDesignNetNames <filelist>` where *<filelist>* is the name of a file that contains the names of the `tmp/e*.msg` files that should be processed to generate `tmp/criticalPathReport.msg`. Without the *<filelist>* argument, `getCriticalPathDesignNetNames` processes only the `tmp/et3compile.msg` file, and the resulting `criticalPathReport.msg` file might be incomplete.

If the `-crit_paths` *<N>* option is used (where *<N>* is a number), et3compile generates *<N>* preschedule critical paths, and for each trial, it generates *<N>* post-schedule paths of each type (by-levels and by-steps). Omitting `-crit_paths` *<N>* is equivalent to specifying `-crit_paths 1`.

At times, you might see net names in the `criticalPathReport.msg` file in the following form:

```
trNet_<number>
```

Such names, typically, appear for those nets added to the design by the compiler. In some cases, you can get more meaningful net names instead of `trNet` names by setting the `SMDB_SAVE_MC` environment variable to `1` before precompile.

To be able to see readable net names similar to the original net names in the `criticalPathReport.msg` file, set the `SMDB_SAVE_MC` environment variable to 1 before precompile.

The `SDMB_SAVE_MC` setting is not done by default because it can greatly increase the size of the database size and compilation time. Set the `SMDB_SAVE_MC` environment variable only if it is needed.

In a Palladium Z1 compile with the default (high) effort, some nets in the critical path report have no names. Names are absent when logic optimization rearranges combinational logic to create nets that are not equivalent to specific user nets. If you want to see names for all nets, compile with low effort (using `compilerOption -add {compilerEffort LOW}`) and assume that the resulting critical path is similar to the path with default effort.

Here is an example of a pre-schedule critical path. To save space, only the beginning and end of the path are shown.

© 2021 All Rights Reserved.

CRITICAL PATH:

```
311(          2): ETF0C0->core.y_int[63] **
310(         75): ETBB8B->core.n15975 **
309(       129): ET607F->
308(       214): ET030F->core.U15336.NET00
307(       133): ETFDF0->core.n15971 **
...
3(6947187): ET028A->core.r_index[0] **
2(2960589): ET000F->QT_smNet15610357
1(7994848): FD01->core.n18_12193925@QTdelay
```

The path is printed starting at the downstream end, with one line for each net in the path. Each line is of the form:

```
level(count): function->netname
```

where

- level is the ET6CRIT level number for this net.
- count is the number of nets in the design having this ET6CRIT level.
- function is the letters "ET" followed by the 16-bit truth table
- for the combinational gate driving this net, or FD01 if the net is driven by a delay cell.
- netname is the name of this net. If the name is from the user design, the symbol ** is printed at the end of the line.

Here is an example of a post-schedule by-levels critical path. To save space, only the beginning and end of the path are shown. A by-steps critical path has the same format, except that the Level column is omitted.

```
Critical path  1     sinks
Count  Location    Step Jump Level Type  ID ins:cr Name
----------------------------------------------------------------
. [002,075,013,1] 451 +  0   NET 59794 1 core.n307 **
1 [002,075,013,1] 451 +  01  LOGIC 80088 4:3
. [002,075,013,0] 451 +  0   NET 29487 1 core.n305 **
2 [002,075,013,0] 451 +  02  LOGIC 79944 3:1
. [002,075,013,0] 450 +  1   NET 29388 3 core.cken **
3 [001,079,048,0] 419 + 31   28  LOGIC 79821 4:3
...
209 [003,068,025,3]  37 + 32  539  LOGIC 11258 3:3
. [003,068,025,0]  36 +  1   NET 3579658 QT_smNet6629
210 [001,079,048,3]  20 + 16  565  LOGIC 59676 2:1
. [001,079,048,0]  19 +  1   NET 35800 3 core.n18 **
211 [003,069,025,2]   0 + 19  591  FD01  11284 4:0
```

The report contains two types of lines called *box* and *net* lines. A *box* is a logic primitive, such as a combinational gate, memory port primitive, or delay cell. A *net* connects the output of one logic primitive to inputs of other primitives. The top of the report corresponds to the *downstream* end of a combinational path.

© 2021 All Rights Reserved.

Box and net lines have the following fields:

- *Count* is the number of logic levels from the downstream end of the path to the current point.
- *Location* is the location of a given logic primitive in the emulator. A location consists of four decimal numbers: board number, chip number within a board, cluster number within a chip, and processor number within a cluster. (Each domain has 12 chips. In the most common compile mode, each chip has 64 clusters and each cluster has 4 processors.)
- *Step* is the step count used for the path, from the upstream end to the current point.
- *+ Jump* is the change in step count from the current line to the previous line.
- `Level` is the SCHED level number for this box.
- *Type* is either NET, or a box type. The box types LOGIC and LOGICx are combinational primitives. XMPRx, XMPRi, XMPWx, XMPRi are memory port primitives; the lower-case `x` and `i` stand for *external* (EXTRAM) and *embedded* (INTRAM) memory, respectively. `FD01` is a 1-FCLK delay primitive that is a part of the implementation of a design flip-flop.
- *net* lines usually end with the name of a net. If the name is from the design, it is marked with **.

At the end of the path is a summary, including the maximum jump in step count for any single pair of lines. Large jumps in step count usually occur when a resource is heavily utilized, so that the scheduler needs to wait some steps for the resource to become available.

Using Critical Path Reports to Improve Step Count

The ET6LOADA section of `tmp/et3compile.msg or tmp/et3pass2.*.msg` reports the average and peak boxes per emulation processor (ep).

Consider the following example:

```
00 ET6LOADA | Type              Peak           Avg/Peak
...
00 ET6LOADA | Boxes per ep:     ep[1,41,56]    Boxes=360/439
```

Here, the average boxes per ep is 360 and the peak is 439. The average boxes per ep gives the most accurate report of the fraction of ET primitives used.

```
                                        average boxes per ep
    Fraction of ET primitives used = ------------------------
                                                512
```

The peak boxes per ep is usually not important because after this report is printed, the scheduler can move boxes between processors to balance the load. However, the average boxes per ep is important:

© 2021 All Rights Reserved.

The design step count cannot be better (that is, cannot be lower) than the average boxes per ep. If the step count is no higher than 1.3 times the average boxes per ep, it means the step count is close to the best possible at the given percent utilization of ET primitives. In such a case, the only way to get a large improvement in step count is to reduce the size of the design (e.g. by removing modules, tying test-mode signals, improving asic library cell models, or reducing 1x mode shadow nets), or increase the number of domains or boards used in the compile.

If the step count is higher than 1.3 times the average boxes per ep, then it is likely that either long logic paths in the design contribute to the high step count, or lack of an emulation resource other than ET primitives (for example, memory or memory bandwidth) contributes to the high step count. In such cases, the critical path reports are the best tool to find the reason for high step count.

If the three types of critical path reports show different paths, it is expected that the following reports are most significant:

- ❑ Post-schedule by-levels paths
- ❑ Post-schedule by-steps paths containing many logic levels, or post-schedule by-steps paths containing large jumps in step count at the locations of specific resources as explained below.

When a critical path contains large jumps in step count (jumps of more than 25 steps for a single logic level), use the following check-list to understand the jumps and, in some cases, improve emulation speed:

- If large jumps occur when the path goes from one board to another, then adding more board-to-board cables to the emulator can improve the speed. Unfortunately, if the emulator has 4 or more boards, usually all cable connectors are already used, and it is not possible to add more cables.
- If large jumps occur at memory ports (shown in the report as boxes of type XMPW or XMPR rather than LOGIC) see Effects of Memory on Step Count section.
- In the report, boxes of type DELAY are delayBoxes specified by `delayBox` commands in the compile script. If DELAY boxes are found in the critical path, you can likely improve emulation speed by reducing the size (amount of delay) in the corresponding delayBox commands. Reducing delayBox delays must be done with caution, since IO timing constraints might be needed for functionality.
- Large jumps in step count that occur at points in the path other than those listed above, usually have no significance. Large jumps are especially likely when emulator utilization is high, or at points in a path where the step number is 512 or higher. If large jumps occur in by-steps paths, check the previous points in the check-list. If those points do not apply, but step count exceeds 1.3 times the average boxes per ep, use the by-levels paths

© 2021 All Rights Reserved.

rather than by-steps paths, or use the `-crit_paths <N>` option to generate a large number of by-steps paths, and look for by-steps paths with many logic levels. Also, use the methods in the General Techniques to Improve Step Count section.

When a critical path contains many logic levels, use the following checklist to improve emulation speed:

- Does the design logic for part of the long path have a regular, repeating structure?

  If so, it is likely that you can eliminate this part of the long path. For example, in one design, a memory model written for emulation contained an 1024-to-10 encoder written using a basic “for” loop, which produced a chain of 1024 muxes. When the “for” loop was replaced by a tree-structured encoder, the long path was eliminated and step count improved. In another design, a long path was part of a testability circuit which did not need to be tested in emulation. The path contained a certain cell repeated over and over. When this cell was modified to eliminate the long path, the step count improved.

  To find a repeating structure in the critical path, it is usually helpful to view the post-schedule path in the schematic viewer, using the method described in the next section.

- What RTL signal is generated by this path?

  A critical path might contain only net names like `n1234` generated by synthesis that are not a part of the RTL. Usually, the most downstream net connects to the D input of a flip-flop, and the Q output of the flip-flop generates an RTL signal. Use the DUT design browser, the `verigen` program, or a text editor, to inspect the gate-level view of the design and find the flip-flop at the end of the path. This technique can help you identify an RTL construct whose implementation results in a long path.

- Is most of the long path in a single module of the design?

  Using a synthesis tool other than xeCompile, you can re-synthesize the specific module, optimizing for speed rather than area. Some synthesis tools have an option for *retiming* (balance_registers) which can redistribute flip-flops into combinational paths to reduce the path length.

  If the module is a Synopsys DesignWare component, consider using DesignWare models implemented specially for Palladium Z1 (available from Cadence). Alternatively, try using Synopsys Design Compiler to synthesize it. A library is available that enables Design Compiler to target the Palladium Z1 `qtref` library.

  You can get an upper-bound estimate of the possible improvement by changing a particular module or instance. To do so, use the `moduleStub` or `instanceStub` command to replace the module or instance by an empty module or instance. Use the precompileOptions `keepStubInputs` and `keepStubOutputs` to prevent extra optimization due to undriven module outputs or loadless module inputs.

© 2021 All Rights Reserved.

- Does the long path include paths from D to Q of latches?

  To find latches in the critical path, it is usually helpful to view the post-schedule path in the schematic viewer, using the method described in the next section.

  If the latches are in master/slave pairs, replace the master/slave pairs, if possible, with flip-flops from the `qtref` library.

  If the latches use a non-overlapping clock scheme, provide information about the non-overlapping clocks using the `noOverlapClock` command.

  You can use the `precompile -syncData` option to insert a delay on the D input of every latch in the design. The inserted delay will eliminate paths going from D to Q of latches. However, in most designs, using `-syncData` increases the CAKE oversampling number required for the design, so that `-syncData` has no overall benefit and is not recommended.

- Does the long path include paths from flip-flop or latch set or reset to Q?

  You can eliminate paths from flip-flop or latch set or reset to Q by using the `precompile -syncReset` option. In some designs, using `-syncReset` increases the CAKE oversampling ratio required for the design, so that the option has no overall benefit. However, in some designs, especially with CAKE 2 or higher, the option improves step count without requiring extra oversampling.

- Is the starting point for the path a primary input?

  If a one-FCLK delay on this primary input will not cause a functional problem, you might be able to slightly improve the step count by using `breakNet` to insert a break at this primary input. If the slight improvement enables the compiler to stay below 512 steps, a larger improvement might occur.

  If a one-FCLK delay will cause a problem, you should not use a `breakNet`, but if there is a `terminalTiming` constraint on the given input, consider reducing the size of the constraint.

- Does the path include one or more I/O pads in the middle of the path?

  An emulated design sometimes contains a “top-level wrapper” that instantiates a “design under test” (DUT) and other logic or memories. The critical path might go through several bidirectional I/O pads of the DUT block. Such paths are often not needed for functionality, and can be broken using breakNets without harming functionality.

  If the I/O signals of the DUT are made I/O of the top-level wrapper, and these I/O are not connected to a target, breaks will be added automatically unless you specify the `delayBidir off` compilerOption. If the I/O signals are connected to a target, then unless you use `terminalTiming` to constrain input components to be scheduled later

© 2021 All Rights Reserved.

than outputs, the paths from output to input of an I/O pad will in effect be broken automatically.

- Does the path include a design clock with many levels of logic upstream from the clock? In a silicon chip, the path taken by a clock transition from a clock source such as a PLL to a flipflop clock pin is usually short -- otherwise, skew between clocks is difficult to control. If the critical path report shows an internal clock net with many logic levels upstream, usually this long path is not a true clock path. You might be able to eliminate the long path by tying a test-mode signal, or break the path using a breakNet, or you might be able to simpify the generation of this clock in the design, or make the clock a CAKE clock source. Any of these techniques might improve the emulation speed.

  If a certain clock clocks a large part of the design, making that clock a CAKE clock source can sometimes improve emulation speed even if there are only a few logic levels upstream from the clock.

  Changes in clock generation must be done with caution because they might change design functionality.

- Does the path include nets with the string `QTshadow` in the name?

  Nets with `QTshadow` in the name are shadow nets. Shadow nets occur in 1X-mode compiles, and in compiles that use the `precompile -breakHalfCycle` option. Consider whether the shadow nets appearing in the critical path can be eliminated, using techniques described in 1X Mode Capacity Cost on page 422.

  If you use the multisampled I/O feature, multisampled inputs and outputs can appear in the critical path with `QTshadow` as part of the name. If the given input or output does not need to be multisampled, make it non-multisampled by not specifying it as a multiSampledTerminal, or by using enableMultiSampledIO 2 instead of 1.

Viewing the Critical Path in the Schematic Viewer

The reports in `criticalPathReport.msg` show net names, but do not show the type of gate driving each net. Using a schematic viewer can be helpful in finding a repeating structure within the critical path, or in identifying latches within the critical path. If the Verdi schematic viewer is used, a repeating structure usually shows up as a very noticeable “diagonal line” in the path displayed in the schematic viewer.

To view the critical path in the schematic viewer, perform the following steps:

- Compile the design, using either `compileFind -names_crit [other_options]` or `getCriticalPathDesignNetNames` to generate the file `tmp/criticalPathReport.msg`.

© 2021 All Rights Reserved.

- In the design directory, with your Unix path set to include VXE software, run the `crit.sh` script (<*VXE_INSTALL_HOME*>`/share/vxe/gift/misc/crit.sh`), with no arguments. The script reads the first post-schedule path in `tmp/criticalPathReport.msg`, and generates a Verilog file named `crit.v`, which is a gate-level netlist of the critical path. If there is no post-schedule path in `tmp/criticalPathReport.msg`, the script instead reads the first pre-schedule path.
- The script will run for a long time (for example, it took 45 minutes for a 95M gate design where the postschedule path had 3000 levels). If the run time for the script is intolerably long, edit the `tmp/criticalPathReport.msg` to remove part of the path.
- The script uses the `verigen` program with the `-path` option to fill in the gates and nets between the nets that are shown in the critical path report. If the critical path report contains sequences of more than one or two nets with no names, the critical path in `crit.v` might have gaps. You can avoid having nameless nets by compiling with `compilerOption -add {optimizationEffort low}`, but the resulting critical path might be different from the path with default `optimizationEffort`.
- Invoke the *SimVision Schematic Viewer* by executing the following command:

  ```
  xrun crit.v -access rwc -gui &
  ```

  In the *Design Browser*, highlight the top module. Then, click the *Schematic Viewer* icon. When the Viewer comes up, double-click the top module to view its contents, which will be the critical path.
- If the schematic view of the critical path is confusing, correlate it carefully with the report in `criticalPathReport.msg` file.

Reports of Resource Utilization

In the ICE flow, an RTL design is usually synthesized using `vaelab` or `hdlSynthesize`. The synthesis program generates a report in `vaelab.log` or `hdlIce.log` like the following:

```
=========================================================
module        Area/each   Count     Total_area
=========================================================
hostif          3305.00     1         3305.00
framer         24684.00     1         24684.00
sequencer      331212.00    2         662424.00
...
---------------------------------------------------------
Total Area (250 modules & 535 instances) : 235533277.00
```

"Area" means 2-input gate count and does not include hard-macro memories.

"Area/each" gives the gate count for a given module, not including its submodules,

"Count" gives the number of instances of the given module.

© 2021 All Rights Reserved.

"Total_area" gives the product of the two previous columns.

"Total Area" in the last line is the total 2-input gate count for the design.

In the ixcom flow, a similar report is printed in the `ixcom.log` file.

The precompile command prints messages like the following in xe.msg:

> Size of design in 2-input gates originally: 232202013
>
> Size of design in 2-input gates after remodeling and cleaning: 217538052
>
> Size of design in 2-input gates after transformations: 207565883

The first line (containing the word "originally") shows the "precompile gate count". It is the 2-input gate (not including hard-macro memories) for the design before any optimization by precompile. Usually the precompile gate count agrees closely with the gate count reported by synthesis. (The gate counts differ slightly because the two programs assign different gate counts to certain gates, latches and flipflops.)

The marketing specification for Palladium Z1 is that for most designs, each domain can implement 4M (or 4.5M) gates. The et3compile program prints a "design utilization" message in xe.msg:

```
Design utilization: 87.5%
```

Design utilization is the precompile gate count, divided by the marketing specification for the size and type of emulator being compiled for.

In `tmp/et3compile.msg` or `tmp/et3pass1.msg`, et3compile prints an "Estimated instruction usage" message like the following:

```
00 ET6NLOPT | Estimated instruction usage    74.27%
```

Estimated instruction usage is the percent of the emulator's ET primitives used before et3compile transforms shallow memories to flipflops and latches. After the shallow memory transformation, et3compile prints a "New instruction usage" message:

```
00 ET6MPART | New instruction usage: 71.2
```

At the end of the memory transformation process, et3compile prints a memory capacity summary similar to the following:

```
00 ET6MPART | ========New memory statistics after "memTransform"========
00 ET6MPART | INTRAM needed= 12305 (if no EXTRAM is available)
00 ET6MPART | INTRAM available = 1536
00 ET6MPART | EXTRAM needed=  32
00 ET6MPART | EXTRAM available = 128
```

In the example here, the design memories are much too large to fit in INTRAM, but easily fit in INTRAM+EXTRAM, using only 32 of 128 available EXTRAMs.

© 2021 All Rights Reserved.

Later, et3compile prints in tmp/et3compile.msg or tmp/et3pass2.*.msg, an “Average boxes per ep” message, which, as described earlier, is the most accurate measure of ET primitive utilization.

```
00 ET6LOADA | Type          Peak       Avg/Peak
...
00 ET6LOADA | Boxes per ep: ep[1,41,56]Boxes=360/439
```

In this example, the percent of ET primitives used is 360/512, or 70.3 percent, and the best (lowest) possible step count is 360.

The utilization reported by LOADA differs slightly from the earlier reports because the earlier reports do not accurately account for the ET primitives needed for memory ports. As described earlier, the ET primitive utilization imposes a lower bound on step count, and an upper bound on emulation speed. If this limit is unacceptable, reduce the size of the design, or increase the number of domains or boards.

The LOADA section also reports the minimum, average, and maximum number of accesses (per FCLK cycle) for any INTRAM and any EXTRAM in the emulator:

```
00 ET6LOADA | Total INTRAM Accesses: 27021
00 ET6LOADA | Accesses per INTRAM:   0 / 17.6 / 57 (min/avg/max)
00 ET6LOADA |
00 ET6LOADA | Total EXTRAM Accesses: 216
00 ET6LOADA | Accesses per EXTRAM:   0 / 1.7 / 6 (min/avg/max)
```

The design step count cannot be better (that is, cannot be less) than 4 steps times the maximum number of INTRAM accesses per FCLK, nor better than approximately 30 steps times the maximum number of EXTRAM accesses. If these limits are unacceptable, see the Effects of Memory on Step Count section.

Finally, et3compile reports a minimum step count due to probes:

```
00 SBSCHED  | Min step count due to probes is 432
```

If this limit is unacceptable, use visionMode DYNP instead of FV, or use moduleFV, or increase the number of domains or boards.

Effects of Memory on Step Count

The synthesis step of compilation can implement a multi-dimensional register array using "hard-macro memories" (indicated by MPR and MPW primitives in the synthesized netlist) or using flip-flops and latches. Later, the `et3compile` step can convert hard-macro memories to flip-flops and latches.

Hard-macro memories (except those converted to flip-flops and latches) are implemented using Palladium Z1 INTRAM and EXTRAM memory resources. Each domain has `48 MBytes` of INTRAM. If the domain has a User Memory Card, the User Memory Card provides

© 2021 All Rights Reserved.

8 GByte of EXTRAM. Multiple user memories can be packed into a single INTRAM or EXTRAM memory, which must then be accessed multiple times per FCLK. INTRAM accesses take fewer steps than EXTRAM accesses. Therefore, the compiler uses INTRAM whenever possible, and uses EXTRAM only for deep memories, or when the EDRAMs are already used up by shallower memories.

The compiler obeys the following rules, which are required for INTRAM and EXTRAM accesses. EXTRAM is an order of magnitude slower than INTRAM. For EXTRAM, when successive accesses are for the same user memory, and in some other cases, the accesses must be at least 28 or 34 steps apart. The memory latency (delay from memory input to output, not including routing delay for the signals) is 11 steps for INTRAM and 51 steps for EXTRAM.

The MPART module in et3compile might convert hard macro memories to latches and flip-flops, and perform other transformations. For more information, see the Transforming Memories section. The ET6MPART section of tmp/et3compile.msg or tmp/et3pass1.msg reports any tranformations that were performed. The MPART module generates the files dbFiles/<top>_et5mpart and dbFiles/<*top*>_et5mpart_transformed. <*top*>_et5mpart lists the hard macro memories in the design given to et3compile. <*top*>_et5mpart_transformed lists the hard macro memories after transformation. Both files have the same format, shown in the following example:

```
Addr  Data  Port  Name
-----------------------------------
   3     8     2  uPlatform.RFifo
   4    38     3  uPlatform.TestRows
   6    38     3  uPlatform.Fusemem
```

The line for each memory shows the number of address bits, the number of data bits, and the number of ports for that memory.

To find out whether a particular memory was implemented in INTRAM or EXTRAM, you can use the dbFiles/<*top*>.memschedule file. The file contains one or more lines for each memory (one line for each port, or if the memory is sliced into pieces, one line for each port of each piece). In each line, the RAM column has the string i00 to i48 if the memory is in INTRAM or x00 if the memory is in EXTRAM. You can force a memory to be in INTRAM or EXTRAM by creating a file in the compile directory named <*top*>.mempreset before issuing the compile command. The file can contain the line

```
EDRAM_MAX <n>
```

Here, <*n*> is an integer such as 24. Any memory with more than <*n*> address bits will be forced to EXTRAM. The file can also contain lines of the form <*target*> <*memory*> where <*target*> is INTRAM or EXTRAM, and <*memory*> is the name of a memory as it appears in the <*top*>_et5mpart_transformed file. You should never need to force a memory into INTRAM. When possible, the compiler slices large memories into many pieces that can fit into

© 2021 All Rights Reserved.

INTRAM. Usually, implementing memories in INTRAM gives better speed, but occasionally, routing congestion caused by slicing causes compile failure or poor emulation speed. In such a case, you can improve compile success rate or emulation speed by forcing deep memories (for example those with more than 24 address bits) into EXTRAM.

The post-schedule critical path report sometimes includes memory port primitives, shown as boxes of type XMPRe or XMPWe (for INTRAM) or XMPRx or XMPWx (for EXTRAM), as in the following example:

```
Count Location Step Jump Type  ID    ins:cr Name
---------------------------------------------------------------
...
. [00,022,025,0] 338 + 0  NET 527963     1 mem.dout
10 [00,022,000,0] 266 + 72 XMPRx 502888 29:29 ET6_V1_H1.mem.U3
. [00,022,000,0] 232 + 34 NET 185858     1 mem.n170
11 [00,022,000,0] 232 + 0  XMPWx 503094 66:43 ET6_V1_H1.mem.rf
...
```

If memory port primitives appear in the critical path report, it is possible that the scheduling of memory accesses is contributing to the step count for the design; especially if ports of a particular memory consistently appear in the report in multiple trials, and this memory is implemented in EXTRAM.

When memory accesses contribute to a high step count, you can sometimes improve the step count by one of the following actions:

- Turn off *write-through behavior* for the given memory, or for all memories.

  By write-through behavior, we mean that if an address *A* is both written and read in the same FCLK cycle, the value read will be the newly written value. xeCompile normally synthesizes memories to have write-through behavior. The `memWritethru` userdata command and the `memoryWritethru` compiler option can be used to turn off write-through behavior for a given memory, or for all memories in the design. With write-through off, a value written to an address *A* at same FCLK cycle does not appear on the memory data-out until the next FCLK cycle.

- Provide `shadowsData` information, if appropriate.

  If nets before or after an XMPR or XMPW port in the critical path report have QTshadow in the name, the port is a shadow port whose purpose is to allow more than one memory access per cycle of the fastest design clock. If the memory is never accessed more than once per cycle of the fastest design clock, consider using supplying `shadowsData` information as described in 1X Mode Capacity Cost on page 422.

- Re-model the memory

  Carelessly-written RTL can cause xeCompile to synthesize a memory with unexpectedly many ports. If the `tmp/<top>_et5mpart` file shows an unexpectedly large number of

© 2021 All Rights Reserved.

ports for a memory that appears in the critical path, review the RTL to see if the large number of ports is really needed for the memory function.

- Specify the memory as a sparsely-used memory, if appropriate.

  A synthesizable testbench can contain a very deep memory used in a “sparse” fashion: a given test writes only a small fraction of the locations in the memory. To reduce the EXTRAM requirement for such a memory, Palladium Z1 provides the `memoryTransform` userdata command. The `memoryTransform -add {inst SPARSE n p}` command specifies that memory *inst* should be implemented using a virtual memory scheme, with a set of 2^n pages, each of depth 2^p. At run time, if the design writes more than the available number of pages, an error message is generated (and if you are using STB mode, the emulation stops). The run-time `memory -sparse_mem_info` command gives status information for sparse memories at run time, and the command `memory -reset` *inst* clears the specified memory, freeing all its pages.

- Adjust the threshold for shallow memory transformation.

  In an ICE mode compile, by default, `et3compile` transforms the shallowest memories of the design into flip-flops and latches until such transformation would increase the ET primitive utilization above 80%. Depending on whether there is a lack of INTRAM bandwidth or logic resources, a higher or lower threshold, respectively, might give better emulation speed. You can adjust the threshold for shallow memory transformation using the `shallowMemoryMaxUtilization` option of the `memoryTransform` command. You can automatically try different values of the threshold using the `compileFind -mem` command.

  In an `ixcom` compile, the shallow memory transformation is disabled, by default, to improve the speed at which the testbench in the simulator can access memories in the DUT.

- Specify memory compaction, if appropriate.

  The memory compaction step of `et3compile` can replace deep memories less than 64 bits wide, with shallower 64-bit-wide memories. Compaction of memories is done automatically if the compile would fail without it.

  Memory compaction can either increase or decrease step count. Compaction can increase step count because compacting a memory increases its port count, so that more memory accesses must be scheduled within each FCLK. Compaction can decrease step count by reducing the amount of the emulator’s memory required for user memories, so that the compiler need not pack as many user memories into a given emulator memory.

© 2021 All Rights Reserved.

In rare cases, you can improve step count by forcing more compaction than occurs by default. To force extra compaction, use the `COMPACT` option of the `memoryTransform` command.

Using breakNet and breakPin

You can use the `breakNet` and `breakPin` userdata commands to break long paths at specific locations. In some designs, it is possible to find a small number of breakNets which break long paths and improve step count.

However, for most designs, use of breakNets should be considered only as a last resort. Use of breakNets has the following drawbacks:

- It is usually difficult to find a small number of breakNets that significantly improve the step count.
- The required breakNets can change as the design changes
- The added breaks can cause functional problems or require extra oversampling.

If you decide to use breakNets, the following guidelines can be helpful:

- Try to delay nets whose value never changes, or for which a one FCLK delay will not affect functionality.
- Avoid adding multiple delays on one combinational path.
- Avoid delaying clocks.
- Avoid delaying nets whose names can change with resynthesis.
- When delaying a bit of a bus, it is sometimes helpful to delay all bits of the bus.
- If the post-schedule path always ends in a certain module, it is sometimes helpful to put breaks on all inputs of that module.

The delay added by breakNet and breakPin is normally 1 FCLK cycle. For certain nets in 1X or 0.5X mode, or for all nets if the precompile -breakHalfCycle option is used, the default delay added by breakNet and breakPin is only one-half FCLK. A one-half FCLK break will not give any improvement in step count; in such a case, specify a breakNet delay of *2 units*, which is one FCLK cycle.

© 2021 All Rights Reserved.

Effect of Vision Mode

In some designs, your choice of vision mode can significantly affect step count. Usually, DYNP has the best step count, followed by FV.

After compiling, the `tmp/et3compile.msg` or `tmp/et3pass2.*.msg` files include the message:

```
SBSCHED  | Min step count due to probes is <number>
```

If <*number*> is close to the design step count, compiling for DYNP instead of FV, or compiling for more domains or boards, will usually improve the step count.

Use of compilerEffort, optimizationEffort and placementEffort

In Palladium Z1, you can specify a non-default value LOW for any of the compiler options `compilerEffort`, `optimizationEffort` and `placementEffort`. (`compilerEffort LOW` is equivalent to the combination of `optimizationEffort LOW` and `placementEffort LOW`). Usually, the default value `HIGH` for all these options gives the best step count.

Use of partitionerEffort

In Palladium Z1, you can specify a non-default value `MIN`, `LOW`, `MED`, or `HIGH` for the `partitionerEffort` compiler option.

It is suggested that you start with `AUTO`, move towards `HIGH` if performance is desired, or move towards `MIN` if speed is desired. Using `MIN` effort speeds up the compilation process, but it might prevent successful compilation of some multi-rack designs. Using `HIGH` effort improves performance, but it also increases the compilation time.

Using Design Retiming

Design flipflops and latches are implemented in Palladium using delay cells and combinational logic. Retiming is an optimization that moves delay cells within combinational logic to shorten combinational paths and improve emulation speed without affecting functionality. Retiming is enabled by default but can be disabled using the following command:

```
compilerOption -add {retiming off}
```

Do not disable retiming. The ability to disable retiming is provided only as a workaround in case of unexpected problems.

© 2021 All Rights Reserved.

Files to Send to Customer Support

Palladium Z1 Customer Support engineers can help improve emulation speed. When requesting assistance from Customer Support, always send the complete design database (actually, `QTDB` directory, `et3confg` file and `xe.msg` from a compile are enough) so that compile experiments can be performed at Cadence. If you cannot send the design directory, then send at least the following:

- complete `tmp` directory after running the following command

  `compileFind -all -max_trials <N> -crit_paths 20 -names_crit [other_options]`
- dump of userdata
- `xe.msg`
- `dbFiles/*mpart*` and `dbFiles/*.mem*` files
- `dbFiles/*.ctl` file
- `dbFiles/*confg*`
- `crit.v` and `crit.v.pathnet` files after running the `crit.sh` script as described earlier.

Improving Step Count for Designs with External Target Interfaces

Palladium Z1 has a long signal propagation delay from the emulator to the target and back. This delay is known as Round Trip Cable Delay (RTCD). It can result in an increase in step count. Try out the following methods to improve the step count:

- Reduce terminalTiming delay values
- Put breaknet on asynchronous reset pins
- Put breaknets on inputs and/or outputs if there is combinational logic at the top level
- Re-arrange terminalTiming constraints to sample inputs at time 0
- Use 5M cables instead of 30M cables

If these methods do not result in an improvement, then using Pipelined I/O might help improve the emulation performance for DUTs that have target I/O in the critical timing path. Normally, we schedule the outputs from the emulator to the target and expect to receive the inputs from the target in the same emulation cycle. However, some target protocols do not require receiving the inputs from the target in the same emulation cycle. In such a case, avoid waiting

© 2021 All Rights Reserved.

for the inputs to arrive before the end of the cycle, rather sample the inputs in the next cycle. This leads to inputs pipelining by adding one cycle delay between the outputs and the inputs. Pipelined I/O only affects timing of signals that are input to the emulator from external targets. For designs that either do not use targets or that have internal logic paths, which form the critical timing path, Pipelined I/O should have little or no effect on emulation performance. Shortening the longest timing paths results in databases that operate at faster emulation speeds.

A target interface that cannot tolerate an extra clock cycle of delay cannot use the new Pipelined I/O capability. On the other hand, serial protocols that incorporate pipelined transmit and receive data paths that have no input-to-output signal relationships within a clock cycle can use the new Pipelined I/O feature.

Ensure you perform the following steps to be able to use Pipelined I/O:

Determining Signals Included in Critical Timing Paths

Pipelined I/O can be used to improve emulation performance for DUTs that have target I/O in the critical timing path. Determine if target signals form part of the critical timing path, by running a critical path timing report during the design compilation process.

To generate a critical path timing report, modify the compile command as follows:

```
compileFind -best -max_trials <N> -names_crit -crit_paths <N>
```

This report lists the longest timing paths in the compiled database that impacts the emulation performance by increasing the overall step count of the design. If target interface input signals are part of the longest timing paths, then Pipelined I/O might be an option for increasing the emulation performance.

When analyzing timing paths that impact emulation performance, analyze the post-scheduling critical paths report produced by `getCriticalPathDesignNetNames` command to know if a particular signal is on the critical path.

See Critical Path Reports to know more about generating a critical path timing report.

Note: HDDC Muxing produces long timing chains of input and output signals at the target interface. These timing chains often become critical timing paths in the emulated design. Designs that use HDDC Muxing might see significant performance improvements when using Pipelined I/O. See Benefit of Advanced Capabilities of Terminal Timing Constraints for more information on HDDC Muxing.

© 2021 All Rights Reserved.

Determining Emulation Cables that Connect to Targets

Palladium Z1 supports two lengths of emulation data cables: 5M and 30M. Cable length has an impact on RTCD, and therefore can affect target I/O timing paths. Using 30M cables will result in longer timing paths as compared to 5M cables. Longer cables always result in worst timing. If you want to use longer cables, then Pipelined I/O can reduce the impact on step count. However regardless of the cable length, if RTCD is a contributing factor to the design performance, you must consider Pipelined I/O for performance improvement.

Use the following command to enable Pipelined I/O:

```
pipelineTargetInputs -add|-rm {<cable_label1>} {<cable_label2>}...
```

Here, `<cable_label>` refers to the label used in the `cableConnection` user data commands. You can enable Pipelined I/O for one or more individual cables, or can specify "*" in place of `<cable_label>` to enable Pipelined I/O for all Palladium Z1 target cables. For example, to enable or remove Pipelined I/O for all Palladium Z1 target cables including virtual targets, use the following command:

```
pipelineTargetInputs -add|-rm {*}
```

If there is a virtual target instance such as `MyTargetInst` created using `targetInst -add <target_inst_id> <target_location_id>` command that includes cable labeled `Cable0`, then to enable Pipelined I/O for that specific virtual target instance, use the following command:

```
targetType -add <target_type_name> {...pipelineTargetInputs -add
{<cable_label1>} {<cable_label2>}...}
```

Synchronizing with Target Clock to Resolve FCLK Delays

Sometimes, the target interface clock frequency is different from the design's fastest clock (FCLK) frequency. For interfaces running on CAKE clocks, introducing a one-FCLK delay on input data might result in the DUT not sampling the input data correctly. Some data may be repeated, while some may be dropped. This issue can be addressed using a FIFO for input signals when Pipelined I/O is enabled for a target interface.

See Chapter 13, "Using FIFO" for more information.

Troubleshooting System Problems

This section describes how to troubleshoot problems for running the Palladium Z1 software.

- **Problem:** The compilation process crashes or unpredictable failures occur during the debug process.

© 2021 All Rights Reserved.

Possible Reason: When you are compiling or debugging designs on Palladium Z1, it is possible that the maximum limit of open files on Linux is reached. This might cause the compilation process to crash or unpredictable failures during the debug process.

For example, if the maximum limit of file descriptors is reached then the following message is displayed and the program terminates:

```
Too many open files in system.
```

Solution: Evaluate the number of open files during both the compilation and debug processes. You can review the current maximum limit of all Linux shell system-wide resources by using the `limit` command in C shell or the `ulimit -a` command in Bash shell. Then, provide a reasonable Linux setting.

The recommended Linux settings for running the Palladium software are as follows:

```
cputime        unlimited
filesize       unlimited
datasize       unlimited
stacksize      unlimited
memoryuse      unlimited
vmemoryuse     unlimited
descriptors    8192
memorylocked   unlimited
maxproc        unlimited
```


© 2021 All Rights Reserved.

8

Behavior-Restricted Compilation

This chapter describes a method that you can use to enhance emulator utilization and emulation speed.

Introduction to Behavior-Restricted Compilation

Behavior-restricted Compilation is a strategy for optimizing emulator capacity and speed using behavioral data observed in an earlier runtime session of the design. The strategy is as follows:

- Step 1: Run Characterization Tests

  In this step, you run a set of characterization tests on the design. During these tests, the Palladium system collects information about the behavior patterns of certain *monitored nets* in the design.

  By default, most internal clock and reset nets are monitored, but you can specify additional nets to be monitored in a file named `instrumentEdges.request`. Example behavior patterns are:

  - Clock C1 was always low
  - Clock C2 never had a transition coinciding with negedge of the fastest CAKE clock

- Step 2: Behavior-Restricted Compile

  In this step, you compile the design with behavior restrictions. Behavior restrictions instruct the compiler to assume that the monitored nets will always follow the behavior patterns observed in Step 1: Run Characterization Tests. The assumption of behavior patterns can reduce the emulator capacity needed for the design, and can improve emulation speed.

- Step 3: Design Verification

  In this step, you run all available tests on the design, using the compiled database created in Step 2: Behavior-Restricted Compilation. The set of tests in this step is normally much larger than the set of characterization tests in Step 1. If a test in this step

© 2021 All Rights Reserved.

would normally cause a monitored net in the design to violate the assumed behavior patterns, the net in question will behave incorrectly; it will obey the behavior pattern assumed in the compile. However, the Palladium system will detect the violation and report the nets that had violations. Then, you can remove the assumptions about the reported nets, and recompile. For future versions of the design, you can either maintain a list of signals for which assumptions should not be made, or you can add the problematic test to the set of characterization tests.

Limitations of Behavior-Restricted Compilation

Behavior-restricted compilation is less convenient than ordinary emulation:

In ordinary emulation, the behavior of the emulated design always follows the design RTL, and any misbehavior indicates a problem in the RTL.

In behavior-restricted compilation, during design verification, you run new tests that were never run as characterization tests. If these new tests cause a monitored net to not follow its behavior pattern observed in the characterization tests, the emulated design will not behave according to the RTL. Palladium notifies you that a violation occurred, but for properly running design verification, you will need to correct the compilation assumptions, and recompile the design.

The behavior-restricted compilation strategy might be unsuitable for some designs. As an example, consider a design connected to a target system that behaves differently in every run. If the run-to-run variations in target system behavior cause monitored nets in the design to have different patterns of behavior in each run, then violations of the compile assumptions may occur in every run. To use the strategy, you must identify the monitored nets with varying behavior, and direct the compiler not to make assumptions about these nets.

Recommended Usage of Behavior-Restricted Compilation

Behavior-restricted compilation uses three types of information from the characterization tests:

- *Dead Clock* information identifies the monitored nets that have constant high or low values. The term Dead Clock is used for historical reasons. Dead Clocks need not actually be clocks.
- *Gated Clock* information identifies the monitored nets that have the same behavior, or the inversion of the behavior of a CAKE clock source.
- *Edge Alignment* information describes the relative alignment of the transitions of a monitored net with the transitions of the fastest CAKE clock.

© 2021 All Rights Reserved.

This section describes three recommended usages of behavior-restricted compilation:

- Two compilation steps, targeting `CAKE 2`

  The second compilation is for `CAKE 2`. It uses the Dead Clock and Gated Clock information from a run of the first compilation.

- Three compilation steps, targeting `CAKE 1`

  The third compilation is for `CAKE 1`. It uses the Dead Clock, Gated Clock, and Edge Alignment information from the runs of the first two compilations.

- Two compilation steps, targeting `CAKE 1`

  The second compilation is for `CAKE 1` It uses only Edge Alignment information from a run of the first compilation.

The three usages are described below, first for the ICE mode, followed by the SA mode.

ICE Mode

- Two compilation steps, targeting `CAKE 2`

  a. In directory D1, compile using `CAKE 2` and `precompileOption -add instrumentEdges`. Run characterization tests in directory `D1R`, which can be the same as `D1`.

  b. In directory `D2`, compile using `CAKE 2`, without `precompileOption -add instrumentEdges`, but with `precompileOption -add {1XrestrictPath <path_to_D1R>}`. Use this compiled database for design verification.

- Three compilation steps, targeting `CAKE 1`

  a. Same as Step a. above, using directories `D1` and `D1R`.

  b. In directory `D2`, compile using `CAKE 2`, `precompileOption -add instrumentEdges`, and `precompileOption -add {1XrestrictPath <path_to_D1R>}`. Rerun the same characterization tests in directory `D2R`, which can be the same as `D2`.

  c. In directory `D3`, compile using `CAKE 1`, without `precompileOption -add instrumentEdges`, but with `precompileOption -add {1XrestrictPath <path_to_D2R>}`. Use this compiled database for design verification.

- Two compilation steps, targeting `CAKE 1`

© 2021 All Rights Reserved.

a. Same as Step a. above, using directories `D1` and `D1R`.

b. In directory `D2`, compile using `CAKE 1`, without `precompileOption -add instrumentEdges`, but with `precompileOption -add {1XrestrictPath <path_to_D1R>}`. Use this compiled database for design verification.

SA Mode

Perform all compilations in this section with the `+1xua` option of the `ixcom` command. In the steps that use the `instrumentEdges` option, the xeCompile part of the compile is automatically executed in non-1xua mode, even though the `+1xua` option was specified.

In the SA mode, you normally specify the precompileOptions in the `compilerOptions.qel` file.

- Two compilation steps, targeting `CAKE 2`

  a. In directory D1, compile with `precompileOption -add instrumentEdges`. Run characterization tests in directory `D1R`, which can be the same as `D1`.

  b. In directory `D2`, compile without `precompileOption -add instrumentEdges`, but with `precompileOption -add {1XrestrictPath <path_to_D1R>}`. Use this compiled database for design verification.

- Three compilation steps, targeting `CAKE 1`

  a. In directory D1, compile with clock generation for CAKE 1 using the `ixclkgen` program. Use `precompileOption -add instrumentEdges`. Run characterization tests in directory `D1R`, which can be the same as `D1`.

  b. In directory `D2`, compile wit `precompileOption -add instrumentEdges` and `precompileOption -add {1XrestrictPath <path_to_D1R>}`. Rerun the same characterization tests in directory `D2R`, which can be the same as `D2`.

  c. In directory `D3`, compile without `precompileOption -add instrumentEdges`, but with `precompileOption -add {1XrestrictPath <path_to_D2R>}`. Use this compiled database for design verification.

- Two compilation steps, targeting `CAKE 1`

  a. Same as Step a. above in Three compilation steps, targeting `CAKE 1`, using directories `D1` and `D1R`.

  b. In directory `D2`, compile without `precompileOption -add instrumentEdges`, but with `precompileOption -add {1XrestrictPath <path_to_D1R>}`. Use this compiled database for design verification.

© 2021 All Rights Reserved.

Choosing a Usage

The preceding section described two usages targeting `1X` mode.

The advantage of the three-steps usage is that it uses all possible characterization information. So, it achieves the the best possible capacity and emulation speed.

The advantages of the two-step usage are:

- Faster compilation because it involves only two steps.
- If a net is identified as a Dead Clock during characterization, but becomes live during Design Verification, the three-step usage compilation assumption is violated, and a functional failure is likely. The two-step usage, on the other hand, makes no assumption about Dead Clocks. Therefore, no functional failure occurs.

Details of Behavior-Restricted Compilation

Enable behavior-restricted compilation by using the `instrumentEdges` and `1XrestrictPath` options of the `precompileOption` command. The following sections describe the steps in detail.

Step 1: Run Characterization Tests

This step involves the following sub-steps.

Step 1.1: Compile with the instrumentEdges option of the precompileOption command.

Use the following command:

```
precompileOption -add instrumentEdges
```

When you compile with the instrumentEdges option, logic is added to the design to monitor the behavior of most of the internal clocks and reset nets in the design. To include additional nets for monitoring, create a file named `instrumentEdges.request` in the design directory and include the names of the nets to be monitored, one net name per line.

For ICE mode, compile with CAKE 2. With ICE mode, the `instrumentEdges` option implies the `cake_like_1x` option, so that the CAKE clock waveforms will be as in CAKE 1. For SA mode, compile with `+1xua` option of the `ixcom` command. Generate clocks by a method that is compatible the `+1xua` option of the `ixcom` command. Cadence recommends using `ixclkgen` with CAKE 1.

© 2021 All Rights Reserved.

Step 1.2: Run the Characterization Tests

Characterization information of three types (Dead Clocks, Gated Clocks, and Edge Alignment) is recorded.

Edge Alignment information is similar to the information you provide using the `shadowsData` command. This information can improve emulator capacity and speed in a subsequent 1X mode compile. Dead-clock and gated-clock information might improve capacity or speed in a subsequent compilation for any mode.

Implementation Notes (Subject to Change)

When you run a design compiled with `instrumentEdges`, the behavior of certain nets in the design is monitored and recorded. The characterization information is stored in the following files in the `instrumentEdges` subdirectory of the run-time session directory:

- `dead.clocks`

  A monitored net whose value remains constant over the entire session is called a Dead Clock. For each Dead Clock, the `dead.clocks` file has the following pair of lines:

  ```
  <N>
  <value>
  ```

  Here, <*value*> is the value (`0` or `1`) that <*N*> had during the run.

- `gated.clocks`

  If the logic driving a monitored net <*N*> depends only on one CAKE clock <*C*>, and the <*N*> and <*C*> nets always have the same value, the `gated.clocks` file contains the following pair of lines:

  ```
  <N>
  == <C>
  ```

  If <*N*> and <*C*> always have opposite values, the `gated.clocks` file contains the following pair of lines:

  ```
  <N>
  != <C>
  ```

- `1Xmode.restrict`

  If a monitored net <*N*> is not dead, the `1Xmode.restrict` file uses two-digit codes to report any edge types that did not occur on <*N*>. The following codes are used:

  - `11`: on posedge of the fastest clock, net N never had a posedge
  - `10`: on posedge of the fastest clock, net N never had a negedge
  - `01`: on negedge of the fastest clock, net N never had a posedge

© 2021 All Rights Reserved.

- `00`: on negedge of the fastest clock, net N never had a negedge

The file contains pairs of lines:

```
<netname>
<code>
```

A given clock net has between zero and three such pairs. The pairs for a given net are usually not contiguous in the file.

Note: The content of the files depends only on the events that occurred during the current download session; not on the occurrences in previous sessions.

Scripts to Manipulate instrumentEdges Directories

The format of the files in the `instrumentEdges` directory might change in future releases. Do not directly edit these files. Instead, use the following scripts that are provided in the following directory:

`<install_dir>/share/uxe/gift/misc`

In each script, the comments at the beginning of the script give details about the usage.

`getEdges` reads the files in the `instrumentEdges` directory and creates a new directory `instrumentEdges_txt`, with `*.txt` files that are suitable for editing. The format of these files is user-friendly, with exactly one line per monitored net.

`putEdges` reads the `*.txt` files in an `instrumentEdges_txt` directory, and creates a new `instrumentEdges` directory.

`mergeEdges` merges the files created during multiple run-time sessions of an `instrumentEdges` design. The runs might be done in a single session directory, or in multiple session directories.

An alternative method of merging run-time sessions is described in the Controlling Behavior-Restricted Compilation section.

Step 2: Behavior-Restricted Compilation

In this step, compile the design with the following option:

```
precompileOption -add {1XrestrictPath <d>}
```

Here, <*d*> is the (absolute or relative) pathname to a run-time session directory of the same design compiled with the `instrumentEdges` option.

Following are three cases of compilation, called **1**, **2A**, and **2B**:

© 2021 All Rights Reserved.

- Case 1: the compilation is for ICE mode with CAKE 2 or higher:

  In this case, the compiler uses the Dead Clock and Gated Clock information from directory <*d*>, but not the Edge Alignment information. The compiler makes the following assumptions:

  - Dead Clocks: a net recorded in <*d*> as a Dead Clock will always be dead, with the same value it had in <*d*>.
  - Gated Clocks: a net N recorded in <*d*> as having behaved the same as C, or same as the inversion of C, will always have the recorded behavior.

- **Cases 2A and 2B**: the compile is ICE mode with `CAKE 1`, or SA mode with IXCOM compiler.

  - **Case 2A**: directory <*d*> was compiled without `1XrestrictPath` option.

    In this case, the compiler uses the Edge Alignment information from <*d*>, but not the Dead Clock or Gated Clock information. If a net *N* is a Dead Clock in <*d*>, the compiler makes no assumption about *N*. If net *N* had a transition in <*d*>, the compiler assumes that net *N* will never have an edge type that was not observed in <*d*>.

  - **Case 2B**: directory <*d*> was compiled with `1XrestrictPath` option.

    In this case, the compiler uses all three types of characterization information from <*d*>, making the following assumptions:

    - Dead Clocks: a net recorded in <*d*> as a Dead Clock will always be dead, with the same value it had in <*d*>.
    - Gated Clocks: a net N recorded in <*d*> as having behaved the same as C, or same as the inversion of C, will always have the recorded behavior.
    - Edge Alignment: a clock net *N* that had a transition in <*d*>, will never have an edge-type that was not observed in <*d*>.

    In **Case 2B**, the compiler uses information from the <*d*>`/instrumentEdges` and <*d*>`/PDB` directories.

Technical Notes

If you compile with `1XrestrictPath` <*d*>, the design for <*d*> directory must have the same gate-level netlist and same user data (except `instrumentEdges`, `1XrestrictPath` and `1XrestrictLevel` and CAKE number) as the current design.

When you compile with 1X mode and `1XrestrictPath`, the `precompileOption breakHalfCycleAll` option is automatically enabled. In some cases, this might cause a

© 2021 All Rights Reserved.

compile with `1XrestrictPath` to require more emulator capacity than an ordinary 1X mode compile without `1XrestrictPath`.

Implementation Note

A compile with `1XrestrictPath` stores the assumptions used in the compile in files `PDB/dead.clocks.compile`, `PDB/1Xmode.restrict.compile`, and `PDB/gated.clocks.compile` files.

Controlling Behavior-Restricted Compilation

For a compile with `1XrestrictPath` <*d*>, the compiler uses, by default, the information from directory <*d*> as described in **cases 1, 2A** and **2B** above. However, you can use the `1XrestrictLevel` option as follows:

```
precompileOption -add {1XrestrictLevel <string>}
```

This option specifies that the compiler should not use a type of information it would otherwise use. That is, you can say *Do not use dead clock information*, *Do not use gated clock information*, or *Do not use edge alignment information*.

In the `precompileOption` command given above, <*string*> can be a combination of the `1X`, `DEAD`, and `GATED` (case insensitive) tokens, separated by the plus (`+`) sign. If you omit one or more of the `1X`, `DEAD`, and `GATED` tokens from <*string*>, it specifies that the compiler should not use the information for which the token was omitted. For example, if <*string*> is `1X`, it means *Do not use gated clock information*, *Do not use dead clock information*.

Step 3: Design Verification

After the compilation in Step 2 finishes, run all available tests on the resulting compiled design.

If the tests that are run in this step would, in a normal compilation, cause a clock net to violate the assumptions that were used in the `1xrestrictMode` compilation, the clock net will behave incorrectly because it will obey the assumptions instead of violating them. So, the design might behave incorrectly. However, Palladium will detect the violation, and notify you in two ways:

- Signals indicating violations

  In any FCLK cycle in which there is a violation of the compilation assumptions, the signal `CVA_UA_VERIFY_SHADOWS_DATA` will have value `1`. If there is no violation, it has value

© 2021 All Rights Reserved.

`0`. With this signal, you can can use SDL to trigger when a violation occurs. See the Detecting shadowsData Violations section for more information.

Violations of any of the three types of assumption : Dead Clocks, Gated Clocks, and Edge Alignment are detected.

For each clock net, during precompilation, three signals are added to the design that can be used to detect violations on this net. The three signals are described in the Detecting shadowsData Violations section.

- Files indicating violations

  After running a design compiled with `1XrestrictPath`, when you release the emulator or exit xeDebug, files named `BAD_EDGES` and `BAD_GATED` are generated in subdirectory `instrumentEdges` of the run-time session directory. The `BAD_EDGES` file lists the names of all clock nets which had violations of the assumptions regarding Dead Clocks or Edge Alignment. The `BAD_GATED` file lists the names of all clock nets that had violations of the assumptions regarding Gated Clocks. If the files are empty, there were no violations. If either file is non-empty, the test results observed during the run cannot be trusted.

Technical Notes

To avoid differences in runs in CAKE 2 and CAKE 1, consider the following:

- In ICE mode with `timeUnit = fclk`, the command `run <N>` in `CAKE 1` runs <*N*> cycles of the fastest clock, but in `CAKE 2` it runs <*N*>`/2` cycles of the fastest clock. This could cause a run script to behave differently in `CAKE 1` and `CAKE 2`. To avoid this difference, use a `timeUnit` other than FCLK (for example, `dclk` or `ns`). Also, ensure a `run` <*N*> command always runs the exact number of cycles of the fastest design clock.
- When FCLK is free-running, `force` or `release` operations in `CAKE 1` are aligned to the falling edge of the fastest CAKE clock, but in `CAKE 2`, these operations could be aligned to either edge of the fastest CAKE clock. It is best to perform all initializations (`force`, `deposit`, `memory load`) before the `run` starts, or using a deterministic script with `run` <*N*> commands.

© 2021 All Rights Reserved.

9

Compiling Designs for In-Circuit Emulation with xeCompile

The ICE compilation flow involves a series of steps to convert the design from the format created by the design tools into the compilation database that can be downloaded to the emulator. The structure of the compilation database depends on the specifications given at each step of the compile process. This compilation flow can be used when the design components are known to be synthesizable and downloadable after compilation to the emulator. This compilation flow is also automatically invoked in Simulation Acceleration flow by IXCOM to compile the synthesizable portions for the emulator while partitioning the rest of the design portions to the software simulator. The Simulation Acceleration flow with IXCOM is described in Chapter 16, “Compiling Designs for Simulation Acceleration Using IXCOM.”

Important

The content of this chapter pertains only to ICE mode.

The compile process uses XEL commands, UNIX commands, and input and output files created to import, check, and compile a design.

The flow chart shown below depicts a typical ICE compilation flow.

© 2021 All Rights Reserved.

Figure 9-1 ICE Design Compilation Process

© 2021 All Rights Reserved.

Preparing for Design Import

Before importing the design into the emulator, perform the following tasks:

- Familiarize yourself with the directory structure.
- Create a design directory to store all emulation design files.
- Store the imported HDL source code files in a working library directory.

The following sections discuss the required preparations for design import.

Creating the Directory Structure

The directories that contain files used in xeCompile design import process are illustrated in Figure 9-2.

Figure 9-2 Directory Structure

```
<design_dir>
    ├── <work>       This directory contains all design emulation files.
    │
    └── <library>    This directory contains imported HDL source code files.
```

Design Directory

The directory, *<design_dir>*, is the design directory from which xeCompile is invoked. This directory will contain all of the RTL emulation design files, and can be anywhere on your network. It can contain the HDL source code that you want to import into xeCompile.

Working Library

A work library is a directory in which you store some or all of the imported HDL source code files. This library is not necessarily a subdirectory of the *<design_dir>*; it can be located anywhere on your network.

Note: For some designs, however, you might find it more convenient to store all of the design files, including all of the HDL working libraries, in the *<design_dir>* for the specific chip design project.

© 2021 All Rights Reserved.

A default work library named `work` is automatically created during the import process. The default location of this library is a subdirectory of the *<design_dir>*. If you use this default, all of the compiled HDL source code for the design is stored in this single subdirectory.

Create Additional Working Libraries

Create additional working libraries in any directory, and store portions of the compiled RTL source code in separate work libraries. Ideally, these additional work libraries should be within the same overall directory tree as the *<design_dir>*.

Relative path can be used to specify working libraries in the same directory tree as the *<design_dir>*. However, if a working library is in a different directory tree, specify the full path to the working library.

Setting the Absolute Path for Libraries

For changing the current working directory at any time during the import or synthesis process, use the absolute path to define logic libraries when importing the design. A logic library is a symbolic name for a directory, usually a subdirectory of the *<design_dir>*.

The symbolic names for the additional working libraries can represent any directory on the network. However, you should normally create the additional working libraries as subdirectories of the *<design_dir>*, and name the subdirectories the same as the working library name. This way, all files used in the design are in one directory, and the subdirectory names are easy to remember.

To set the absolute path for the libraries, execute the following XEL command before importing the design into xeCompile:

```
hdlDefineLib -add <library_name> <absolute_library_path>
```

For example:

```
hdlDefineLib -add IBMCU11VDK /nas/caenac/<working_directory>/log/siliconoptix/
                              sr32854150/TestCase_infra/work
```

Example: Creating a VHDL Work Library

Consider the following VHDL source code that contains references to VHDL libraries other than `work`:

```
# Reference first library, package, and object.
LIBRARY lib1;
USE lib1.pkg1.obj1;
# Reference second library, package, and object.
```


© 2021 All Rights Reserved.

```
LIBRARY lib2;
USE lib2.pkg1.obj1;
```

Note: Although the library names in this example are different (`lib1` and `lib2`), the package and object names are both `pkg1.obj1`, relative to each library.

In this example, before importing the design into the VHDL database, create two VHDL libraries, named `lib1` and `lib2` (refer to Figure 9-3).

Figure 9-3 Example of VHDL Libraries

While creating a VHDL logic library, assign the specified VHDL logic library *<lib>* to the selected directory *<dir>*. xeCompile then adds the new VHDL working library to the list of existing VHDL logic library names. In this example, the symbolic name lib1 should represent the `./lib1` path, and the symbolic name `lib2` should represent the `./lib2` path.

After creating these two VHDL directories, first import the two `pkg1.obj1` source code files.

- When the first source code file named `pkg1.obj1` is imported into the `lib1` VHDL library, importing compiles the file into xeCompile database format, and stores it in the *<design_dir>*`/lib1` directory.
- When the second source code file named `pkg1.obj1` is imported into the `lib2` VHDL library, importing compiles the file into the xeCompile database format, and stores it in the *<design_dir>*`/lib2` directory.

Now import the remaining VHDL files into the default VHDL library, named `work`. After compiling these source code files into the xeCompile database, the import process stores the compiled files in the *<design_dir>*`/work` directory.

In this example, if the VHDL libraries named `lib1` and `lib2` are not created, or the two `pkg1.obj1` files are not imported into these VHDL libraries, xeCompile import cannot find all of the information that it needs. xeCompile cannot import the RTL design into the work library, and reports an error.

© 2021 All Rights Reserved.

The VHDL work library created is a symbolic name, or alias, for the physical directory path. Therefore, you should normally specify a relative path, such as `lib1`. This ensures that all VHDL libraries used in one design are subdirectories of the same *<design_dir>*.

Also, the relative path should usually match the symbolic VHDL working library name. For example, it could be confusing if the symbolic name `lib1` represents the directory name `directory1`.

Specifying RTL Design Files for Import

Register Transfer Level (RTL) designs are those that use hardware description language, either Verilog or VHDL. When an RTL code is imported, xeCompile takes the code and maps it to functionally equivalent gate-level code.

By default, source files are imported incrementally. If the source was previously imported into the working library, or if the time stamp for the file being imported is earlier than the last import, source is not re-imported. This makes incremental import faster than a full import, as files need not be re-read and compiled unless it is necessary.

When importing a design for the first time, or to make sure the design database does not contain previously imported design files (for instance, if some parts of the design were deleted), select to fully import the design.

Note: Import now supports more than 1M terminals.

Full import performs the following operations:

- Deletes the specified Verilog HDL database or VHDL working library, including all design files previously imported into that database or library.
- Creates a new, empty Verilog HDL database directory or VHDL working library with the specified name.
- Imports all the specified files into the Verilog HDL database or VHDL working library.

The import procedure for Verilog and VHDL designs is discussed in detail in Specifying Verilog Designs for Import and Specifying VHDL Designs for Import.

Specifying Verilog Designs for Import

To import a Verilog RTL design into the emulation design database, use the following procedure.

© 2021 All Rights Reserved.

Note: The import and synthesis process of the HDL Compile tool supports a subset of Verilog IEEE STD 1364-2001.

1. Change to the emulation design directory, if not already in it, by executing the following command on the UNIX command-line prompt:

   `% cd <design_dir>`

2. Invoke xeCompile, if it is not already running.

   `% xeCompile`

3. (Optional) Specify the path and library name of the directory containing the Verilog HDL database by executing the following command on the `XEC>` prompt:

   `hdlWorkPath <dir>`

   A Verilog HDL database is a symbolic name, or alias, for a directory that stores compiled versions of the Verilog RTL design files that were imported.

   If the `hdlWorkPath` XEL command is not used, then the design files are imported into a default working library named `work`, representing a subdirectory named `work` in the emulation design directory.

4. Specify the names of the Verilog HDL source code files to import using the `hdlInputFile <file>` XEL command.

   The `<file>` option can include the path to the specific Verilog HDL source code file.

   Multiple `hdlInputFile` commands can be executed to add or delete from the list of design files to import. Refer to the *HDL-ICE Compiler Commands* chapter of the *VXE Command Reference Manual* for more details on the command syntax.

5. (Optional) Select an ASIC library to import using the `hdlInputFile -l <file>` XEL command.

   This step is required only if the Verilog RTL source code files refer to Verilog modules not defined within that source code file—that is, if a mixed-level design is being imported.

   When a gate-level netlist is synthesized from the imported design, the synthesis process first looks for cell definitions in the individual design files that were imported.

   For cells not defined in the imported files, synthesis uses cell definitions from the ASIC library selected, or treats them as “blackboxes” (empty cell definitions) if synthesis does not find cell definitions. Different parts of the design can use cell definitions from different ASIC libraries. The Import process resolves the blackboxes.

   The resulting gate-level netlist completely defines all cells in the design, regardless of where each cell definition originated.

© 2021 All Rights Reserved.

6. Execute the `hdlImport -2001` command to turn on the Verilog 2001 support and start the import process. By default, the `hdlImport` command updates or creates a library directory, as specified using the `hdlWorkPath` command in step 3.

   a. To import all Verilog RTL design files into the Verilog HDL database, use the following command:

      ```
      hdlImport -full
      ```

      Full import first removes all previously imported design files from the Verilog working library. It then imports all the Verilog RTL design files, even if none of these files were created or updated since the last Verilog design import.

   b. Using the `hdlImport` command without any parameter imports only new or updated Verilog RTL design files into the Verilog HDL database.

      By default, xeCompile imports the Verilog RTL design files incrementally. Thus, only Verilog RTL design files created or updated since the last design import are imported into the Verilog HDL database.

   c. For referencing a library use the `hdlImport -l <lib>` command.

      For example:

      ```
      hdlImport -l lib1 -l lib2
      ```

      If `<lib>` is an ASIC library, then the xeCompile looks for a cell definition in the ASIC library.

   Importing creates an HDL database directory under the `<design_dir>`. xeCompile checks the individual Verilog modules in the imported design files to verify that internal logic can be synthesized for each module. It also performs some interface checking.

   If a Verilog design file cannot be successfully imported, an error message is generated. If errors are encountered, fix those files for which errors are reported, and re-run with the Incremental Import function.

   For more details about the supported sections of the IEEE Standard Verilog Hardware Description Language, IEEE Std 1364-2001, refer to the `hdlImport -2001` command in the *HDL-ICE Compiler Commands* chapter of the *VXE Command Reference Manual*.

   Note: Alternatively, from the UNIX command-line interface, execute the `vavlog` utility comparable to the `hdlImport -2001` XEL command. It enables you to import Verilog source files. For syntax and usage details, refer to vavlog.

© 2021 All Rights Reserved.

Specifying VHDL Designs for Import

To import a VHDL design into the emulation design database, use the following procedure. The VXE Compiler import and synthesis for emulation supports a subset of VHDL IEEE STD 2008 and 1076-1993.

1. Change to the emulation design directory, if not already in it, by executing the following command on the UNIX command-line prompt:

   ```
   % cd <design_dir>
   ```

2. Invoke xeCompile, if it is not already running.

   ```
   % xeCompile
   ```

3. (Optional) Specify the path and library name of the directory containing the VHDL design database using the following command:

   ```
   hdlWorkPath <dir>
   ```

   A VHDL working library is a symbolic name, or alias, for a directory that stores the compiled versions of the imported VHDL RTL design files.

   If the `hdlWorkPath` command is not used, then the design files are imported into a default working library named `work`, representing a subdirectory named `work` in the emulation design directory.

4. Specify the HDL language of the design files that need to be imported. This can be done using the `hdlLang vhdl` command.

   Note: If this command is not used, xeCompile considers the default language as Verilog.

5. Specify the names of the VHDL source code files to import using the `hdlInputFile <file>` XEL command.

   The `<file>` option can include the path to the specific VHDL source code file.

   Use multiple `hdlInputFile` commands to add or delete from the list of design files to be imported. Refer to the *HDL-ICE Compiler Commands* chapter of the *VXE Command Reference Manual* for more details on the command syntax.

Execute the `hdlImport` command to start the import process. To import RTL code with VHDL 2008 constructs, use the `-v2008` option.

   a. To import all VHDL RTL design files into the VHDL working library, use the `-full` option with the `hdlImport` command as shown below:

   ```
   hdlImport -full
   ```


© 2021 All Rights Reserved.

Full import first removes all previously imported design files from the VHDL working library. It then imports all the VHDL RTL design files, even if none of these files were created or updated since the last VHDL design import.

b. Using the `hdlImport` command without any parameters imports only new or updated VHDL RTL design files into the VHDL working library.

By default, xeCompile imports the VHDL RTL design files incrementally. This option imports only VHDL RTL design files created or updated since the design was last imported into the VHDL working library.

Any design file not selected for import is not imported into the VHDL working library, even if it was created or updated since the design was last imported.

Any design file selected for import is not imported into the VHDL working library, if it was not created or updated since the design was last imported.

If this is the first time that the VHDL RTL design is being imported into the VHDL working library, incremental mode imports all selected design files, because they are all new files in the VHDL working library.

```
For referencing a library use either the hdlImport -l <lib> command or the
use clauses in VHDL code.
```

For example:

```
hdlImport -l lib1 -l lib2
library lib1, lib2; use lib1.pkg1.all; use lib2.all;
```

If *<lib>* is an ASIC library, then the xeCompile looks for a cell definition in the ASIC library.

Importing creates a VHDL working library under the *<design_dir>*, compiles the VHDL design into a format that xeCompile can synthesize into a gate-level netlist, and stores the compiled design in the VHDL working library.

xeCompile checks individual VHDL design units in the design files to ensure that the Synthesize HDL Design window can synthesize internal logic each VHDL design unit. It also does some interface checking.

If the Import HDL Design function cannot successfully import a VHDL design file, then it generates an error message. However, the importing continues until all source code files that can be successfully imported are in the VHDL work library. Fix the source code files for which errors are reported, and rerun the Import HDL Design process using the Incremental Import options. Only the changed files are imported.

Note: Alternatively, from the UNIX command-line interface, execute the `vavhdl` utility comparable to the `hdlImport` XEL command. It enables you to import VHDL source files. For syntax and usage details, refer to vavhdl.

© 2021 All Rights Reserved.

Specifying Mixed-Language Libraries for Import

Verilog modules and VHDL entities can be imported into the same library using separate hdlImport sessions as illustrated in the following example:

```
hdlLang vhdl
hdlInputFile -add sub.vhd
hdlImport
hdlLang verilog
hdlInputFile -add top.v
hdlImport
```

In the above example, the hdlLang vhdl command specifies that the HDL design file that needs to be imported into the working library is in VHDL. The hdlInputFile -add command adds the sub.vhd file to the list of HDL design files available for import using the hdlImport command. After the VHDL file has been imported into the working library, executing the hdlLang verilog command clears the list of files from hdlInputFile, and resets the type of HDL design file as Verilog. This enables the import of top.v Verilog file into the working library.

Now, consider the following example.

```
hdlLang vhdl
hdlInputFile -add sub.vhd
hdlLang verilog
hdlInputFile -add top.v
hdlImport
```

Here, only the top.v Verilog file is imported because the hdlImport command was not executed to import the sub.vhd VHDL file. Consequently, the hdlLang verilog command cleared the list of files specified using the first hdlInputFile command and reset the HDL language setting to import Verilog files only.

Note: You will get an information message when hdlLang is used while files are in the hdlInputFile list.

hdlLang also resets the current working directory (hdlWorkPath) to the default work. So, hdlWorkPath should always be used after hdlLang.

The following XEL commands will import the VHDL entities and Verilog modules into the ./libtest directory:

```
hdlLang vhdl
hdlWorkPath ./libtest
hdlInputFile -add sub.vhd
hdlImport
hdlLang vlog
hdlWorkPath ./libtest
hdlInputFile -add top.v
```


© 2021 All Rights Reserved.

```
hdlImport
```

Binding the Components

When a Verilog entity is bound to a mixed-language library, all Verilog modules in the library are searched first to find a case-sensitive match; then all VHDL entities in the library are searched to find a case-insensitive match.

Component binding in VHDL is case-insensitive, but component binding in Verilog is case-sensitive. Two Verilog modules of the same name of different case, such as ADDER and Adder, are allowed in a Verilog library and are considered different modules. However, they are considered the same entity if they are in VHDL. The latest imported VHDL entity is used.

You cannot use a Verilog module and a VHDL entity with the same name, and this will cause xeCompile to fail with an error message.

Synthesizing the RTL Design for Emulation

Designs of RTL format need to be synthesized before starting compilation. The synthesis process converts the imported RTL design written in Verilog or VHDL into a functionally equivalent gate-level netlist or a database.

Synthesis operates on the design files imported into the Verilog HDL database or VHDL working library, but stores the gate-level netlist and other synthesis results in your `<design_dir>` (usually one directory higher than the HDL database or working library).

Optionally, the synthesized (gate-level) netlist can also be saved as a file for use with design and verification tools other than Palladium Z1.

Cadence supports a synthesizable subset of Verilog and VHDL, and you can add the required emulation directives into the source code to constrain synthesis. For information about unsupported Verilog and VHDL functions, and about Cadence emulation directives, refer to Appendix 14, “Modeling Guide for Verilog Designs for ICE Mode” and Appendix 15, “Modeling Guide for VHDL Designs for ICE Mode.”

Cadence recommends using Palladium Z1 HDL-ICE Compiler to synthesize the RTL designs. For usage details of Palladium Z1 HDL-ICE Compiler, refer to the *HDL-ICE Compiler Commands* chapter in the *VXE Command Reference Manual*.

The Palladium Z1 HDL-ICE Compiler runs much faster (approximately 30 - 60 times faster) than silicon synthesis tools. It generates gate counts within ±10% of silicon-synthesis gate counts. This minor difference in gate count occurs because VXE compiler uses emulation-

© 2021 All Rights Reserved.

specific optimization, instead of the general-purpose technology mapping used by silicon synthesis tools.

The topics covered in this section are:

- Synthesizing Verilog or VHDL Designs
- Synthesizing Mixed-Language Designs
- Incremental Synthesis
- Preserving the RTL Names
- Maintaining Log Files
- Using Synthesis Directives for Emulation
- Synthesizing Designs Using Old Synthesis Engine
- Synthesizing Designs with Memories
- Synthesizing Designs with Acceleratable Testbench (ATB) Constructs
- Executing Distributed Synthesis during RTL Synthesis
- Using Conformal to Verify HDL-ICE Compiler-Generated Netlist
- Using UNIX Commands for RTL Import and Synthesis

Synthesizing Verilog or VHDL Designs

After importing the RTL design, the VXE compiler takes the RTL code and synthesizes it. Synthesis operates on the design files imported into the HDL database, and stores the gate-level netlist and other synthesis results in the `<design_dir>` for mapping into the PDB database for compile. The RTL design can be in Verilog or VHDL. A subset of VHDL IEEE STD 1076-1993 can be imported and synthesized for emulation.

Following steps describe the procedure for synthesizing a Verilog or VHDL RTL design:

1. Specify the file in which the output of HDL synthesis should be written by using the `hdlOutputFile` command.
2. (Optional) If certain modules or VHDL design units should not be synthesized, declare those using the `hdlSkipModule` command.
3. Use the `hdlSynthesize` command to perform the synthesis. Unless the `hdlImport -full` command was used during the design import, this command performs content-

© 2021 All Rights Reserved.

based incremental synthesis. This means that a module is synthesized again only if its contents are different from the previously synthesized design.

Synthesizing All or Specific Modules:

All modules and design units in the work library can also be synthesized by using the `-all` option with the `hdlSynthesize` command. However, this option should be used only when compiling the cells of a library that does not have a top module or design unit.

Note: Avoid using the `hdlSynthesize -all` command with the `hdlOutputFile -f qel` command, or the output file will not be functional.

You need to specify the name of the top-level module as the <*top_module*> or <*top_unit*> (for VHDL designs) option of the `hdlSynthesize` command.

The top module is the name of the Verilog design file describing the top level of the design hierarchy to be synthesized. The top design unit is the name of the VHDL design file that contains the entity and architecture description for the top-most design unit to be synthesized.

The specified top-level module (or top design unit) and all modules (and design units) in the design hierarchy below it are synthesized. If you do not want to synthesize the entire design, specify the name of the Verilog design module (or design unit) from which synthesis should start. The selected module (or design unit) is regarded as the top-level module (or design unit) of the design, and design modules (and units) at or above the same hierarchical level as the specified top-level module (or design unit) are not synthesized.

Synthesizing Memories:

To synthesize Verilog or VHDL memories use the `hdlSynthesize -memory` command. This enables to synthesize memories into constructs that preserve the functions defined in the RTL source code. For detailed information of the memory synthesis process, refer to Synthesizing Designs with Memories.

Using the `-keepVhdlCase` option with the `hdlSynthesize` command retains the case of the names within the VHDL design file as much as possible. If this command option is not specified, the VHDL names are set to all-uppercase in the output.

The length of time required to synthesize a design depends on the size and complexity of the design. While synthesis is running, messages describing the progress of the process are displayed on the screen.

© 2021 All Rights Reserved.

Synthesizing Mixed-Language Designs

Mixed-language designs can also be synthesized using the XEL commands. A mixed-language design consists of modules in libraries with different languages.

Use the following commands:

- `hdlDefineLib`: Defines the directory path of either a Verilog or VHDL library.
- `hdlSynthesize`: Synthesizes an RTL design to get a gate-level netlist in Verilog, in VHDL, or in both Verilog and VHDL.
- `hdlImport`: Imports VHDL and Verilog files into different libraries.
- `hdlSynthesize -keepVhdlCase`: Retains the case of all the names in the RTL design in the Verilog and VHDL netlists.
- `hdlSynthesize -vhdlToLowercase`: Sets the case of all names in the RTL design to lower case in the Verilog and VHDL netlists.
- `hdlImport -l <lib>`: References libraries to search for modules undefined in library work.

Important

If a VHDL module is instantiated by a Verilog module, import the VHDL part of the RTL code first, then import the Verilog part of the RTL code.

Using VHDL Port Types for Verilog

To successfully use a mixed-language design that contains both Verilog and VHDL code, use the correct type of VHDL ports to interface with Verilog. These ports must be `std_logic` or `std_logic_vector`, `std_ulogic`, `std_ulogic_vector`, `bit`, or `bit_vector`.

To use either `std_logic` or `std_logic_vector` port type in VHDL, include the following statements:

```
library ieee;
use ieee.std_logic_1164.all;
```

Example: Mixed-Language Synthesis

To synthesize a mixed-language design, run a script similar to the following example:

```
hdlLang vhdl
hdlWorkPath lib1
hdlInputFile a1.vhd a2.vhd
```


© 2021 All Rights Reserved.

```
hdlImport
hdlLang verilog
hdlWorkPath lib2
hdlInputFile b1.v b2.v
hdlImport -l lib1
hdlOutputFile -f verilog g.ver
hdlSynthesize -keepVhdlCase verilog_top
```

Incremental Synthesis

The Palladium Z1 compiler supports the following types of incremental synthesis:

- Time-based or file-based incremental import – Only files with new time stamp will be re-imported.
- Content-based or module-based incremental synthesis – Only modified modules will be re-synthesized.

By default, incremental synthesis is enabled. If you are working with a previously synthesized gate-level netlist of a Verilog RTL design, only those Verilog design modules imported again are synthesized again. If the entire Verilog RTL design was fully imported, the entire design is synthesized again.

To ensure that all the design components are re-synthesized completely, use the following command at the time of HDL design import:

```
hdlImport -full
```

Content-based incremental synthesis enables you to quickly implement the Engineering Change Orders (ECOs). Additionally, Palladium Z1 compiler can perform a high-capacity synthesis of RTL designs to generate the gate-level netlist. It can process large, multi-million gate designs, using the available disk swap space of a workstation's memory. You do not need to manually partition the original RTL design into blocks for synthesis.

The names of all RTL signals and cell instances are preserved during the netlist generation; so it is easier to debug the design.

The memory information (such as number of ports, and width of the address and data bus) is extracted from the RTL Verilog or VHDL designs, and the memory compilation information is written directly into the design database. Optionally, you can generate VHDL or Verilog memory files when synthesizing memories. You do not need to synthesize memories as blackboxes, unless you want to change the emulator memory configuration from the default.

© 2021 All Rights Reserved.

Preserving the RTL Names

RTL name preservation enables for easy debugging. To use name preservation, follow the instructions below:

- Do not remove RTL net during optimization.

```
assign z1 = (!a & b) | (a&!b); Q_XOR2 u1 (z1, a, b);
                               Q_INV u1 (n1,a);
assign z1 = !a & b;                Q_INV u2 (n2,b);
assign z2 = a & !b;                 Q_AN02 u3 (z1, n1, b);
assign z = z1 | z2;                 Q_AN02 u4 (z2, n2, a);
                                    Q_OR02 u5 (z, z1, z2);
```

- Insert buffers for equivalent RTL nets:

```
input [1:0] a;
wire [1:0] b;                   Q_BUF
                                u1 (.Z(b[0]), .Z (a[1]));
if (a==2'b10) b=2'b11;
else if (a==2'11) b=2'b01;
else b=2'b10;
```

- Signals declared at the module level are 100% preserved. Variables declared locally inside a block, process, function, or task might not be fully preserved.

Using Command Options for Name Preservation

- Turn off name preservation to save gate counts:

```
hdlSynthesize -noNamePreservation
```

- Turn off name preservation on nets without fanout to primary output:

```
hdlSynthesize -fanoutNetOnly
```

- Avoid merging equivalent flip-flops and optimizing out flip-flops with constant inputs (useful for easy formal verification) and “force value” debugging.

  Below is an example of `hdlSynthesize` without the `-keepAllFlipFlop` option:

```
hdlSynthesize
always @(posedge clk)      Q_FDP0 ul(.CK(clk),.D(c),.Q(a));
begin                      Q_BUF u2(.Z(b),.A(a));
a = c;
b = c;
end
```

  Below is an example of `hdlSynthesize` with the `-keepAllFlipFlop` option:

```
hdlSynthesize -keepAllFlipFlop
always @(posedge clk)
begin
a = c;                     Q_FDP0 u1(.CK(clk),.D(c),.Q(a));
b = c;                     Q_FDP0 u2(.CK(clk),.D(c),.Q(b));
end
```


© 2021 All Rights Reserved.

- For VHDL, a record-type signal name “a.b”, use “\a.b” instead of “a_b” in the netlist by using:

  ```
  hdlSynthesize -keepRtlSymbol
  ```

Handling Case Sensitivity

- To import Verilog in a case-insensitive way:

  ```
  hdlImport -u
  ```

- By default, RTL VHDL symbols are converted into all uppercase in gate-level netlists.
- To preserve the VHDL case declared at RTL use:

  ```
  hdlSynthesize -keepVhdlCase
  ```

- To convert RTL VHDL symbols at all lowercase use:

  ```
  hdlSynthesize -vhdlToLowerCase
  ```

Maintaining Log Files

By default, the HDL synthesis log is maintained in the `hdlIce.log` file. To specify a user-specific log file, use the following command:

```
hdlLogfile <mylogfile>
```

Like other XEL commands executed on the xeCompile CLI, the commands executed for synthesizing the RTL designs are also recorded in the `xe.msg` file.

Using Synthesis Directives for Emulation

A set of directives are added to constrain the RTL synthesis process, and to help you to synthesize a design for emulation. Use these directives to debug designs and facilitate emulation in HDL designs. The HDL-specific synthesis directives are discussed below.

The following directives are supported in VHDL designs:

```
built_in SYN_INTEGER_TO_UNSIGNED
built_in SYN_UNSIGNED_TO_INTEGER
built_in SYN_INTEGER_TO_SIGNED
built_in SYN_SIGNED_TO_INTEGER
built_in SYN_ZERO_EXTEND
built_in SYN_SIGN_EXTEND
resolution_method three_state
resolution_method wired_and
```


© 2021 All Rights Reserved.

```
resolution_method wired_or
translate_off
translate_on
synthesis_off
synthesis_on
preserve
keep_net
```

The following directives are supported in Verilog designs:

```
translate_off
translate_on
parallel_case     // all items are mutually exclusive
full_case         // full coverage
preserve
keep_net
```

The default prefixes can be `pragma`, `synopsys`, and `quickturn`.

Use the command-line option `-synth_prefix` to add prefixes, and `-rm_synth_prefix` to remove `pragma` or `synopsys` default prefix. For example:

```
hdlInputFile -synth_prefix <pragma_name>
```

Here, *<pragma_name>* is the string used in the design.

Note: If the `-rm_synth_prefix` option is used, the `quickturn` pragma cannot be removed.

Examples

The following example shows the support of `built_in SYN_UNSIGNED_TO_INTEGER`:

```
function typeconv ( s : std_ulogic_vector ) return integer is:
-- pragma built_in SYN_UNSIGNED_TO_INTEGER
begin
--  synopsys translate_off
    <Actual description for this function>
--  synopsys translate_on
return int_result;
end typeconv;
```

In this example, *<Actual description for this function>* is ignored and `function typeconv` is replaced by `built-in function SYN_UNSIGNED_TO_INTEGER` in xeCompile.

Usage is the same for the following:

```
built_in SYN_INTEGER_TO_UNSIGNED
```


© 2021 All Rights Reserved.

```
built_in SYN_UNSIGNED_TO_INTEGER
built_in SYN_INTEGER_TO_SIGNED
built_in SYN_SIGNED_TO_INTEGER
built_in SYN_ZERO_EXTEND
```

The following are used to define the resolution methods which are not synthesizable.

```
resolution_method three_state
resolution_method wired_and
resolution_method wired_or
```

Use the following directive definitions:

```
translate_off / synthesis_off
```

Stops checking the input file from this point to the end of the file, or until finding a `translate_on` directive.

```
translate_on / synthesis_on
```

Continues checking the input file from this point to the end of the file, or until finding a `translate_off` directive.

Preserving Expected Behavior when Nets are Forced

The synthesis process might not preserve the expected behavior of some RTL signals when other signals are forced. For example, suppose the design contains the following line:

```
wire B = A;
```

Then, the synthesis process considers `A` and `B` equivalent.

If the design contains another signal depending on `B`, for example:

```
wire D = B & C;
```

Then, because `A` and `B` are considered equivalent, the synthesis process might use `A` instead:

```
wire D = A & C;
```

Here, `A` is more likely to be used instead of `B` if `A` is the output of a flip-flop.

With this implementation, if you force signal `B`, then, contrary to expectation, signal `D` is not affected.

To inform the synthesis process that a certain signal might be forced at run time, use the `keep_net` directive. The directive must appear after the declaration of the affected signal.

In each module where the `keep_net` directive is used, include the following module-level `keepNet` directive:

© 2021 All Rights Reserved.

```
// quickturn keepNet
```

keep_net pragma works for IXCOM flow only. To keep record of nets with keep_net property of IXCOM flow, keepNet directive has been introduced for ICE flow. There is a difference between the spelling of keep_net and keepNet in the two directives. The module-level keepNet directive does not specify a particular signal name. This directive instructs the synthesis process to pass any keep_net declarations for the current module to xeCompile (using PROP_KEEPNET pragmas in the netlist) so that the keep_net signals can be forced at run time without needing to specify them with the keepNet user data command.

keepNet directive obeys the following rules:

- keepNet must work together with keep_net directive. For example,

  ```
  // quickturn keepNet
  wire foo;
  // quickturn keep_net foo
  ```

  This results in HDL-ICE Compiler writing out the following pragma information into the netlist:

  ```
  // pragma CVASTRPROP NET foo PROP_KEEPNET " "
  ```

- keepNet takes effect in the entire module scope where it is inserted.
- keepNet only propagates for single scalar or vector type signals. As Palladium does not allow force on memories, so HDL-ICE Compiler does not support propagation of keepNet on complex type signals, which may become memories.
- keepNet can be located anywhere in the declarative part of a module. Cadence suggests adding keepNet before port_declaration.

Note: In an SA Mode compile, or in an ICE Mode compile where vaelab -KeepAllFlipFlop option is used, the synthesis process automatically treats any flipflop output, latch output, or undriven net as keep_net, except that the keep_net property is only propagated to xeCompile if an explicit keep_net pragma is given.

Here is a Verilog example:

```
// quickturn keepNet
wire A;
// quickturn keep_net A
```

Here is a VDHL example:

```
-- quickturn keepNet
signal A : bit;
-- quickturn keep_net A
```

The keep_net directive is effective for Verilog wires, and for VHDL signals and Verilog registers that are assigned only by non-blocking assignments. If a signal or register is

© 2021 All Rights Reserved.

assigned by multiple blocking assignments in an `always` block or process, multiple names might be used and the `keep_net` directive might not have the desired effect.

Supporting keepNet for Multi-Dimensional Array

Only a single pragma is added for each multi-dimensional array indicating whether the whole array is a `keepNet` or not.

If you need to mark the whole array as `keepNet`, ensure only the name of array appears in pragma.

Example:

```
// pragma HDLICE module keepnet A
```

NOT

```
// pragma HDLICE module keepnet A[0][1]
```

Preserving Signal Names

The synthesis process preserves module-level signal names, but it does not always preserve the signals declared in a local name scope such as a block or process. To preserve such signals, use the `preserve` directive. The directive must appear after the declaration of the affected signal.

Here is a Verilog example:

```
wire A;
// quickturn preserve A
```

Here is a VHDL example:

```
signal A : bit;
-- quickturn preserve A
```

Synthesizing Designs Using Old Synthesis Engine

CUDD (CU Decision Diagram) is the new synthesis engine that is set as the default engine to support Binary Decision Diagram (BDD) related operations in HDL-ICE Compiler.

This new synthesis engine also provides support for parallel synthesis. In most cases, this new synthesis engine ensures that the HDL-ICE Compiler does not turn unresponsive when handling BDD.

© 2021 All Rights Reserved.

In some cases when the new synthesis engine does not respond, use `-legacy_bdd` option with `vaelab` and `hdlSynthesize` commands to revert to the old, legacy synthesis engine, which might work fine.

For syntax and usage details, refer to `vaelab` section and `hdlSynthesize` command in *HDL-ICE Compiler Commands* chapter of the *VXE Command Reference Manual*.

Synthesizing Designs with Memories

All the memories not modeled as gates are modeled with VXE memory primitives. These primitives are MPR (Memory Port Read) and MPW (Memory Port Write). These primitives can be combined to create a memory with any number of read and write ports. You can also create read-before-write and write-through ports. Cadence recommends that you obtain assistance from the respective Field Application Engineer before using these primitives.

Note: These primitives cannot be simulated, only accelerated. So, a congruent simulation is not possible.

Memory synthesis enables you to synthesize a wide variety of memory constructs, including variable arrays, signal arrays, multi-level nested arrays, sliced indexes, and both level-sensitive and edge-sensitive write enables. To preserve the original RTL functionality, memory synthesis does the following:

- Automatically coordinates multiple sequential memory access timing.
- Places address offsets in the memory.
- Wraps extra “glue logic” around the synthesized memory definition, as needed. This extra logic ensures that the emulation memory more closely resembles the original memories, as defined in the RTL source code.
- Calculates appropriate memory size.
- Models level-sensitive and edge-sensitive write enables within the memory.
- Performs port sharing and memory merging, to reduce the number of ports and gates in the memory.

Using Memory Synthesis

By default, memory synthesis is turned off, and all the arrays at RTL are mapped to gates and flip-flops in the output netlist. For write-only memory, in the netlist, the HDL-ICE Compiler keeps only the memories that are declared at module level.

© 2021 All Rights Reserved.

Note: Using xeCompile without memory synthesis for large arrays can cause gate count explosion.

Turn on memory synthesis by using the following XEL command:

```
hdlSynthesize -memory
```

Note: Alternatively, from the UNIX command-line interface, execute the `vaelab` utility comparable to the `hdlSynthesize -memory` XEL command. This utility enables you to synthesize the design from top module. For syntax and usage details of the `vaelab` utility, refer to vaelab.

Using Memory Synthesis Selectively

Use the `hdlSynthesize -memory<n>` command to notify xeCompile that all memories smaller than <*n*> bits (width x depth) should be mapped to gates and flip-flops. For example:

```
hdlSynthesize -memory256 TOP
```

Note: Notice that there is no space between `-memory` and `256`. The `-memory256` option informs xeCompile that any memory smaller than 256 bits (width x depth) should be mapped to gates and flip-flops. In certain cases, larger memories might also be mapped to gates and flip-flops, but definitely all memories smaller than 256 bits will be. If the `-memory` option is specified with no number, the default is 32.

In some designs, using `-memory<n>` with <*n*> larger or smaller than the default improves the emulation speed or capacity.

Use the `no_hardmacro` directive, `quickturn no_hardmacro <array_name>` to tell xeCompile to map a specific memory to gates and flip-flops. For example, in VHDL:

```
type mem_type is array(127 downto 0) of std_logic_vector(0 downto 0);
signal MemArray : mem_type;
-- quickturn no_hardmacro MemArray
```

or in Verilog:

```
reg  [DATA_WIDTH-1:0] MemArray [((1<<ADR_WIDTH)-1):0];
// quickturn no_hardmacro MemArray
```

The opposite of `no_hardmacro` is `use_hardmacro`. The `use_hardmacro` directive tells xeCompile to map a specific memory to VXE memory primitives regardless of how small the memory is.

© 2021 All Rights Reserved.

Modeling Verilog Memories

This section gives examples of memory RTL descriptions which xeCompile can synthesize to VXE memories. In these examples, each module contains only one memory. However, xeCompile can also synthesize multiple memories within a single module.

Asynchronous Verilog Memory

The following example illustrates an asynchronous memory with one read-write port:

```
module mem(ADDR, DI, WE, DO);
parameter ADR_WIDTH  = 4;
parameter DATA_WIDTH = 32;
input  [ADR_WIDTH-1:0]      ADDR;  // Address input
input  [DATA_WIDTH-1:0]     DI;    // Data input
input                       WE;    // Write enable
output [DATA_WIDTH-1:0]     DO;    // Data output
reg  [DATA_WIDTH-1:0] MemArray [((1<<ADR_WIDTH)-1):0];
always @ (WE or ADDR or DI)
  begin
    if(WE)
      MemArray[ADDR] <= DI;
  end

assign DO = MemArray[ADDR];
endmodule
```

Synchronous Verilog Memory

The following example illustrates a synchronous memory with two read ports and two write ports, all controlled by a single clock. In this example, the assignments to MemArray[WAC] and MemArray[WAD] are done in a single `always` statement to avoid ambiguity when ports C and D write to the same address on the same rising edge of CLK. To avoid differences between emulation and simulation, avoid writing Memory RTL, whose behavior is ambiguous.

```
module mem(CLK, RAA, DOA, RAB, DOB, WAC, DIC, WEC, WAD, DID,WED);
parameter ADR_WIDTH  = 4;
parameter DATA_WIDTH = 32;
input                       CLK;  // Clock input
input  [ADR_WIDTH-1:0]      RAA;  // Address input, port A
output [DATA_WIDTH-1:0]     DOA;  // Data output, port A
input  [ADR_WIDTH-1:0]      RAB;  // Address input, port B
output [DATA_WIDTH-1:0]     DOB;  // Data output, port B
input  [ADR_WIDTH-1:0]      WAC;  // Address input, port C
input  [DATA_WIDTH-1:0]     DIC;  // Data input, port C
input                       WEC;  // Write enable, port C
input  [ADR_WIDTH-1:0]      WAD;  // Address input, port D
input  [DATA_WIDTH-1:0]     DID;  // Data input, port D
input                       WED;  // Write enable, port D
reg  [DATA_WIDTH-1:0] MemArray [((1<<ADR_WIDTH)-1):0];
reg  [ADR_WIDTH-1:0]  RegRAA;
reg  [ADR_WIDTH-1:0]  RegRAB;
always @ (posedge CLK)
  begin
```


© 2021 All Rights Reserved.

```
    if(WEC)
      MemArray[WAC] <= DIC;
    if(WED)
      MemArray[WAD] <= DID;
  end

always @ (posedge CLK)
  begin
    RegRAA <= RAA;
  end

assign DOA = MemArray[RegRAA];
always @ (posedge CLK)
  begin
    RegRAB <= RAB;
  end

assign DOB = MemArray[RegRAB];
endmodule
```

Synchronous Verilog Memory with Byte Write

The following example illustrates a synchronous memory with one read-write port. It has a byte-write input allowing individual 8-bit-wide bytes to be written. A similar approach can be used to model a bit-write input, or byte-write with any width byte.

```
module mem(CLK, ADDR, DI, WE, BW, DO);
input             CLK;   // Clock input
input  [3:0]      ADDR;  // Address input
input  [31:0]     DI;    // Data input
input             WE;    // Write enable
input  [3:0]      BW;    // Byte write
output [31:0]     DO;    // Data output
reg  [31:0] MemArray [15:0];
wire [31:0] TmpData;
wire [31:0] DataIn;
reg  [3:0]  RegADDR;
assign TmpData = MemArray[ADDR];
assign DataIn[31:24] = (BW[3]) ? DI[31:24] : TmpData[31:24];
assign DataIn[23:16] = (BW[2]) ? DI[23:16] : TmpData[23:16];
assign DataIn[15:8]  = (BW[1]) ? DI[15:8]  : TmpData[15:8];
assign DataIn[7:0]   = (BW[0]) ? DI[7:0]   : TmpData[7:0];
always @ (posedge CLK)
  begin
    if(WE)
      MemArray[ADDR] <= DataIn;
  end

always @ (posedge CLK)
  begin
    RegADDR <= ADDR;
  end

assign DO = MemArray[RegADDR];
endmodule
```


© 2021 All Rights Reserved.

Modeling VHDL Memories

This section gives examples of memory RTL descriptions which xeCompile can synthesize to VXE memories. In these examples, each module contains only one memory. However, xeCompile can also synthesize multiple memories within a single module.

Asynchronous VHDL Memory

The following example illustrates an asynchronous memory with one read-write port.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity mem is
  port (
    ADDR : in std_logic_vector(3 downto 0);
    DI : in std_logic_vector(31 downto 0);
    WE : in std_logic;
    DO : out std_logic_vector(31 downto 0)
  );
end  mem;

architecture mem_arch of mem is
type mem_type is array(15 downto 0) of std_logic_vector(31 downto 0);
signal MemArray : mem_type;
begin
  process( WE, ADDR, DI )
  begin
    if( WE = '1' ) then
      MemArray(conv_integer(ADDR)) <= DI;
    end if;
  end process;

  DO <= MemArray(conv_integer(ADDR));
end mem_arch;
```

Synchronous VHDL Memory

The following example illustrates a synchronous memory with two read ports and two write ports, all controlled by a single clock. In this example, the assignments to MemArray[WAC] and MemArray[WAD] are done in a single process, to avoid ambiguity when ports C and D write to the same address on the same rising edge of CLK. To avoid differences between emulation and simulation, avoid writing Memory RTL whose behavior is ambiguous.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity mem is
  port (
    CLK : in std_logic;
    RAA : in std_logic_vector(3 downto 0);
    DOA : out std_logic_vector(31 downto 0);
```


© 2021 All Rights Reserved.

```
    RAB : in std_logic_vector(3 downto 0);
    DOB : out std_logic_vector(31 downto 0);
    WAC : in std_logic_vector(3 downto 0);
    DIC : in std_logic_vector(31 downto 0);
    WEC : in std_logic;
    WAD : in std_logic_vector(3 downto 0);
    DID : in std_logic_vector(31 downto 0);
    WED : in std_logic
  );
end  mem;

architecture mem_arch of mem is
type mem_type is array(15 downto 0) of std_logic_vector(31 downto 0) ;
signal MemArray : mem_type;
signal RegRAA : std_logic_vector(3 downto 0) ;
signal RegRAB : std_logic_vector(3 downto 0) ;
begin
  process( CLK )
  begin
    if( CLK'event and CLK = '1' ) then
      if( WEC = '1' ) then
        MemArray(conv_integer(WAC)) <= DIC;
      end if;
      if( WED = '1' ) then
        MemArray(conv_integer(WAD)) <= DID;
      end if;

    end if;

end process;

process( CLK )
begin
  if( CLK'event and CLK = '1' ) then
    RegRAA <= RAA;
  end if;
end process;
DOA <= MemArray(conv_integer(RegRAA));
process( CLK )
begin
  if( CLK'event and CLK = '1' ) then
    RegRAB <= RAB;
  end if;

end process;

DOB <= MemArray(conv_integer(RegRAB));
end mem_arch;
```

Synchronous VHDL Memory with Byte Write

When a memory write operation does not write every bit in the data bus, the xeCompile memory synthesis uses a read-modify-write approach to implement it. An extra read port with the same address is created, if it does not already exists. Then, the multiplexers are synthesized for the individual bit-enable signal. This synthesis process helps in identifying the memory data input, which can be either the data input or the data output read from the read port.

© 2021 All Rights Reserved.

When the width of a segment selected by each write-enable condition is not in multiples of 8, using the read-modify-write approach derives better results. Forcing the use of byte-select signals in the hardware can increase the data width of the memory. Therefore, when the width of each segment is in multiples of 8, xeCompile tries to utilize the new byte-select pins of the memories to save the capacity for an extra read port and additional multiplexers.

The following example illustrates a synchronous memory with one read-write port. It has a byte-write input allowing individual 8-bit-wide bytes to be written. A similar approach can be used to model a bit-write input, or byte-write with any width byte.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity mem is
  port (
    CLK : in std_logic;
    ADDR : in std_logic_vector(3 downto 0);
    DI : in std_logic_vector(31 downto 0);
    WE : in std_logic;
    BW : in std_logic_vector(3 downto 0);
    DO : out std_logic_vector(31 downto 0)
  );
end  mem;

architecture mem_arch of mem is
type mem_type is array(15 downto 0) of std_logic_vector(31 downto 0) ;
signal TmpData : std_logic_vector(31 downto 0) ;
signal DataIn : std_logic_vector(31 downto 0) ;
signal MemArray : mem_type;
signal RegADDR : std_logic_vector(3 downto 0) ;
begin
  TmpData <= MemArray(conv_integer(ADDR));
  DataIn(31 downto 24) <=
    DI(31 downto 24) when BW(3) = '1' else TmpData(31 downto 24);
  DataIn(23 downto 16) <=
    DI(23 downto 16) when BW(2) = '1' else TmpData(23 downto 16);
  DataIn(15 downto 8) <=
    DI(15 downto 8) when BW(1) = '1' else TmpData(15 downto 8);
  DataIn(7 downto 0) <=
    DI(7 downto 0) when BW(0) = '1' else TmpData(7 downto 0);
  process( CLK )
  begin
    if( CLK'event and CLK = '1' ) then
      if( WE = '1' ) then
        MemArray(conv_integer(ADDR)) <= DataIn;
      end if;

    end if;

  end process;

  process( CLK )
  begin
    if( CLK'event and CLK = '1' ) then
      RegADDR <= ADDR;
    end if;

  end process;
```


© 2021 All Rights Reserved.

```
  DO <= MemArray(conv_integer(RegADDR));
end mem_arch;
```

Dual Port Memories with Two Clocks

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity rad_edge_sensitive is
   generic ( DataW  :   integer := 16 ;           -- word-width
             Addrw    : integer := 16 );
   port(  RD0   : out std_logic_vector(DataW - 1 downto 0);
       WD0   : in  std_logic_vector(DataW - 1 downto 0);
       RA0   : in  std_logic_vector(Addrw - 1 downto 0);
       WA0   : in  std_logic_vector(Addrw - 1 downto 0);
       WE0   : in  std_logic;
       RD1   : out std_logic_vector(DataW - 1 downto 0);
       WD1   : in  std_logic_vector(DataW - 1 downto 0);
       RA1   : in  std_logic_vector(Addrw - 1 downto 0);
       WA1   : in  std_logic_vector(Addrw - 1 downto 0);
       WE1   : in  std_logic );
END rad_edge_sensitive;

architecture ram of rad_edge_sensitive is
type mem_type is array (Addrw-1 downto 0) of std_logic_vector(DataW-1 downto 0);
signal MemArray : mem_type;
begin
   process(WA0, WE0, WD0)
   begin
     if (WE0'event and WE0 = '1') then
      MemArray(conv_integer(WA0)) <= WD0;
     end if;

   end process;

   process(WA1, WE1, WD1)
   begin
     if (WE1'event and WE1 = '1') then
       MemArray(conv_integer(WA1)) <= WD1;
     end if;

   end process;

   RD0 <= MemArray(conv_integer(RA0));
   RD1 <= MemArray(conv_integer(RA1));
end ram;
```

Synthesizing Designs with Acceleratable Testbench (ATB) Constructs

The Palladium HDL-ICE Compiler synthesizes some testbench constructs that are not normally synthesizable. These are then placed on the emulator for faster verification.

© 2021 All Rights Reserved.

Use Model for ATB

To synthesize a module with ATB constructs, import the module using `hdlImport` with the `-atb` option. The HDL-ICE Compiler will pass ATB-related data to the emulator run-time system.

An `hdlOutputFile -f qel <file>` command should be added before `hdlSynthesize` in the XEL compiler script. Alternatively, the `-outputQel <file>` option can be used with the `vaelab` command for the same purpose. This command instructs the HDL-ICE Compiler to output a file containing run-time XEL commands.

Now, compile and download the design. After downloading the design, issue the `hdlConfig` run-time XEL command immediately after the `configPM` command . This initializes registers and memories.

Example:

The following script illustrates how to compile some modules with ATB mode:

```
hdlInputFile modA.v
hdlImport -atb
hdlOutputFile -f verilog hdl_gate.v
hdlOutputFile -f qel hdl_gate.qel
hdlSynthesize top
```

The following script illustrates a typical compile and run-time script:

```
//Compile Script
==================
delimiterRule Verilog
compilerOption -add {visionMode FV}
compilerOption -add {mode ice}
refLib qtref
netlistFile {verilog hdl_gate.v}
designImport
design design_top top
emulatorConfiguration -add {host emu_host}
precompile
compile

//Run-time Script
==================
debug .
host emu_host
download
configPM -stb

hdlConfig hdl_gate.qel
run
```


© 2021 All Rights Reserved.

Using hdlConfig Command at Run Time

The `hdlConfig` command is used during run time, right after the command `configPM -stb`. There are two possible syntax variants for this command. Choose only one syntax for the `hdlConfig` command; the two syntax styles cannot be combined.

First Syntax for hdlConfig

This syntax, with `<file>`, must be invoked before the first `run` command at emulation time, and it can be invoked only once in an entire run session.

```
hdlConfig <file> [-cycleTime=<time> ] [-top=<instance>] \
          [-displayDebug][+plusargs]
```

For detailed information on the above syntax, refer to the *hdlConfig for Initializing the System Tasks* section of the *hdlConfig* command description in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

For details related to configuring `$test$plusargs` and `$value$plusargs` using the `+plusargs` option mentioned above, refer to $test$plusargs and $value$plusargs on page 276.

Second Syntax for hdlConfig

This syntax, without `<file>`, must be invoked after executing the `hdlConfig` command using the first syntax. It can be invoked at any time during a run session, and can be invoked more than once if necessary.

```
hdlConfig [-disable {<instPath> | -all | -top }]
          [-enable {<instPath> | -all | -top }]
          [-disableBuffer ] [ -enableBuffer]
          [-stopCallBack <tclProc>] [-finishCallBack <tclProc>]
          [-timeOut <cycleNum> ]
```

For detailed information on the above syntax, refer to the *hdlConfig for Enabling or Disabling the Compile Options* section of the *hdlConfig* command description in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

For details related to configuring `$stopCallBack` and `$finishCallBack` using the `-stopCallBack` and `-finishCallBack` options mentioned above, refer to $finish and $stop system tasks on page 279.

Supported Testbench Constructs

The following constructs are supported in a testbench:

© 2021 All Rights Reserved.

- Initial blocks
- Initial values
- `$qel` system task
- $readmemb and $readmemh
- $writememb and $writememh
- $display and $write
- $time System Functions used as an argument of `$display` or `$write`
- $test$plusargs and $value$plusargs
- force and release Statements
- repeat Statements
- report Statements
- $finish and $stop system tasks

$readmemb and $readmemh

Both `$readmemb` and `$readmemh` system tasks are supported to download memory content file into a memory. The HDL-ICE Compiler supports all the syntaxes in the standard:

```
$readmemb("<filename>", <memname> [ , <start_addr> [ , <finish_addr> ] ] );
$readmemh("<filename>", <memname> [,  <start_addr> [,  <finish_addr> ] ] );
```

The third and fourth arguments are optional and can be either constants or any expressions. The unknown value (x) and the high impedance value (z) in the memory content file will be converted into 0. Both system tasks can appear in either initial blocks or always blocks. However, it is not allowed to read or write the same memory after a `$readmemb` or `$readmemh` within the same clock cycle in an initial or always block. There can be multiple `$readmemb` or `$readmemh` system tasks for the same memory.

$writememb and $writememh

`$writememb` and `$writememh` dump memory contents to a file. The output files are readable by `$readmemb` and `$readmemh`.

`$writememb` and `$writememh` do not support enumeration types in SystemVerilog, or arrays with more than two dimensions.

© 2021 All Rights Reserved.

$display and $write

Note: For useful information on performance enhancements, refer to Memory Buffer for $display and $write on page 275.

The following format specifications are supported for `$display` and `$write`:

- %h
- %m
- %d
- %o
- %b
- %c
- %s
- %t
- %x

Note: `%x` is the functional equivalent of `%h`. It is not covered in IEEE standards, but is supported by popular Verilog simulators.

%m with $display

`%m` is supported in `$display` and `$write`. `%m` shows the hierarchical path of the instance where the `$display` or `$write` is located. The Xcelium simulator shows a hierarchical path starting with the top module, followed by the instances in the path. The HDL-ICE Compiler follows this format.

Example

`$display` in instance `u1.u2` under top module test:

```
$display("ERROR: assertion failed at %m.");
```

will show the following message at run time:

```
ERROR: assertion failed at test.u1.u2.
```


© 2021 All Rights Reserved.

Memory Buffer for $display and $write

The execution of the hardware emulator is suspended during the process of displaying a message. When `$display` occurs frequently, the accumulated time of small interruptions slows down emulator performance.

A memory buffer to store messages is applied to improve `$display` and `$write` performance. A message is not immediately displayed, but is stored in memory. The accumulated messages are then displayed together at a later time.

The memory buffer is not enabled by default. To enable the memory buffer to improve performance when there are frequent occurrences of `$display`, add the following option to the `hdlSynthesize` command:

```
-displayBuffer[+depth=<num>][+timeOut=<num>][+port=<num>]
```

where:

`[depth=<num>]`:

The depth of the memory for `$display` buffer. The default value is 8192 (8K). Increase the depth to improve the performance if `$display` still occurs frequently. Decrease the depth if there is a capacity limit. The value should be at most 2**30, otherwise an error message will be shown during `hdlSynthesize`.

`timeOut=<num>`:

The maximum number of emulation fast clock cycles per `$display` which can stay in the buffer. The default value is 2**28, or approximately 256 million cycles. If the fast clock is 500KHz, that means a `$display` message can be delayed up to 128 seconds. Without such a limit, when a message does not occur, you will not know whether it is buffered, or whether something is wrong to prevent it from occurring. A smaller value also helps to reduce the width of the memory buffer. Increase the value to improve performance: Decrease the value to reduce memory width. The range of the values is from 2**20 to 2**30; an error message will be shown during `hdlSynthesize` if an out-of-range value is given.

`port=<num>`:

Specifies the number of ports of the memory for `$display` buffer. The default value is `1`. If the number of simultaneous `$display` exceeds the number of ports, the messages are displayed immediately without buffering. Increase the ports to reduce performance overhead caused by simultaneous `$display`. Increasing the ports can increase emulation capacity or step count. Turn on the `-displayDebug` option for the `hdlConfig` command to determine how many memory ports are optimal.

Typically, you do not need to change the default value. Type:

© 2021 All Rights Reserved.

```
hdlSynthesize -displayBuffer ...
```

The action of dumping the contents of the memory buffer to displayed messages is referred to as a flush. A flush will happen when:

- The memory buffer is almost full; that is, the number of messages stored is close to the depth of the memory.
- The time out limit is reached, as calculated by the number of fast clock cycles from the time the first message is stored in the memory.
- The `run <cycle>` command finishes with the specified number of cycles.
- A user-specified trigger stops the emulator.
- An emulator stops because of `$finish` or `$stop`.
- A software error is detected.
- The number of instances of `$display` occurring at the same time exceeds the number of memory buffer ports.

$time System Function

By default, the HDL-ICE Compiler ignores the `$time` system function in a way such that timing constraints can be satisfied. For instance:

```
if (delay < $time)
```

is always evaluated to TRUE while

```
if (delay > $time)
```

is always evaluated to FALSE.

When the `-atb` option is specified, `$time` will still be ignored as usual, unless it appears in an argument of `$display` or `$write`. The value of `$time` will be shown by `$display` or `$write`. The precision of `$time` is at most the cycle time of the fast clock cycle, so it is possible that there might be a minor difference between emulator `$time` and Xcelium `$time`. `$time` cannot combined in an expression, as for example, `$time + 2`, as an argument in `$display` or `$write`. Otherwise, an error message will be shown during `hdlImport`.

$test$plusargs and $value$plusargs

Note: When using this argument, you must specify `-keepSourceless` when doing precompile, otherwise the ATB designs with `$value$plusargs` will potentially generate an error during `hdlConfig`.

© 2021 All Rights Reserved.

`$test$plusargs` existed in the Verilog-1995 standard, and `$value$plusargs` was added into the Verilog-2001 standard. Both system functions provide capabilities similar to `define` and `ifdef` to control the conditional execution of Verilog code. However, to change the values for `define` and `ifdef`, you formerly needed to recompile your design, whereas for `$test$plusargs` and `$value$plusargs`, you are able to change the values to control the conditions during run time without recompiling the design.

When the `-atb` option is executed with the `hdlImport` command, both `$test$plusargs` and `$value$plusargs` are supported together with other system tasks. `$test$plusargs` and `$value$plusargs` can appear in any synthesizable expressions as well as any initial or always block. Typically, these two system functions appear in `if` expressions like:

```
if ($test$plusargs("debug_on"))
    debug_mode = 1;
else
    debug_mode = 0;
if ($value$plusargs("maxCycle=%d", maxc))
else maxc = 500;
```

To change the argument value during run time, add `+<arg>` to the `hdlConfig` command in the same manner as you would specify such arguments for Xcelium. The syntax of the `hdlConfig` command becomes:

```
hdlConfig <file> [ -cycleTime=%d][ -top=%s] [ +plusargs ]...
```

For example:

```
hdlConfig tb_init.qel +debug_on + maxCycle=500
```

The implementation of `$test$plusargs` and `$value$plusargs` introduces no performance overhead at run time.

force and release Statements

Both `force` and `release` statements are invoked when the `-atb` option is used with the `hdlImport` command. Both statements can appear in either initial blocks or always blocks.

When the value being forced is not a constant, the behavior of Verilog `force` is illustrated by this example:

```
force a = b;
```

When Verilog `force` is executed, the value of signal `a` will change whenever the value of signal `b` changes. In this case, the HDL-ICE Compiler follows the Verilog standard in implementing Verilog `force` statements.

When two `force` statements on the same signal are activated at the same simulation time sequentially, that is, in the same initial or always block, the first `force` statement is disabled.

© 2021 All Rights Reserved.

Similarly, when a `force` statement and a `release` statement are activated sequentially at the same time, the first statement is disabled.

When a `force` statement and a `release` statement are activated at the same time concurrently, that is, in different initial or always blocks, the `force` statement takes precedence, and the `release` statement is ignored.

When two `force` statements on the same signal are activated at the same time concurrently, the order cannot be predicted.

The implementation of `force` and `release` causes no performance overhead during run time.

repeat Statements

Verilog `repeat` statements are not considered to be synthesizable by industry standard. However, the HDL-ICE Compiler synthesizes `repeat` statements when the repeat count can be calculated into a constant during compile time. This capability is enabled by default, whether or not the `-atb` option is used. An example using a `repeat` statement is:

```
repeat (10) @(posedge clk);
```

In this case, the HDL-ICE Compiler waits for 10 clock cycles before the next `repeat` statement starts.

There is no performance overhead for `repeat` statements.

report Statements

By default, the HDL-ICE Compiler ignores a `report` statement. To enable support for such statements, you need to specify the `-atb` option with either `vavhdl` or `hdlImport` command.

Note: A `report` statement used with an `assert` statement is ignored.

A VHDL `report` statement requires an expression of string type. You can use `image` VHDL predefined attribute to convert a variable value into a string and concatenate it to other strings using the `&` operator as illustrated in the following example:

```
report "counter = " & integer'image(counter);
```

You can also append a severity (`NOTE`, `WARNING`, `ERROR`, `FAILURE`) to a `report` statement as shown below:

```
report "counter = " & integer'image(counter) severity WARNING ;
```


© 2021 All Rights Reserved.

A report message starts with the word `REPORT`, followed by severity, the fast cycle number of the Palladium Z1 run (like `$time` system function does when specified in a Verilog `$display` statement), the instance path (like `%m` returns when specified in a Verilog `$display` statement), and the value of the string expression in the VHDL `report` statement.

For example:

```
REPORT/NOTE (cycle: 100) from test.u1.u2: counter = 32
```

In the report message illustrated above, `counter = 32` is the string expression from the example `report` statement shown before it, that is `"counter = " & integer'image(counter).`

The report messages are buffered like Verilog `$display` statements when the `-displayBuffer` option is specified with the `hdlConfig` command to maximize Palladium Z1 performance.

$finish and $stop system tasks

Both `$finish` and `$stop` system tasks are supported. They can be used in either initial blocks or always blocks. According to the Verilog standard, `$finish` and `$stop` take three possible values: 0, 1, or 2; if no argument is given, the default value is 1. The HDL-ICE Compiler treats all non-zero parameter values as 1 for `$finish` and `$stop`. The HDL-ICE Compiler follows the format of Verilog-XL for `$finish`.

Callback Functions for $finish and $stop

`$finish`

> `$finish` is useful for pausing an emulation run to conduct offline debugging. Using `$finish` prevents the next test result from over-writing the current emulation snapshot. For example, when an error occurs, you might want to upload the waveform, or continue to run 1000 cycles to collect additional debug information. You can institute a callback function to execute these actions automatically.

`$stop`

> When `$stop` causes simulation interruptions, you have the option of executing debugging commands or continuing the simulation run. A callback function automatically executed after `$stop` will enable you to execute XEL commands whose functionality cannot be supported through Verilog code.

At run time, the callback functions for `$finish` and `$stop` can be specified using the following options of the `hdlConfig` command:

© 2021 All Rights Reserved.

```
-finishCallBack <tclProc>
-stopCallBack <tclProc>
```

<tclProc> is a user-defined TCL procedure. If the TCL procedure accepts at least one argument, the instance path where `$stop` or `$finish` occurs will pass as the first argument to the callback function. If the TCL procedure accepts at least two arguments, the RTL module name where `$stop` or `$finish` occurs will pass as the second argument. You can use this information to identify where the `$stop` or `$finish` occurs in the callback function.

An example callback function is shown below:

```
hdlConfig tb_init.qel
proc myCallback { inst cell } {
  puts "Finish at instance $inst of cell $cell."
}
hdlConfig -finishCallBack myCallback
run 200000
```

For the usage details and syntax of the `hdlConfig` command, refer to the *hdlConfig* command description in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Synthesizing Initial Blocks with Synthesizable Constructs

By default, the HDL-ICE Compiler ignores initial blocks and their contents, and provides warning messages. Using the `-atb` option, the HDL-ICE Compiler synthesizes initial blocks with synthesizable constructs. If the statements in an `initial` block are synthesizable when they appear in an `always` block, the `initial` block is synthesizable; otherwise the HDL-ICE Compiler reports error messages for those non-synthesizable constructs in the `initial` blocks.

The HDL-ICE Compiler utilizes its capability for Verilog-2001 constant functions to evaluate initial blocks and statically calculate initial values during compile time. Consider the following example:

```
module test (clk, d, p);
   input clk;
   input [0:98] d;
   output [2:100] p;
   reg [2:100] p;
   integer i;
   function isPrime;
     input [6:0] n;
     integer i;
     begin
   isPrime = 1;
   begin: block
     for (i = 2; i < n; i = i + 1) begin
       if (i * i > n)
         disable block;
       if (n % i == 0) begin
```


© 2021 All Rights Reserved.

```
          isPrime = 0;
          disable block;
        end

      end

    end

end

      endfunction
    initial
    for (i = 2; i <= 100; i = i + 1)
      p[i] = isPrime(i);
    always @(posedge clk)
      p <= d;
endmodule
```

Here, all the function calls, for-loops, if-statements, and intermediate variables used only in the initial block will disappear in the synthesized netlist, and either 0 or 1 will be assigned an initial value to each of p[2] to p[100]. When the initial block is ignored, signal “p” will be synthesized into instances of Q_FDP0 cell. With initial block synthesis, a Q_FDP0I1 cell will drive a net with initial value 1 and a Q_FDP0I0 will drive a signal with initial value 0. For the above example, the following instances can be found in the synthesized netlist.

```
Q_FDP0I1 U89 ( .CK(clk), .D(d[9]), .Q(p[11]), .QN( ));
Q_FDP0I0 U91 ( .CK(clk), .D(d[7]), .Q(p[9]), .QN( ));
```

Each net driven by cell Q_FDP0I1 will be appended to a list in the output of the `hdlOutputFile -f` XEL command, and all nets in the list will be set to 1 by the `signal -set` command when `hdlConfig` is invoked during run time. Because the initial values of flip-flops are 0’s, there is no action required during the run time for Q_FDP0I0 cell. The HDL-ICE Compiler also supports Verilog-2001 initial values, such as:

```
reg q = 1;
```

Initial value support relies on the `signal -set` command to set the value of a net which is forcible. Consider the following example:

```
module test (clk, d1, en, a, z);
input clk, d1, d2, en;
input [2:0] a;
output [2:0] z;
eg [2:0] z;
initial z = 3’b10z;
always @(posedge clk) begin
  if (en)
    z = a;
  else
    z = 3’bzzz;
endmodule
```

In the above example, signal `z` will be synthesized into tristate buffers. A tristate buffer output is not directly forcible, but the HDL-ICE Compiler can initialize its value to 1 by setting both the enable pin and the data pin of the tristate buffer to 1, or initialize it to Z by setting its enable

© 2021 All Rights Reserved.

pin to 0. The following VHDL example demonstrates the same technique to initialize a signal `q` which is a wired OR of 2 flip-flops.

```
package pack_res_func1 is function wired_or ( input : bit_vector ) return bit
; end ;

 package body pack_res_func1 is
    function wired_or ( input : bit_vector ) return bit is
    -- pragma resolution_method wired_or
    begin
      for i in input'range loop
        if ( input (i) ='1' ) then return '1' ; end if;
      end loop ;
      return '0' ;
    end ;

end ;

use work.pack_res_func1.all;
entity test is port ( clk1, clk2, d1, d2 : bit; res_sig : out bit); end ;

architecture rtl of test is
signal q :  wired_or bit := '1';
begin
  process begin
    wait until clk1 = '1';
    q <= d1;
end process;

  process begin
    wait until clk2 = '1';
    q <= d2;
  end process;

  res_sig <= q;
end;
```

By default, the HDL-ICE Compiler optimizes out flip-flops whose inputs are constant, and preserves all the flip-flops when the `-keepAllFlipFlop` option is used. With the `-atb` option, a flip-flop with a constant input will be optimized only when there is no initial value for it or the initial value is the same as the constant input value. Consider the following example:

```
module test (clk, q1, q2);
input clk;
output q1, q2;
reg q1 = 1;
reg q2 = 0;
always @(posedge clk) begin
  q1 = 0;
  q2 = 0;
end

endmodule
```

For the above example, neither the `-atb` nor the `-keepAllFlipFlop` option, the synthesized netlist will have both q1 and q2 driven by ground. With only the `-atb` option used, q1 will be driven by a Q_FDP0I1, while q2 will be driven by ground. Using only the `-keepAllFlipFlop` option, both q1 and q2 will be driven by Q_FDP0 cells. Using both the `-atb` option and the `-keepAllFlipFlop` option, q1 will be driven by a Q_FDP0I1 and q2

© 2021 All Rights Reserved.

will be driven by a Q_FDP0I0. With the `-atb` option, the HDL-ICE Compiler can selectively disable flip-flop optimization and reduce the overhead of `-keepAllFlipFlop`.

When there are multiple `@(posedge clk)` in an initial block, the HDL-ICE Compiler will synthesize it as an implicit finite-state machine. For an implicit finite-state machine in an always block, the last state will go back to the first state. For an implicit finite-state machine in an initial block, the last state will go back to itself.

Initializing Registers and Memories

The following constructs are supported:

- Assignment in Verilog initial blocks.

```
reg q;
initial q = 1;
```

- Declaration assignment in Verilog -2001.

```
reg q = 1;
reg [0:4] q = ’b1101;
```

- Declaration assignment in VHDL

```
signal q: std_logic_vector(3 downto 0) := "1001";
```

- VHDL initial process: a process with no sensitivity list and with a `WAIT` statement as the last statement.

```
signal q : std_logic_vector(3 downto 0);
process begin
     q <= "1001";
     WAIT;
end
```

- All HDL synthesizable constructs, including for-loop, while-loop, if-then-else, functional calls, and all expressions are allowed in a Verilog initial block or a VHDL initial process when applicable.

Example

The following Verilog example demonstrates how `p[i]` is initialized to `1` when `i` is a prime number.

```
reg [2:100] p;
integer i;
function isPrime;
nput [6:0] n;
integer i;
begin
     isPrime = 1;
     begin: block
```


© 2021 All Rights Reserved.

```
        for (i = 2; i < n; i = i + 1) begin
            if (i * i > n)
               disable block;
            if (n % i == 0) begin
               isPrime = 0;
               disable block;
            end
        end
      end
   end
endfunction
initial
   for (i = 2; i <= 100; i = i + 1)
      p[i] = isPrime(i);
```

Support for LA Mode

If you import a design with the `-atb` option and run in the STB mode, all the ATB features are supported. If, however, you run in the LA mode, only the following ATB features are supported:

- Verilog `force` and `release` statements
- `$readmemh`, `$display`, and other system tasks in initial blocks at time 0 before executing the `run` command are supported. The implementation uses XEL `memory` and `value` commands respectively.

Note: `$readmemh`, `$display`, and other system tasks in an always block, or after time 0, or after `run` are not supported in LA mode.

Executing Distributed Synthesis during RTL Synthesis

The distributed synthesis feature of the HDL-ICE Compiler is useful for synthesizing VHDL or Verilog RTL designs containing tens of million gates. The feature enables distribution of the synthesis process into tasks at module level. The tasks are dynamically allocated during run time to balance the load across multiple workstations or processes. This feature helps to increase the run-time speed during the RTL design synthesis.

Distributing the Synthesis Process

Distributed synthesis can be performed using either of the following three ways:

- Using parallel.host File
- Using Multiple CPUs of the Same Workstation

© 2021 All Rights Reserved.

- Using Load Sharing Facility (LSF) Queue
- Using Network Computer (NC) for Parallel Synthesis

Using parallel.host File

Distributed synthesis can be performed by using the workstations listed in the `parallel.host` file, which is located in the design directory. Each line of this file contains the name of a workstation. The workstations can be of mixed platforms, for example Red Hat EL 4.0 running on an AMD Opteron workstation or SuSE Linux Enterprise Server 11 running on an Intel Xeon-EM64T workstation.

Execute either of the following commands to run distributed synthesis using the `parallel.host` file:

- `vaelab -parallelHost <top_module>`
- `hdlSynthesize -parallel <top_module>`
- `vaelab -parallelHostSSH <top_module>`
- `hdlSynthesize -parallelHostSSH <top_module>`

  Note: With `-parallelHostSSH` option, HDL-ICE Compiler uses SSH to distribute synthesis over hosts listed in `parallel.host` file.

Hosts and job numbers can be specified in `parallel.host` file as shown in the following example:

```
HostA -jobs=20
HostB -jobs=30
```

Here, 20 HDL-ICE Compiler processes are specified to start on HostA and 30 HDL-ICE Compiler processes are specified to start on HostB.

The maximum number of total parallel HDL-ICE Compiler processes on all hosts is set to 248.

For using `-parallelHostSSH` option, ensure all hosts listed in `parallel.host` file is accessible through `SSH` command in user environment and there is no need to specify username, password, and SSH key again.

Tip

The `vaelab` command can be executed either from the UNIX or xeCompile command prompt. However, the `hdlSynthesize` command can be executed only from the xeCompile command prompt.

© 2021 All Rights Reserved.

The tasks are distributed across all workstations on the network. Each workstation checks out the licenses. For example, if you have four licenses installed and ten workstations are listed in `parallel.host`, the distributed synthesis feature of the HDL-ICE Compiler will select only the first four workstations specified in the `parallel.host` file to perform synthesis.

Warning messages are generated in the `tmp/hdlice.server.log` file if:

- The `parallel.host` file is empty:

  ```
  WARNING: parallel.host is empty.
  ```

- The `parallel.host` file lists hosts that cannot be connected.

  ```
  WARNING: ignore host <xxx> that can not be connected.
  ```

- The `-parallelHostSSH` option of the `vaelab` command is used and the host specified in the `parallel.host` file cannot be accessed with SSH:

  ```
  WARNING: ignore host <xxx> that can not be accessed with ssh.
  ```

Using Multiple CPUs of the Same Workstation

If a particular workstation has multiple CPUs, several tasks can be distributed to run in parallel on those CPUs.

To distribute a definite number of tasks on the current workstation, use either of the following two commands:

```
vaelab -parallel <number> <top_module>
hdlSynthesize -parallel<number> <top_module>
```

The maximum *<number>* of parallel HDL-ICE Compiler processes on the current workstation has increased from 64 to 248.

Note: In the `hdlSynthesize` command syntax given above, the `-parallel` and *<number>* arguments should be specified together without any space in between.

Using Load Sharing Facility (LSF) Queue

Distributed synthesis processes can also be submitted to an LSF queue using either of the following commands:

- `vaelab -parallelLSFSameHost <number> <top_module>`
- `hdlSynthesize -parallelLSFSameHost<number> <top_module>`
- `vaelab -parallelLSF <number> <top_module>`
- `hdlSynthesize -parallelLSF<number> <top_module>`

© 2021 All Rights Reserved.

Note: In the `hdlSynthesize` command syntax given above, the `-parallelLSFSameHost` or `-parallelLSF` and `<number>` arguments should be specified together without any space in between. The `<number>` is a required argument that should be greater than zero. It is used to specify the number of jobs to submit to the LSF queue.

The maximum `<number>` of parallel HDL-ICE Compiler processes that can be passed as an argument is 248.

Use `-parallelLSFSameHost` option to assign one workstation for HDL-ICE Compiler and avoid network latency between multi-workstations during parallel synthesis,

To use the LSF queue for distributed synthesis, ensure that the LSF environment has already been set up such that the `bsub` executable is available in the `PATH` UNIX environment variable. If `bsub` is not available, the HDL-ICE Compiler will exit the process with an error message.

Using LSF_options File

Options to the bsub executable can be specified in a file called `LSF_options`. See LSF_options File for information on the contents of the `LSF_options` file.

If a file named `LSF_options` exists under the current working directory, the string will be appended to the `bsub` command when a job is submitted to an LSF queue. All the options must be defined in one line. The HDL-ICE Compiler does not check for syntax of these options.

The `LSF_options` file might exist in a location other than the current working directory. Define the full path to the `LSF_options` file using the `CDS_LSF_OPTIONS` environment variable. When you execute the `hdlSynthesize` or `vaelab` command, the current working directory is searched for the `LSF_options` file. If the file is not found there, the full path specified with the `CDS_LSF_OPTIONS` environment variable is opened as an `LSF_options` file.

Using Network Computer (NC) for Parallel Synthesis

Some use NC instead of LSF as NC is a competing tool to LSF.

Parallel synthesis processes can also be submitted to NC using either of the following two commands:

```
vaelab -parallelNCSameHost <number> <top_module>
hdlSynthesize -parallelNCSameHost<number> <top_module>
```


© 2021 All Rights Reserved.

Use `-parallelNCSameHost` option to assign one workstation to run parallel synthesis. Also note, the maximum *<number>* of parallel HDL-ICE Compiler processes that can be passed as an argument is 248.

Scheduling the Tasks

The tasks are scheduled based on the dependency among the modules. For example, a parameterized Verilog module, or a VHDL entity with generic values, will not be synthesized until the upper-level modules are synthesized.

A single netlist is generated by automatically collecting results from all workstations. The final synthesis result is the same as that synthesized by a single workstation. The HDL-ICE Compiler occasionally can produce slightly different gate counts on different machines due to differing optimization ordering.

Limitations of Distributed Synthesis

The distributed synthesis feature of the HDL-ICE Compiler does not support the following:

- Use of the `-atb` option with the `-displayBuffer` option of either the `hdlSynthesize` or `vaelab` command.
- Signals or variables in packages
- SystemVerilog compilation unit scope

Using Conformal to Verify HDL-ICE Compiler-Generated Netlist

After synthesizing an RTL design, the HDL-ICE Compiler generates a netlist that can be used as an input for the emulation process. Before using this netlist, you can use Conformal to check the correctness of synthesis. This helps in detecting synthesis problems at an early stage of the verification process with a minimum of effort.

Note:

- To use Conformal, you must purchase its licenses from Cadence. Refer to the *Conformal Equivalence Checking User Guide* for installation and platform support of Conformal software.
- Palladium Z1 execution should be the primary methodology to validate the correctness of the RTL mapping process. Occasionally, a netlist is functionally

© 2021 All Rights Reserved.

correct on Palladium Z1 with respect to the RTL specification, but Conformal does not conclude its equivalence to the RTL design because of capacity limitations, different interpretation between ASIC implementation and/or verification models. Conformal can be a valuable assistant in diagnosing potential issues and problems only when Palladium Z1 fails to perform the expected RTL functionality.

To improve the accuracy of an equivalency check, if there are plans to use Conformal, ensure that the product uses the same optimizations and interpretations of the RTL as Palladium Z1. Cadence suggests that you consult the Conformal User Guide before attempting to verify the netlist.

The process requires you to perform the steps discussed in the following sections:

- Generating Script for Conformal
- Starting Conformal
- Analyzing Results Using Conformal

Generating Script for Conformal

While synthesizing the design using the HDL-ICE Compiler, generate a script for Conformal by giving the following `hdlOutputFile` XEL command:

```
hdlOutputFile -add -f lec <lec_script_file_name>
hdlOutputFile -add -f verilog <verilog_netlist_name>
```

These commands should be used before the `hdlSynthesize` XEL command.

The `-f lec` option specifies the creation of a single script file which contains the Conformal commands necessary to perform a hierarchical comparison between RTL and HDL-ICE Compiler output Verilog netlists. A hierarchical comparison is useful because:

- It reduces the complexity of equivalent checking and speeds up the verification process.
- When Conformal reports non-equivalent items, it is easier to identify the module that is non-equivalent.

Note: The script collects information from all source files used by all synthesized modules. This process is supported for both sequential and distributed synthesis.

Example: Conformal Script Generated by hdlOutputFile

```
set log file lec.do.log -replace -nobackup
set dofile abort exit
set compare effort auto
```


© 2021 All Rights Reserved.

```
set compare option -allgenlatch
set undefined cell black_box -both -noascend
set naming rule "_" -parameter -golden
set naming rule "." "" -field_delimiter -golden
set naming rule "%_REG" -register -golden
set mapping method -nets -nobbox_name_match
set flatten model -latch_transparent
set datapath option -auto
read library -verilog -both $XEHOME/share/vxe/etc/hdlice_ref.v
read design -vhdl -golden tri.vhd -noelaborate
read design -verilog -golden -define CDNS_DW_LEC test.v -root test
write design lec.do.bbox -golden -bbox_only -replace
read design -revised -verilog -root test lec.do.bbox test.vs
write hier_compare dofile lec.do.hier -replace -threshold 0
dofile lec.do.hier
exit -force
```

Starting Conformal

Conformal can be started from the design directory with the following command:

```
lec [-XL ][-nogui ]-dofile <lec_script_file_name>
```

Analyzing Results Using Conformal

The following sections discuss the considerations for analyzing HDL-ICE Compiler results using Conformal:

Improving Comparison Accuracy

The HDL-ICE Compiler optimizes out certain registers during synthesis. Therefore, Conformal might not be able to map all the key points and it might subsequently report false non-equivalence. To avoid this, Cadence recommends that the `-keepAllFlipFlop` option be specified with the `hdlSynthesize` command. This option is automatically turned on and a warning message is issued if this option is not specified with the `-f lec` option. This option will slightly increase the gate count of the HDL output netlist.

To improve the Conformal mapping process and the accuracy of comparison, Cadence recommends turning on the `-keepRtlSymbol` option. This instructs the HDL-ICE Compiler to use Conformal-friendly names when it writes output Verilog netlists. This option is automatically turned on if the Conformal script generation option is present and a warning message displays.

© 2021 All Rights Reserved.

Comparing Output with Memory Instances

When there are memories inferred from the RTL arrays, xeCompile uses VXE memory primitives (MPR/MPW) which cannot be understood by Conformal. Even with the simulation model that xeCompile created in the netlist, Conformal either takes a long time for large memories to finish or reports that modules are non-equivalent in many instances.

The generated script includes the Conformal commands to exclude all the modules inferring VXE memory primitives from the comparison:

```
add notranslate modules -design -golden <RTL_memory_module_name>
add notranslate modules -design -revised <Netlist_memory_module_name>
```

Comparing Output with Operators (Adder/Subtractor/Multiplier/Divider)

When synthesizing a design with above operators, Conformal performs better (in terms of accuracy and speed) when you use the following command option to invoke Conformal, if there is a Conformal Ultra license:

```
lec -XL
```

Without a Conformal Ultra license, the above command has no effect except to issue a warning. Cadence recommends that you acquire the license if the design uses these operators.

Omitting Skipping Modules from Equivalence Comparison

To skip certain modules for equivalence comparison (to avoid long verification times, and so on), specify the module names in a text file named <*lec_script_file_name*>.moduleSkipEC. Each module name should be on a separate line. The Conformal script created by HDL-ICE Compiler reads this file, and applies it to the modules to be skipped.

Customizing LEC Script File Using a Configuration File

Customize the default LEC script generated by HDL-ICE Compiler by including a set of custom commands in a configuration file. The configuration file should be a text file with the name <*lec_script_file_name*>.config, and it should be located in the same directory as the original LEC script file.

In this file, a set of LEC commands can be included, with each command beginning on a new line. The original LEC script generated by HDL-ICE Compiler sources this configuration file, if it exists in the same directory containing the original LEC script.

© 2021 All Rights Reserved.

For example this configuration file can be used to set the multi-threading feature in LEC, which is not set by default in the LEC script.

```
set parallel option -threads 4
```

The above command in the configuration file sets the maximum number of threads that you want to use to `4`.

Conformal Log Files

Conformal generates the following files in the directory in which it is executed:

- Log file (`<lec_script_file_name>.log`)
- Intermediate do file
- File of blackbox definitions

These files are not removed after Conformal finishes. So, you can check the results and investigate further, if required.

Using UNIX Commands for RTL Import and Synthesis

The UNIX command-line interface supports a Library Mapping File called `va.lib` and three executable, `vavhdl`, `vavlog`, and `vaelab`. These provide a way to import and synthesize an RTL design without using XEL commands.

These executables are designed to provide UNIX command-line interface in syntax similar to Xcelium. For syntax and usage details of the three executables, refer to:

- vavhdl
- vavlog
- vaelab

Library Mapping File

A library mapping file is used to provide a mapping between the logical library names and the physical directory paths where those libraries are stored. This file should be saved with the name `va.lib`. Add the details to the `va.lib` file in the following format:

```
DEFINE <logical_lib_name> <physical_lib_path>
DEFINE WORK [<logical_lib_name> | <physical_lib_path>]
```

For example:

© 2021 All Rights Reserved.

```
DEFINE siliconLib /nas/caenac/group/testcase/lib1
DEFINE rtlLib $TESTHOME/rtlLib
DEFINE cpulib ../rtlLibs/cpulib
DEFINE WORK cpulib
```

In the `va.lib` file, multiple logical library names can be mapped to the same physical location as illustrated in the following example:

```
DEFINE libA ./cpu_lib
DEFINE libB ./cpu_lib
DEFINE libC ./cpu_lib
DEFINE WORK ./cpulib
```

Note:

- The logical library names and the `DEFINE` command are case-insensitive.
- If the `hdlImport` and `hdlSynthesize` XEL commands are used, the library mapping is specified using the `hdlDefineLib` XEL command, which is similar to a `DEFINE` command in the `va.lib` file.

Setting Variables in va.lib Files

The `set` command can be used to set the local variables. Use of `$var` or `${var}` indicates variable substitution. For example:

```
set libpath nas/caenac/group/testcase/allLibs
define cpuLib $libpath/cpuLib
```

There are three predefined variables: `vavlogopts`, `vavhdlopts`, and `vaelabopts`. The value of `vavlogopts` is appended to all vavlog commands. Similarly, the values of `vavhdlopts` and `vaelabopts` are appended to vavhdl and vaelab commands, respectively. For example:

```
set reflibs -reflib "libA libB libC"
set worklib -work work2
set vavlogopts $reflib $WORKLIB
set vaelabopts $WORKLIB
```

Note: If the same predefined variable is set more than once (as shown in the above example), only the last setting takes effect.

Environment variables can also be used to specify the physical library paths, such as `$TESTHOME/rtlLib`. The following search sequence is followed:

- First, the variables already defined in the `va.lib` file are searched.
- Then, the environment variables are searched.

Note: Variable names defined in the `va.lib` file are case-insensitive, but environment variables are case-sensitive.

© 2021 All Rights Reserved.

Additional Statements Supported in va.lib Files

In the `va.lib` file, the following statements can also be given:

- `UNDEFINE <logical_library_name>`

  Undefines the logical library name. No error is issued if the logical library is not previously defined.

  For example:

  ```
  UNDEFINE cpulib
  ```

- `INCLUDE <path>`

  Include the content of the file specified. Issue an error if the file is not found.

  For example:

  ```
  INCLUDE /global/group/va.lib
  ```

- `SOFTINCLUDE <path>`

  Include the content of the file specified. No error is issued if the file is not found.

  For example:

  ```
  SOFTINCLUDE /global/group/va.lib
  ```

Other Syntax Rules Supported in va.lib Files

The following syntax rules are also supported in a `va.lib` file:

- Use a backslash (`\`) to extend a statement to the next line.
- Use a pound sign (`#`) or double hyphens (`--`) to begin a comment.

  For example:

  ```
  # This is a comment.
  Define cpuLib ./myCPU    -- This is a comment.
  ```

Specifying va.lib File to Palladium Z1

By default, the vavhdl, vavlog, or vaelab executable looks for the va.lib file under the current directory. The `va.lib` file can also be specified using the `-valib` option on the UNIX command line.

For example:

```
vavlog -valib /project/va.lib.test file.v
```


© 2021 All Rights Reserved.

If the `-valib` option is not specified and there is no `va.lib` file under the current directory, `$HOME/va.lib` is used if the `HOME` environment variable is defined and the `va.lib` file exists under the specified directory.

If the `va.lib file` is not found anywhere, the following is assumed:

```
DEFINE WORK ./work
```

This means that the default library is called `work` and is placed in the `./work` directory.

vavhdl

Imports the VHDL source files. Though giving one `vavhdl` command per file is supported, Cadence recommends importing of all VHDL source files together using a single `vavhdl` command, if possible, as illustrated below:

```
vavhdl -work worklib file1.vhd file2. vhd file3.vhd file4.vhd
```

Invoking the `vavhdl` command several times to run `vavhdl` once on each VHDL file slows down the import time and reduces the reliability of some features, such as hierarchical net reference support.

The syntax for this command is:

```
vavhdl [options] <source_file>
```

The `[options]` parameter in the above syntax can have the following values:

-append_log

-atb

-db2

-ErrorMax <arg>

-file <arg>

-full

-help

-Ignore_Pragma <arg>

-lock

-logFile <log_file>

-refLib <arg>

-rm_Synth_Prefix <arg>

-synth_Prefix <arg>

-vaLib <arg>

© 2021 All Rights Reserved.

-V2008

-V1987

-work <arg>

Example

```
vavhdl -work lib2 file3.vhd file4.vhd
```

The above command imports VHDL files `file3.vhd` and `file4.vhd` into library `lib2`.

The following table describes the options of the `vavhdl` command:

vavhdl Command Option	Option Description
`-append_log`	Appends log to an existing log file. When `-logFile` is not present, it appends to `vavhdl.log`. When `-logFile` *<log_file>* is present, it appends to the specified *<log_file>*.
`-atb`	For a list of supported constructs with the *-atb* option refer to Synthesizing Designs with Acceleratable Testbench (ATB) Constructs.
`-db2`	Changes the database format so the sub-directories become files.
`-ErrorMax` *<arg>*	Specifies the maximum number of errors processed.
`-file` *<arg>*	Specifies a file for additional command-line options. Use of `//` comments out the rest of the line in the file.
`-full`	Imports all files even if not edited since last `vavhdl` command.
`-help`	Prints the usage information.
`-Ignore_Pragma` *<arg>*	Ignores the pragmas that exist in the specified modules. Multiple modules can be specified with `+` (plus sign) used between each module name for concatenation.
`-lock`	Turns on library locking for concurrent import sessions.
`-logFile` *<log_file>*	Specifies a log file name. If not specified, `vavhdl.log` is used. This command is comparable to the `hdlLogFile` *<log_file>* XEL command. This overwrites the log file if exists; otherwise, creates it.

© 2021 All Rights Reserved.

vavhdl Command Option	Option Description
`-refLib <arg>`	Specifies the library in which to search for undefined cells. This is comparable to XEL command `hdlImport -l`.
`-rm_Synth_Prefix <arg>`	Removes the specified pragma for the synthesis directives from the default pragma list. The default pragmas are `pragma`, `synopsys` and `quickturn`.
`-synth_Prefix <arg>`	Appends additional pragmas for the synthesis directives to the default pragma list.
`-vaLib <arg>`	Specifies the `va.lib` file to be used. If not specified, the program searches under the current working directory; then, searches under the home directory.
`-V2008`	Required to import VHDL design code that includes VHDL 2008 constructs.
`-V1987`	Turns off VHDL-1993 keywords and warnings.
`-work <arg>`	Specifies the current working library, `WORK`. If not specified, use `WORK` as defined in `va.lib`. If `WORK` is not defined in `va.lib` either, use `work`.

vavlog

Imports the Verilog source files. This utility is comparable to the `hdlImport -2001` XEL command. Though giving one `vavlog` command per file is supported, Cadence recommends all of the Verilog source files should be imported together using a single `vavlog` command, if possible, as illustrated below:

```
vavlog -work worklib top.v
```

Invoking the `vavlog` command several times to run `vavlog` once on each Verilog file slows down the import time and reduces the reliability of some features, such as hierarchical net reference support.

The syntax for this command is:

```
vavlog [options] <source_file>
```

The `[options]` parameter in the above syntax can have the following values:

`-append_Log`

`-atb`

© 2021 All Rights Reserved.

-db2

-define <arg>

-ErrorMax <arg>

-file <arg>

-full

-help

-incDir <path>

-libDir <directory>

-libExt <ext>

-libFile <filename>

-logFile <log_file>

-refLib <arg>

-rm_Synth_Prefix <arg>

-sv

-synth_Prefix <arg>

-upCase

-v1995

-vaLib <arg>

-work <arg>

Example

```
vavlog -work lib1 file1.v file2.v
```

The above command imports Verilog files `file1.v` and `file2.v` into library `lib1`.

The following table describes the options of the `vavlog` command.

vavlog Command Option	Option Description
`-append_Log`	Appends log to an existing log file. When `-logFile` is not present, it appends to `vavlog.log`. When `-logFile` *<log_file>* is present, it appends to *<log_file>*.

© 2021 All Rights Reserved.

vavlog Command Option	Option Description
`-atb`	For a list of supported constructs with the *-atb* option refer to Synthesizing Designs with Acceleratable Testbench (ATB) Constructs.
`-db2`	Changes the database format so the sub-directories become files.
`-define <arg>`	Defines a macro. This is comparable to the `hdlInputFile +define+<var1>` XEL command.
`-ErrorMax <arg>`	Specifies the maximum number of errors processed.
`-file <arg>`	Specifies a file for additional command-line options. Use of `//` comments out the rest of the line in the file.
`-full`	Imports all files even if not edited since last `vavlog` command.
`-help`	Prints the usage information.
`-incDir <path>`	Specifies path to an included directory. This is comparable to the `hdlInputFile +incdir+<path>` XEL command.
`-libDir <directory>`	Specifies a library directory. This is comparable to the `hdlInputFile -y <directory>` XEL command.
`-libExt <ext>`	Specifies the file extensions to be used when searching library directories. This is comparable to the `hdlInputFile +libext+<ext>` XEL command.
`-libFile <filename>`	Specifies a library file. This is comparable to `hdlInputFile -v <filename>` XEL command.
`-logFile <log_file>`	Specifies a log file name. If not specified, `vavlog.log` is used. This command is comparable to `hdlLogFile <log_file>` XEL command. This overwrites the log file, if it exists; otherwise, creates it.
`-refLib <arg>`	Specifies the library in which to search for undefined cells. This is comparable to the `hdlImport -l` XEL command.
`-rm_Synth_Prefix <arg>`	Removes the specified pragma for the synthesis directives from the default pragma list. The default pragmas are `pragma`, `synopsys`, and `quickturn`.
`-sv`	Turns on the SystemVerilog support.

© 2021 All Rights Reserved.

vavlog Command Option	Option Description
`-synth_Prefix <arg>`	Appends additional pragmas for the synthesis directives to the default pragma list.
`-upCase`	Changes all verilog names to uppercase. This is comparable to `hdlInputFile -u` XEL command.
`-v1995`	Turns off Verilog -2001 support. By default, Verilog -2001 is supported.
`-vaLib <arg>`	Specifies the `va.lib` file to be used.
`-work <arg>`	Specifies the current working library, `WORK`. If not specified, use `WORK` as defined in `va.lib`. If `WORK` is not defined in `va.lib` either, use `work`.

Verilog-XL

The following Verilog-XL options are also supported:

Table 9-1 Verilog-XL Commands

Command	Description
`+define+<var1>+...`	Define macros.
`-f <commandFile>`	Specify a file for command arguments.
`+incdir+<path1>+...`	Specify paths for searching include files.
`+libext+<ext1>+...`	Specify library directory file extensions.
`-log | -l <logFile>`	Specify a file for log information.
`+librescan | +liborder`	Specify library scan precedence.
`-u`	The analyzer will be case insensitive.
`-v <filename>`	Specify a library file.
`-y <directory>`	Specify a library directory.

© 2021 All Rights Reserved.

vaelab

Synthesizes the design from top module. This utility is comparable to the following XEL command:

`hdlSynthesize -memory -keepVhdlCase -keepRtlSymbol`

Note: The `hdlSynthesize -d` command option for overriding parameter values does not support sized Verilog values like `1'b0`. This is important because the width of the parameter can be significant in sized arithmetic and concatenation expression. Similarly, the `-generic <arg>` option does not support sized Verilog parameters.

The syntax for this command is:

`vaelab [ <topModule> [ -rename <arg>] | -all | -only <arg> ] [options]`

The `[options]` parameter in the above syntax can have the following values:

`-alu`

`-MinALUSize <arg>`

`-append_Log`

`-append_Output`

`-conformal <arg>`

`-debugHardmacro`

`-displayBuffer`

`-displayDepth <arg>`

`-displayRtlScope`

`-displayTimeOut <arg>`

`-displayPort <arg>`

`-dontStopOnError`

`-fanoutNetOnly`

`-file <arg>`

`-generic <arg>`

`-help`

`-keepAllFlipFlop`

`-legacyName`

`-logFile <log_file>`

`-minMemSize <arg>`

© 2021 All Rights Reserved.

-moduleSuffix <mysuffix>

-noMemory

-noNamePreservation

-outputQel <arg>

-outputVlog <arg>

-parallel <number>

-parallelHost

-parallelHostSSH

-parallelLSF

-parallelLSFSameHost <number>]

-parallelNCSameHost <number>

-portNaming <arg>

-refLib <arg>

-skipModule <arg>

-skipModule “<mod1> ... <modN>”

-skipModule “+skipModule <mod1> ... <modN>”

-skipModule "+enableALU <module1> ... <moduleN>"

-skipModule "+disableALU <module1> ... <moduleN>"

-skipModule “+enableMemory <module1> ... <moduleN>"

-skipModule "+disableMemory <module1> ... <moduleN>"

-skipModule "+enableFanoutNetOnly <module1> ... <moduleN>"

-vaLib <arg>

-vhdl2LCase

-vhdl2UCase

-work <arg>

-outputSkeleton

-legacy_bdd

Examples

- To synthesize the design from `top` module, use the following command:

  ```
  vaelab testbench -outputvlog testbench.ver
  ```


© 2021 All Rights Reserved.

- To overwrite the default VHDL generic or Verilog parameter feedback of the specified instance, use the following set of commands. The hierarchical path to the generic can be specified in either VHDL style with a colon (`:`) or Verilog style with a dot (`.`).

```
vaelab top -generic :sub0:feedback=30 -outputvlog top.ver
vaelab top -generic top.sub0.feedback=30 -outputvlog top.ver
vaelab top -generic ":qrbs_glue_1:genQRBSV(4):QRBS:feedback=555" \
           -outputvlog top.ver
```

The following table describes the options of the `vaelab` command.

vaelab Command Option	Option Description
`-all`	Compiles all modules in `WORK` library. Used when compiling all cells in a library.
`-alu`	Switches on ALU mapping with the default ALU threshold of `23`. See Controlling ALU Synthesis by Operand Size for details.
`-MinALUSize <arg>`	Switches on ALU mapping with the ALU threshold as represented by `<arg>`. See Controlling ALU Synthesis by Operand Size for details.
`-append_Log`	Appends log to an existing logfile. When `-logFile` is not present, it appends to `vaelab.log`. When `-logFile <log_file>` is present, it appends to `<log_file>`.
`-append_Output`	Appends to the HDL output file, rather than overwriting it. This is comparable to the `hdlSynthesize -append` XEL command.
`-conformal <arg>`	Specifies the Conformal script file to be generated.
`-debugHardmacro`	User-friendly net names for hardmacro connections for easier debugging.
`-displayBuffer`	Buffers `$display` and `$write` messages to improve emulator performance at run time.
`-displayDepth <arg>`	Specifies the depth of the memory for the `$display` buffer.
`-displayRtlScope`	Specifies that `$display` ("`%m`") outputs RTL hierarchical name instead of netlist name. Block name of `initial` or `always` is included in RTL hierarchical name. The only limitation is that the output of task and function name is not the same as Xcelium.

© 2021 All Rights Reserved.

vaelab Command Option	Option Description
`-displayTimeOut <arg>`	Specifies the maximum number of Palladium fast clock cycles a `$display` message can stay in the buffer.
`-displayPort <arg>`	Specifies the depth of the memory for the `$display` buffer
`-dontStopOnError`	Specifies that synthesis must not be stopped even when the first module with an error is found.
`-fanoutNetOnly`	Only preserves names of nets connected to output ports to reduce the gate count.
`-file <arg>`	Specifies a file for additional command-line options. Use `//` to comment out the rest of the line in the file.
`-generic <arg>`	Assigns or replaces the default value of VHDL generics or Verilog parameters of the top module or at any level of the hierarchy. This is comparable to the `hdlSynthesize -D` XEL command. The `-Generic <arg>` option does not support sized Verilog parameters.
`-help`	Prints the usage information.
`-keepAllFlipFlop`	Preserves all flip-flops.
`-legacyName`	Use the default naming rule in IXE5.1 or earlier.
`-logFile <log_file>`	Specifies a log file name. If not specified, `vaelab.log` is used. This command is comparable to the `hdlLogFile <log_file>` XEL command. This overwrites the log file if exists; otherwise, creates it.
`-minMemSize <arg>`	Specifies the minimum size (number of bits) of the 2D array to be synthesized into MPR or MPW.
`-moduleSuffix <mysuffix>`	This suffix appends to the end of the module name and might contain letters and digits. This is comparable to the `hdlSynthesize -moduleSuffix <mysuffix>` XEL command.
`-noMemory`	Memories are implemented using flip-flops, gates and latches, instead of hard macros. This is comparable to the `hdlSynthesize` XEL command without the `-memory` option.
`-noNamePreservation`	Do not preserve names to reduce the gate count.
`-only <arg>`	Synthesizes only the specified module.

© 2021 All Rights Reserved.

vaelab Command Option	Option Description
`-outputQel <arg>`	Specifies the filename of XEL commands for ATB mode, to be loaded at run time with the `hdlConfig` command.
`-outputVlog <arg>`	Specifies a filename for the gate-level Verilog HDL output.
`-parallel <number>`	Distributes the specified number of processes on the current machine. The maximum `<number>` of parallel HDL-ICE Compiler processes that can be passed as an argument is 248.
`-parallelHost`	Distributes processes over hosts listed in the `parallel.host` file.
`-parallelHostSSH`	Distributes processes over hosts listed in the `parallel.host` file using SSH.
`-parallelLSF`	Distributes processes using LSF.
`-parallelLSFSameHost <number>`	Distributes the specified number of processes using LSF. The maximum `<number>` of parallel HDL-ICE Compiler processes that can be passed as an argument is 248.
`-parallelNCSameHost <number>`	Distributes the specified number of processes using NC. The maximum `<number>` of parallel HDL-ICE Compiler processes that can be passed as an argument is 248.
`-portNaming <arg>`	Uses the given pattern to name vectored ports. The default pattern is `%s[%d]`.
`-refLib <arg>`	Specifies the library in which to search for undefined cells.
`-rename <arg>`	Replaces the top module name in the HDL output files with the specified `<arg>`. This command structure is similar to the structure of the `hdlSynthesize -rename <arg>` XEL command. Refer to the *HDL-ICE Compiler Commands* chapter of the *VXE Command Reference Manual* for more details.

© 2021 All Rights Reserved.

vaelab Command Option	Option Description

`-skipModule <arg>`	Specifies the modules that are either not synthesized or synthesized differently from the global setting based on the arguments like `+enableMemory`, `+disableMemory`, and `+enableALU`. This is comparable to the `hdlSkipModule <arg>` command. If <*arg*> is more than a single word, or if it contains wildcard characters, it must be enclosed within single or double quotes. `<arg>` might consist of multiple module names and arguments. For example, either of the following commands synthesizes the design with top-level-module `top`, generating output Verilog netlist `gates.ver`, skipping synthesis of `mod1` and `mod2` modules, and enabling memory synthesis in all other modules except `mod3`:
	`vaelab top -OutputVlog gates.ver -memory \`
	`  -skipModule "mod1 mod2 +disableMemory mod3"`
	`vaelab top -OutputVlog gates.ver -memory -skipModule \`
	`  "+disableMemory mod3 +skipModule mod1 mod2"`
	To specify a module within a library, other than the working library, add the library name before the module name, such as `lib.mod`. Wildcard character `*` is supported in all module names.

`-skipModule "<mod1> ... <modN>"`	If module names are given before any arguments, such as `+disableMemory`, synthesis of these modules is skipped.
`-skipModule "+skipModule <mod1> ... <modN>"`	Causes synthesis of the specified modules to be skipped.
`-skipModule "+enableALU <module1> ... <moduleN>"`	Enables ALU synthesis in specified modules when neither `vaelab -alu` nor `-MinAluSize` <*arg*> is used. See the Controlling ALU Synthesis by Module section for details.
`-skipModule "+disableALU <module1> ... <moduleN>"`	Disables ALU synthesis in specified modules when `vaelab -alu` or `-MinAluSize` <*arg*> is used. See the Controlling ALU Synthesis by Module section for details.
`-skipModule "+enableMemory <module1> ... <moduleN>"`	Enables Memory synthesis in specified modules when `vaelab -memory` is not used.

© 2021 All Rights Reserved.

vaelab Command Option	Option Description
`-skipModule "+disableMemory <module1> ... <moduleN>"`	Disables Memory synthesis in specified modules when `vaelab -memory` is used.
`-skipModule "+enableFanoutNetOnly <module1> ... <moduleN>"`	Directs `vaelab` in the specified modules, to preserve names of only those nets that are the outputs of the module. This may reduce capacity cost for the modules.
`-vaLib <arg>`	Specifies the `va.lib` file to be used. If not specified, the program searches under the current working directory; then searches under the home directory.
`-vhdl2LCase`	Converts all VHDL names to lowercase. This command option is comparable to the `hdlSynthesize -vhdlToLowerCase` XEL command.
`-vhdl2UCase`	Converts all VHDL names to uppercase. This command option is comparable to the `hdlSynthesize` XEL command without `-vhdlKeepVhdlCase` or `-vhdlToLowerCase` options.
`-work <arg>`	Specifies the current working library, `WORK`. If not specified, use `WORK` as defined in `va.lib`. If `WORK` is not defined in `va.lib` either, use `work`.
`-outputSkeleton`	Creates a directory that has a hard link to each module netlist file using the `short_name.v` defined in the module skeleton file. The directory also has a design skeleton file called `xc_va_s/skeleton.v` that concatenates the module skeleton files by topological order.
`-legacy_bdd`	Enables HDL-ICE Compiler to switch to the old synthesis engine. In ICE flow, use `-parallel <number>` option with `-legacy_bdd` to enable parallel synthesis in old synthesis engine.

© 2021 All Rights Reserved.

Importing the Design

The ICE compilation process is initiated by importing the designs. The design methodology identifies the form of design import that must be used, that is, either netlist or RTL format. The following forms of design import are supported:

- Register Transfer Level (RTL) Import

  Import an HDL design in RTL Verilog or VHDL format, then synthesize it into a gate-level netlist for the design to be passed for compilation. The following sections of this chapter cover the information related to specifying RTL design for import:

  - "Specifying RTL Design Files for Import" on page 246
    - "Specifying Verilog Designs for Import" on page 246
    - "Specifying VHDL Designs for Import" on page 249

  Then, the specified RTL design must be synthesized before it can be used in xeCompile. Refer to Synthesizing the RTL Design for Emulation on page 252 for more details. Cadence recommends using XEL commands to specify and synthesize the RTL designs. For command syntax and usage details, refer to the *HDL-ICE Compiler Commands* chapter in the *VXE Command Reference Manual*.

- Netlist Import

  Designs described in structural Verilog netlist formats can be imported directly without synthesis. Designs described in a combination of HDL and gate-level source can also be imported directly. For detailed information on the netlist import process, refer to Importing a Netlist.

  - Import an existing gate-level netlist in Structural Verilog Hardware Description Language format.
  - Pattern—Lists cell name patterns to create as empty cells for later definition, such as for memories or physical IP logic circuits.

  Cadence recommends using XEL commands to import a netlist. For command syntax and usage details, refer to the *Import Commands* chapter in the *VXE Command Reference Manual*.

Caution

In netlist files, the xeCompile compiler assumes all out of module hierarchical references to be resolved as scalar nets. Otherwise, precompile can fail with the following error message:

© 2021 All Rights Reserved.

```
Compilation is aborted because of unresolved OOMRs. See tmp/OOMR.msg for
details. Use "precompileOption -add {allowUnresolvedOOMR}" to disable this
error.
```

You need to keep a few considerations in mind when writing a Verilog or VHDL design to be compiled for emulation. These considerations and guidelines for modeling the Verilog and VHDL designs are described in Appendix 14, “Modeling Guide for Verilog Designs for ICE Mode” and Appendix 15, “Modeling Guide for VHDL Designs for ICE Mode.”

Importing a Netlist

A netlist file defines the internal, gate-level logic within each logic block of the system that needs to be emulated. The information includes definitions of cells, instances, terminals, nets, and connections.

The gate-level netlist defines all cells in the design, regardless of from where each cell definition originated. The netlist must also define the input and output ports for all memory cells in the netlist, either by using explicit cell definitions or by references to matching patterns in pattern files. The netlist import process uses these port definitions to generate empty cells for each memory instance.

Perform the following steps to import a netlist:

1. (Optional) Specify all input options for importing the design netlists using the `importOption` command. For command syntax details, refer to the *Import Commands* chapter of the *VXE Command Reference Manual*.

   a. Specify whether to incrementally or fully import the netlists by using the following option of the `importOption` command:

      ```
      importOption {mode incr | full}
      ```

      By default, netlist files are imported incrementally. As the netlist importer reads each cell definition in the netlist, it searches for a previous version of the netlist, and compares the new and old versions. It modifies only the cells that are different from the previous version, and adds new cells to the design directory. This makes incremental import faster than a full import because files need not be re-read and compiled unless it is necessary.

      When importing a netlist for the first time, or to make sure the design database does not contain previously imported netlist files (for instance, if some parts of the netlist were deleted), specify the value of the `mode` option as `full` to fully import the netlist. The full import process overwrites previously-imported cells with the same names when the netlist is imported, even if the cell definitions have not changed.

© 2021 All Rights Reserved.

b. Specify the name of the target design library in which the netlist must be imported by using the following option of the `importOption` command:

```
importOption {library <lib>}
```

The netlist import process converts the netlists into a binary format, and places them in the defined *design library*. Several netlists can be imported into the same design library, or import each netlist into a different design library.

If neither you nor the netlist specifies a design library name, the netlist importer uses the default design library name, that is, `QT_design_top`.

c. Execute the `importOption` `{compatibility sl|ns|ci|cs|cd|ir}` command for setting non-standard import options, such as whether or not to overwrite existing cell definitions or override the default case sensitivity or insensitivity setting for the language.

2. (Optional) Specify one or more ASIC or design library names by executing the `refLib` command. The syntax for this command is:

```
reflib <library_name>
```

Note: All ASIC libraries must be present in, or linked into, the `<design>/QTDB` directory.

The netlist importer searches these libraries for the cell definitions to correctly interpret the cells in the imported netlists. In addition, by default, the netlist importer searches the `qtref` library for definitions of the cell primitives that none of the netlists define. If one or more cells are not defined in any of the libraries, then the netlist importer generates an error.

3. Specify the name of the gate-level netlist file that needs to be imported into the QTDB database by executing the `netlistFile` command. The syntax of the command is as follows:

```
netlistFile {<file_type> <netlist_filename>}
```

Netlist files can be either pattern files, netlists of a specific format, or attribute files.

At least one `netlistFile` command must be executed before initiating the netlist import process. You can execute multiple `netlistFile` commands, but those must be followed by a `designImport` command.

Note: If the netlist importer should explicitly exclude any modules during import, create a file containing a list of such modules and specify the name of that file using the `netlistFile {exclude <filename>}` command. Consequently, the Verilog parser excludes the modules listed in the specified `exclude` file, and neither the module's definition nor its instances are imported into the database.

4. Execute the `designImport` command to import the pattern and netlist files.

© 2021 All Rights Reserved.

After the design netlist import has finished successfully, the memory cell patterns supplied by the ASIC libraries are used to generate memory cell definitions and map the design terminals to them. For most designs, memory cells are automatically configured based on the memory instance names defined in the netlist. However, if desired, memory cells can also be manually configured using a user-defined memory or generic pattern.

Using Parallel Import

By default, the netlist import distributes itself in multiple threads on the same host, if it can. Optionally, it can distribute itself on multiple hosts through LSF.

Normally, the `designImport` XEL command invokes the import process in *parallel import* mode. To disable this, use the `designImport -noparallel` command.

Also, the `designImport` XEL command invokes the import process without LSF. To enable multi-host parallel import with LSF, use the `designImport -useLSF` command. However, to use this feature, LSF must be present and appropriately configured.

To enable parallel import for a collection of netlists, make sure that the collection can be partitioned into groups that do not depend on each other. This means, netlists that contribute to the creation of the same library must be a part of the same group. In addition, netlists that consume that library (through `refLib`) must also be a part of that group, and any libraries that they help create must extend the group. Import automatically separates groups, if they are separable, and imports them in parallel.

For example, consider an XEL script containing the following commands:

```
refLib top1 qtref generic
importOption {library top2}
netlistFile {verilog n2.v}
refLib qtref generic
importOption {library top0}
netlistFile {verilog n0a.v}
importOption {library top0}
netlistFile {verilog n0b.v}
importOption {library top1}
netlistFile {verilog n1.v}
refLib top2 qtref generic
importOption {library top3}
netlistFile {verilog n3.v}
designImport
```

The above XEL script file enables import to use two parallel processes: one for netlists `n0a.v` and `n0b.v`, and another for netlists `n1.v`, `n2.v`, and `n3.v`.

The first group (`n0a.v` and `n0b.v`) results from both netlists being imported into the same library (`top0`).

© 2021 All Rights Reserved.

The second group (`n1.v`, `n2.v`, and `n3.v`) results from netlists using libraries created within the group, such that:

- `n2.v` relies on `top1` library, which is created by `n1.v` netlist
- `n3.v` relies on `top2` library, which is created by `n2.v` netlist

Generally, most designs cannot be easily enabled for parallel import. Therefore, care must be taken to yield a design that can support parallel import.

By default, import supports a maximum of 16 parallel invocations, either on same-host threads or multi-host LSF jobs. This can be altered, up or down, by setting the following environment variable:

```
setenv IMPORT_MAX_PARALLEL <value>
```

If more than the maximum number of parallel invocations are needed, they are executed serially in sets of up to the specified maximum number.

By default, import uses the default LSF queue, established by the LSF administrator. This can be altered by setting the following environment variable:

```
setenv IMPORT_LSF_QUEUE <value>
```

By default, import has a time-out of 18000 seconds (that is, five hours) for an import job assigned by LSF. This can be altered by setting the following environment variable:

```
setenv IMPORT_LSF_TIMEOUT <value>
```

Using Parallel Netlist Structure for Parallel Processing in Compile Flow

The compiler generates a netlist structure that supports maximum parallel processing in xeCompile.This netlist structure provides a topological order needed for importing a design and a global instance tree information for OOMR resolution in xeCompile parallel processing. It also enables repetitive trials of different design partitions for parallel processing without redoing IXCOM or ICE compilation.

For information on `-parallelNL` command option, see *Commands and System Tasks for SA Mode* chapter in *VXE Command Reference Manual.*

Packed design netlist file is generated once the module netlist packing is complete with all module netlist files available for parallel synthesis. This single netlist file can be over 100GB for a Billion gate design, so cannot be processed in parallel. The module `skeleton` helps to overcome this performance issue. The module `skeleton` is a module netlist definition without design contents such as ports, nets, and so on, but provides the path to full module definition and instance structure. Module `skeleton` file can be generated in module synthesis in parallel. The design `skeleton` file serves three purposes as follows:

© 2021 All Rights Reserved.

- It facilitates design partition without reading in the full module netlist files.
- It facilitates parallel OOMR resolution within and between partitions.
- It helps import module netlist files to build netlist partitions without physically duplicating netlist in the file system.

Post design synthesis, HDL-ICE Compiler generates a netlist file for each module in the library directory. To support parallel modular xeCompile analysis, HDL-ICE Compiler separates module port definitions from netlist contents using `PORTS_ONLY` text macro separator. For example,

```
module <module_name> (<port_list>);
    <port_declaration>*
`ifndef PORTS_ONLY
    <net or cell or sub-module instantiation>*
`endif
endmodule
```

HDL-ICE Compiler generates a module skeleton file in the same library directory that has the module netlist file. The module skeleton file should use Verilog netlist file format as follows:

```
// <short_name>.v
module <module_name>;
  //pragma CVASTRPROP MODULE SKELETON "<gate_count>"
  [<module_name> <instance_name>();]*
endmodule
```

The special netlist module pragma `SKELETON` only has the gate count of module to be used for parallel imports.

HDL-ICE Compiler also adds a new `vaelab` option (`-outputSkeleton` *`<skeleton_directory>`*) to create design netlist directory.

See "vaelab" section for more information.

xeCompile has added a new netlist file type called `skeleton` to import file type specification. xeCompile must take multiple `skeleton` design netlist directories following the same `designImport` ordering rules. Note that if a `skeleton` type is used, xeCompile requires all other `netlistFile` commands to be of `skeleton` type too for parallel processing. This means any existing legacy design netlist file that has module concatenated, needs to be broken into separate module netlist files in `skeleton` design netlist directory that can be used in parallel flow.

For information on using `skeleton` file type in IXCOM, see "Generating Skeleton Design Netlists for IXCOM" section.

© 2021 All Rights Reserved.

Compiling the Design

After importing and synthesizing the design, the next step in the ICE compilation process involves setting up of the design files to prepare for the compile process. This involves automatic processes that check design rules, generate logic by processes such as loop breaking and synthesis of flip-flops and latches, generating side files, and adding instrumentation logic. This affects behaviors in how the signals are handled relative to the standard fast clock (FCLK).

The basic steps performed are:

- The nets and primitives are initialized.
- The netlist is flattened.
- Design rule check and data qualification is performed.
- Logic transformation is performed on the design.
- Asynchronous loops are broken.
- Flops and latches are synthesized.
- FullVision probes are set.
- Side files are generated.

Files generated at the end of the precompile process include:

- `PDB/<lib>.<cellname>.loops` - Reports complete asynchronous loops in the design.
- `PDB/<lib>.<cellname>.brk` - Reports where asynchronous loops are broken.
- `PDB/<lib>.<cellname>.MultiClock` - Reports the gate count after precompile has run.
- `PDB/<lib>.<cellname>.rpt` - Reports error and warning messages from the design rule-check phase.

This section covers the following topics that detail the intermediary steps of the design setup process:

- Opening the Design Library
- Selecting the Compilation Mode
- Determining the Vision Mode

© 2021 All Rights Reserved.

- Connecting Nets Using iScopedNets Program
- Breaking the Loops
- Synthesizing Flip-Flops and Latches
- Inserting the Instrumentation Logic
- Running the Precompile Process

Opening the Design Library

Open the design library and the top-level cell by executing the `design` XEL command on the xeCompile command line. This command has the following syntax:

```
design <lib> <cell>
```

The *<lib>* option defines the design library, which is the database containing the design. The *<cell>* option identifies the top-level design cell to open.

Selecting the Compilation Mode

The `mode` option of the `compilerOption` command enables you to define the preferred compilation mode: `ice` or `vd`.

- `vd`: You run the designs in the Vector Debug mode without a target connection. To compile a design, issue the following command:

  ```
  compilerOption -add {mode vd}
  ```

- `ice`: For all other modes, use the `ice` option, as follows:

  ```
  compilerOption -add {mode ice}
  ```

Note: You must specify either `ice` or `vd`. Otherwise, the precompile fails with the following error message:

```
ERROR (legacy-30039): No compilation mode is specified. Use 'compilerOption'
command to set a compilation mode.
```

Determining the Vision Mode

After having identified the compilation mode, you must accordingly determine the vision mode, that is, whether FullVision or Dynamic Probes.

FullVision is the default vision mode during compilation.

© 2021 All Rights Reserved.

Use the following command to specify the vision mode:

```
compilerOption -add {visionMode FV | DYNP}
```

If `compilerOption -add {visionMode 0|DYNP}` is specified, FullVision is turned off and thus the subsequent precompile generates database for DYNP mode.

If `compilerOption -add {visionMode FV}` is used, FullVision is turned on.

You do not have direct control over FullVision trace depth. Compiler determines the trace depth based on the design. In most cases, you can get 2 million fclk cycles of trace depth. If you want more trace depth, then you need to compile the design with more domains.

In DYNP mode, you can change the number of dynamic probes that can be added at run time using the `compilerOption -add {numCapturedNetsPerDomain <value>}` command. For example, to reserve 2048 probes per DCC, compile the design with the following command:

```
compilerOption -add {numCapturedNetsPerDomain 2048}
```

In FV mode you might also need to add dynamic probes in some scenarios. For example, when forcing or releasing a signal with fclk running or when adding fast probes, compiler preserves some resource for dynamic probes. However, if you try adding too many dynamic probes compilation might fail.

Use the following command to reserve extra space for adding dynamic probes:

```
compilerOption -add {numExtraCapturedNetsPerDomain <value>}
```

The `<value>` must be in the range of `512` to `774144`. The `<value>` must be in multiples of `512`, else it will be rounded off to the nearest multiple of `512`. Though the maximum space that can be reserved for dynamic probe is dependent on the design, specifying too large value could result in failure of compilation.

Note, that adding this compiler option also impacts performance as the minimum step count increases due to too many probes in the design. If there is a drop in frequency, review `et3compile.msg` file to view the line “Min step count due to probes”. If it is too high, check to see if `compilerOption -add {numExtraCapturedNetsPerDomain <value>}` has been included in the compile script. If it has been included and you do not need it, remove it from the XE Compile script.

However, if you fail to add dynamic probes at run time, an error message will prompt you to re-compile your design using `numExtraCapturedNetsPerDomain` compiler option. For example, to reserve 10000 probes per DCC, compile the design with the following command:

```
compilerOption -add {numExtraCapturedNetsPerDomain 10000}
```


© 2021 All Rights Reserved.

Note: Setting `numCapturedNetsPerDomain` or `numExtraCapturedNetsPerDomain` value to over half of the max value is not recommended as it can significantly reduce the emulation speed or result in compilation failure.

Selecting a Vision Mode Based on Preferred Debug Strategy

Creating a compiled program suitable for different debug strategies requires setting up the design using different options such as emulation mode, selecting a vision mode (if any), and compiling the design.

The following table identifies the appropriate the `compilerOption` command options that must be executed to define the compilation and vision modes based on different debug strategies:

Table 9-2 Debug Strategy by Option

Debug Strategy	Compilation and Vision Modes	Required `compilerOption` Command Options
Vector Debug	Specify the compilation mode as *Vector Debug*. Vision mode specification is not required.	`-add {mode vd}`
Dynamic Probes (STB or Logic Analyzer)	Specify the compilation mode as *ICE* and vision mode as *Dynamic Probe*.	`-add {mode ICE)` `-add {visionMode DYNP}`
Static Target TestBench (STB)	Specify the compilation mode as *ICE* and vision mode as *FullVision*.	`-add {mode ICE)` `-add {visionMode FV}`
Logic Analyzer (LA)	Specify the compilation mode as *ICE* and vision mode as *FullVision*.	`-add {mode ICE)` `-add {visionMode FV}`

Using InfiniTrace

ICE modes (FV and DYNP) automatically use InfiniTrace. InfiniTrace is *on* by default. Leaving it set to the default is recommended, but in cases where there are a significant enough number of primary inputs as to possibly cause compilation to fail, it can be turned off. Use the following command to enable or disable InfiniTrace:

© 2021 All Rights Reserved.

```
compilerOption -add {infiniTrace on | off}
```

Compiling for STB Mode

Select ICE through XEL using `compilerOption` command when setting up design compilation.

For FullVision:

```
compilerOption -add {mode ice}
compilerOption -add {visionMode FV}
```

For Dynamic Probes:

```
compilerOption -add {mode ice}
compilerOption -add {visionMode DYNP}
```

Compiling for LA Mode

To set the ICE LA mode, through XEL using the `compilerOption` command when setting up design compilation, use the following commands:

For FullVision:

```
compilerOption -add {mode ice}
compilerOption -add {visionMode FV}
```

For Dynamic Probes:

```
compilerOption -add {mode ice}
compilerOption -add {visionMode DYNP}
```

LA mode supports FV on Palladium Z1. However, be aware that you cannot append multiple triggers to one waveform. The previous waveform is cleared unless a different waveform file is specified before the next trigger and upload occurs. InfiniTrace cannot be used in LA mode.

Compiling for VD Mode

To use Vector Debug mode, use the following command:

```
compilerOption -add {mode vd}
```

Connecting Nets Using iScopedNets Program

The `iScopedNets` program enables the connection of nets upward through the design hierarchy to a desired level. It can also be used to make a net the top-level input or output of the design.

© 2021 All Rights Reserved.

The `iScopedNets` program modifies cell definitions in the `QTDB/<library>.llib` files. These files are created by the `designImport` command or by xeCompile synthesis.

The `iScopedNets` program can be run from either the UNIX or XEL command line using any one of the following command syntax:

```
iScopedNets -f <nets_file>

iScopedNets [<options>] <library> <cell> <path> <topName> [<separator>
     [<flags>]]
```

Important

This program should be run after the `designImport` command and before the `design` command.

The `iScopedNets` program has additional capabilities not described here. For a complete description, execute the `iScopedNets` command without any arguments.

Creating Nets File for iScopedNets

To use `iScopedNets`, first create a file (*<nets_file>*) that describes the modifications for `iScopedNets` to perform. This file uses the format shown below, where each line after the VERSION line describes a net to be brought upward through the design hierarchy.

```
VERSION <version_number>
<library> <cell> <path> <topName> [<separator> [<flag>]]
<library> <cell> <path> <topName> [<separator> [<flags>]]
   ...
<library> <cell> <path> <topName> [<separator> [<flags>]]
```

The components of these entries are as follows:

<version_number> is the version of the file. It should be at least version 3, except in legacy applications. (See Providing Vector (Bus) Support, below.)

<library> specifies a library file (*<library>*`.llib)` in the `QTDB` directory.

<cell> specifies a cell within this library

<path> is the instance path to a net within *<cell>*

<topName> is the name of a terminal on the top cell created and connected to the net specified by the *<path>*. To connect the specified net upward through the hierarchy, provided the version is 3 or above, a new terminal named *<topName>* is created at each level and all levels but the lowest level containing *<path>*. If no net named *<topName>* exists at this level, a new net named *<topName>* will be created. If a net named

© 2021 All Rights Reserved.

<topName> already exists at the specified level, it is merged with the net being connected upward. If the version number is 2 or below, and the net specified by *<path>* already connects upward in the hierarchy, the existing upward connections are used to connect as far upward as possible. If an existing connection is used at a specified level, the net being connected upward is not merged with any existing net named *<topName>* at that level.

<separator> is a mandatory string delimiting the hierarchy. It defaults to the standard hierarchy delimiter of "_".

<flags> is a collection of special control parameters. Seven options of "t" (terminal), "h" (hidden), "r" (reference), "c" (connect), "n" (name), "o" (optimize), and "d" (drive) are available.

Consider the following example:

```
MYLIB MYCELL u1__u2__net3 my_top_level_net3
```

The above line entry has `iScopedNets` locate the net named `u1__u2__net3` within the hierarchy of cell `MYCELL` in library `MYLIB`, and bring it upward through the hierarchy using the name `my_top_level_net3`. In addition, a new terminal named `my_top_level_net3` will be created. This terminal will be an output if the net connected to it has a driver anywhere inside `MYCELL`; otherwise it will be an input.

Caution

Running iScopedNets modifies cells in the specified libraries as well as in hierarchically-referenced libraries. This affects subsequent incremental netlist import. If netlists are to be re-imported, run FULL import of ALL netlists, followed by re-running iScopedNets.

Providing Vector (Bus) Support

If the input file (*<nets_file>*) version number (*<version_num>*) is 1, then *<path>* and *<topName>* are treated as scalars, that is, a single net connecting to a single terminal.

If the version number is 2 or higher, *<path>* and *<topName>* can be treated as vectors, referring to groups of nets and terminals.

Following two forms of vector notation are supported:

- Implicit Range Notation
- Explicit Range Notation

The *<path>* and *<topName>* arguments can have different notation forms; *<path>* can be explicit and *<topName>* implicit, or any other combination.

© 2021 All Rights Reserved.

If more elements are specified by `<path>` than by `<topName>`, iScopedNets will fail with errors.

Implicit Range Notation

Implicit range notation consists of the basename, without any index information: for example "`data_bus`".

If implicit range notation is specified, it is first tried as a scalar. If a scalar net matches the path, it is connected. Thus, if `a__b__data` and `a__b__data[...]` exist, and the path is specified via implicit range notation (`a__b__data`), only the scalar net is handled.

A path specified with implicit range notation is then compared with nets of the form `<path>[index]`, with the index starting at zero.

For example, if there exist nets `a__data[4]`, `a__data[5]`, `a__data[6]`, and `a__data[8]`, then the `<path>` of `a__data` is equivalent to the `<path>` of `a__data[6:4]`. `a__data[8]` is NOT included; ranges must be contiguous. The range is in the form `[larger:smaller]`.

If `<topName>` is expressed in implicit range notation, the width of the top port will be the width of the `<path>` vector. (If there are no connection errors, this guarantees that the number of elements in `<path>` and `<topName>` are equal.)

If `<topName>` is expressed in implicit range notation, the names of the top created terminals will bear the indexes of the connected nets.

Explicit Range Notation

Explicit range notation consists of the basename and an index range, for example: `data_bus [7:0]`.

When explicit range notation is used with `<topName>`, the bit order of the range is preserved. For example, a `<path>` of `a__d[0:3]` and a `<topName>` of `bit[3:0]` connects net `a__d[0]` to terminal `bit[3]`, and so on.

Breaking the Loops

When xeCompile runs through the design precompile cycle, it automatically breaks asynchronous feedback loops.

This process generates two log files to aid in debugging. These files are:

© 2021 All Rights Reserved.

- `PDB/<lib>.<cellName>.brk`
- `PDB/<lib>.<cellName>.loop`

xeCompile automatically breaks loops without user intervention. However, by allowing you to manually break loops, performance can be enhanced. The In-Circuit Emulation performance of xeCompile is directly related to the step count, which is correlated to the length of combinatorial paths. By inserting delay elements to break false paths or multi-cycles paths, emulation performance can be improved, possibly dramatically, if the eliminated paths are the critical paths of the design.

Design modeling can also require the insertion of a delay to break combinatorial loops. These loops are not necessarily real combinatorial loops in the design. These can be static loops resulting from the modeling strategy used by xeCompile. To obtain a successful compile, it is necessary to break the loops. User intervention can aid in design modeling by allowing you to manually specify where loop breaks should occur.

Figure 9-4 Loop Breaking

There are three ways to manually break loops.

© 2021
All Rights Reserved.

- By inserting a FCLK flip-flop (`Q_FDP0B`) in the netlist.
- By using the `breakNet` XEL command to insert an FCLK flip-flop at the driver of a particular net.
- By using the `breakPin` XEL command to insert a FCLK flip-flop at a particular pin of an instance.

Each of these methods can be used to break a feedback loop or long combinational path. They insert an FCLK flip-flop which gives a delay of one FCLK cycle. The `breakNet` and `breakPin` commands provide a way of breaking loops or paths without editing design sources. These commands should be run after design import and before the precompile step.

Note: Loop breaking should be executed with extreme caution. The compiler software follows user request and broken design functionality can result from improper use.

The `et3compile.msg` file and the `/dbfiles/<topcell>/.memTransformList` indicates if compaction was performed and if so, gives a list of compacted memory instances in `<topcell>.memTransform`.

Detailed information on loop breaking can be found in Appendix 12, “Loop-Breaking.”

Synthesizing Flip-Flops and Latches

The precompile step uses a default behavior that affects flip-flops and latches as described below.

For flip-flops, when reset goes active, the output goes low in the same FCLK cycle. When reset goes inactive, if the clock goes high in the same FCLK cycle, the output stays low even if data input D was high.

For latches, when reset goes active, the output goes low in the same FCLK cycle.

When the reset goes inactive, one of the following conditions can occur:

- if the enable goes active in the same FCLK cycle, the output gets the value that data input D has in that cycle.
- if the enable goes inactive in the same FCLK cycle, the output stays low even if data input D was high.

Inserting the Instrumentation Logic

This step can include the following:

© 2021 All Rights Reserved.

- **Resolve the tristate drivers**

  Use the `precompileOption -add disconnectTristateDrivers` command to automatically disconnect the tristate drivers (with a warning) from any net that also has regular drivers. By default, if a net has both tristate and regular drivers, `precompile` fails.

- **Trim the parallel drivers**

  Use the following arguments of the `precompileOption -add` command to trim the parallel drivers:

  - `tieSourceless`: Ties sourceless user nets to a "safe" constant
  - `tieSourcelessHigh`: Ties sourceless user nets to power
  - `tieSourcelessLow`: Ties sourceless user nets to ground

- **Remove the sourceless logic**

  Use the `precompileOption -add removeSourceless` command to disconnect the sourceless cones in the design considering them as "weak" logic that does not affect anything.

- **Remove the loadless logic**

  Use the `precompileOption -add removeLoadless` command to direct precompile to remove loadless logic including gates, flip-flops, latches, and memories. The goal of the removal is to reduce the emulator resources needed by a design.

- **Prevent removing certain loadless portions of DUT**

  `keepLoadlessInstance`, `keepLoadlessModule`, and `keepLoadlessNet` commands affect `removeLoadless` behaviour only. For example, `keepLoadlessNet -add <net>` will ensure visibility of net without marking it as `keepNet`. Similarly, `keepLoadlessModule` and `keepLoadlessInstance` prevent the corresponding module or instance from being optimized away by `removeLoadless` option.

- **Merge the buffer-inverter chains**

  Use the `precompileOption -add keepMem` command to cause `precompile` to automatically treat as `keepNets`, those bits of `HDL_MEMORY` arrays that are not driven by buffers, inverters, or combinational gates.

  The *benefit* of treating the `HDL_MEMORY` array bits as `keepNets` is that the bits treated as `keepNets` can be loaded from a file (for example, using a `$readmemh` operation or `memory -load` XEL command). However, if the bits driven by combinational gates (including buffers or inverters) are loaded, the loaded values will remain only transiently before they are overwritten by the driving gates.

© 2021 All Rights Reserved.

The *disadvantage* of treating `HDL_MEMORY` array bits as `keepNets` is a possible severe degradation in step count. The degradation happens when multiple array bits are driven by the same (or equivalent) combinational logic.

Running the Precompile Process

After specifying the required precompile options and User Data, you can explicitly execute the `precompile` command. However, Cadence recommends that you instead do the following:

1. Execute the `design` command to specify the design library and cell name.
2. Use the `precompileOption` command to specify options for precompile.
3. Specify the required User Data.
4. Specify User Data for the target interface (if any).
5. Run the `compile` XEL command.

On running the last step, xeCompile automatically invokes the precompile process before the core compilation. The progress of the precompile process, including warnings and errors, is displayed in the xeCompile console. The process runs tasks such as initializing the nets and primitives; setting the clock, data paths, and exclusivity points; breaking the loops; and inserting the instrumentation logic.

Calling `precompile` multiple times (implicitly or explicitly) within one xeCompile session is unsupported.

For example, the following cases would error out.

Example 1:

```
emulatorConfiguration -add {host emulA} {boards 0.0}
precompile
keepNet -add top.x.y.z
precompile # The explicit second precompile call
```

Example 2:

```
emulatorConfiguration -add {host emulA} {boards 0.0}
precompile
keepNet -add top.x.y.z
compile # The implicit second precompile call
```


© 2021 All Rights Reserved.

Optimizing the Precompile Process

You can use various optimization techniques for collecting the run-time data that might help in optimizing a subsequent compilation. These optimization techniques are based on the toggle clock data, and are described in Chapter 8, “Behavior-Restricted Compilation”.

Verifying Optimization

A short test can produce restrictions for high compilation optimizations. However, a longer test need not follow these restrictions.

For example, consider a test where the clocks are multiplexed so that their edges have different relative alignment in different tests, or a test with a certain clock domain dead in one test and functional in another one.

During precompile, restricted clocks are instrumented so that these restrictions can be verified during the subsequent run. A message like the following is issued at compile time to indicate the instrumentation:

```
INFO (multclk-1400): Compilation restrictions are instrumented for verification.
Net CVA_UA_VERIFY_SHADOWS_DATA will be asserted when violation occurs.
```

For information about the run-time use model for the `CVA_UA_VERIFY_SHADOWS_DATA` signal, refer to Detecting shadowsData Violations section.

You can view the list of clocks that did not follow compile-time restrictions in a file named `BAD_EDGES` in the session directory.

Preparing the Design Files for Compilation

After having specified the precompile options and other required User Data, prepare the design files for core compilation and then run the compile.

This section covers the following topics:

- Setting Up the Design for Precompile
- Incorporating Physical IP
- Running ICE Prepare
- Support for Long Module Names

© 2021 All Rights Reserved.

Setting Up the Design for Precompile

This step involves setting up of the design files to prepare for the compile process through automatic processes that check design rules, generate logic by processes, such as loop breaking and synthesis of flip-flops and latches.

This step also includes specification of the following:

- Emulation host and domains that must be used for compilation
- Compilation and vision mode for which the design is to be compiled
- Parameters for activating clocks, signal properties, and instrumentation logic

Preparation for compilation includes specification of cable connection parameters, such as cable types, terminal assignment, and timing constraints, that need to be used with a target system. In addition, xeCompile runs data qualification and generates the control files for the design database. The design preparation step includes specification of the cable connection parameters that need to be used with a target system. These parameters include cable types, terminal assignment, and terminal timing commands. The commands for setting these parameters are described in Working with the Palladium Z1 Emulators chapter.

Incorporating Physical IP

The Physical IP (Intellectual Property) refers to user hardware that substitutes for certain instances of the design. As part of design setup, you must specify the mapping between IPs and design cells.

In xeCompile, when Emulator Configuration is run, it records information on the IPs present in the IP chassis, as well as their cable connections, into an emulator configuration file.

Physical IP mapping between the IP terminals and pins on the IP chassis is performed at the ICE prepare step in emulation mode.

IP protection can be implemented by using the Cadence `xmprotect` utility. For more information, refer to the `xmprotect` documentation relevant to XMVHDL and XMVerilog, available from Cadence Online Support website.

IP Mapping

Input to the `icePrepare` mapping process is in the form of a `<IP-ID>.fem` file and an optional `<IP-ID>.ttc` file, both provided by the IP Builder utility and the Emulator Configuration file generated by `autoconfig` and user design. The `*.fem` (that is, *front end mapping*) file provides IP terminal connectivity information. The `*.ttc` file provides terminal

© 2021 All Rights Reserved.

timing constraints for the IP. If this `.fem` file is present in the design files, it will be automatically found by the icePrepare process.

During the icePrepare process, design instances are assigned to IPs, and IP terminals are mapped to their corresponding ICTI pins. This mapping is recorded in the output control (*.ctl) file. This process generates two reports, which are placed in the current design directory: `pichInstAssign.report` and `pichSignalMapping.report`. The `pichInstAssign.report` file reports the assignments of design instances to the IPs. The `pichSignalMapping.report` file reports the mapping of the IP terminals to the ICTI card pins on the IP chassis.

The IP Builder constructs the `*.fem` file from a user-supplied input file. The input file must have the structure shown in the example.

The first line must always be `IPID <IP-ID>`.

Use the `IPMAPFILE` entry to name the back end mapping file.

Following is an example of an IP Builder input file:

```
IPID 123
IPName test1
IPDescription test ipBuilder
IPMAPFILE ipbmf_proto.map
Date xx/xx/xx 12:00
Revsion x
Comments
This is an example of ipBuilder input
.
CELL C1 C1
TERM in1 1-22
TERM in2 2-15
TERM out1 3-3
TERM out2 4-8
```

The `*.ttc` file is an XEL script containing the terminal timing constraint commands for a user's design. The sample contents can be:

```
#IPID 123
#IPNametest1
delayBox -add {db1 2 C1:in1}
terminalTiming -add {C1:out1 db1}
```

IP Builder

Palladium Z1 provides a special tool for creating a virtual representation of user Physical IP that can be used by xeCompile. IP Builder is a menu tool that performs IP mapping and creates a front-end mapping (`.fem`) or transposer pin (`.tp`) file for each IP Block instance. These front-end mapping files are read by the compiler, allowing the compiler to route the required signals to and from the Physical IP block.

© 2021 All Rights Reserved.

Running ICE Prepare

The ICE Prepare step is a mandatory step for the ICE flow. During this step, target-specific information is generated for the design database. This includes the ICTI pin assignment, mapping between IP terminals and ICTIs on the IP chassis, and other configuration assignment processes are performed.

The ICE prepare step aims at processing the target-related User Data and generating files that will be used during compile. This process can be invoked in two ways:

- Explicitly execute the `icePrepare` XEL command on the command line

  Or

- Execute the `compile` command after the User Data definitions. This command will automatically call the ICE prepare function before the core compilation.

Support for Long Module Names

The LINUX file system does not support file or directory names that are longer than 256 characters. LINUX also does not support sub-directories exceeding 32k on EXT3 disk. This causes software failure. To avoid this issue, use `-db2` option with `vavlog`/`vavhdl` commands. This option helps HDL-ICE Compiler to split long module name into sub-directories and the last sub-directory is converted to a database format that flattens most of the sub-directories, thereby, resolving the LINUX EXT3 file system limitation on the number of sub-directories. A lesser number of sub-directories enhances the performance of HDL-ICE Compiler.

Long Module Name Compression Rules

Long module name compression obeys the following rules:

- Module names that are longer than 512 characters only are compressed.
- Only the part of the name that follows the first _ (underscore character) after the 512th character is compressed.
- MD5 encoding algorithm is used to generate the 32-digit hexadecimal number.
- The `C` character is added before the MD5 string to indicate that the name is compressed.

Example of Long Module Name Compression

```
RB18_INTERNAL_VLOG_INIT_A_18h0_18h0_READ_WIDTH_A_9_9_x5245474345_x5245474345_SRVA
L_A_18h0_18h0_WRITE_WIDTH_A_9_9_INIT_00_256h72c1623190e1236091b0b312f1e27022d240f
371c321f032538330439050104_256h6052b04152a220318140d291121350208171a130a0c30282e1
```


© 2021 All Rights Reserved.

```
01d2026340101_256h1010101010101010101010101010101010101010101010101010101010101010101
_256h1010101010101010101010101010101010101010101010101010101010101010101_256h10101010
1010101010101010101010101010101010101010101010101010101010101010101_256h1010101010101010101
0101010101010101010101010101010101010101010101010101_Cc4f78ab0300d3953449b249385350af9
```

In the above example:

- The part of the name that appears in the `default` font is the part that is kept intact.
- The **`C`** character in **`bold`** font indicates that the rest of the module name is compressed.
- The part of the example in *italics* font indicates that it is the MD5 string.

Compressed long module names are supported with parallel synthesis. To enable parallel synthesis, execute the following commands:

```
vavlog -db2 <source_file>
vavhdl -db2 <source_file>
vaelab -parallel <number> <top_module>
```

The `vaelab` utility automatically detects the database format and synthesizes accordingly. However, you cannot mix the use of flattened and non-flattened sub-directories. Therefore, either parse all of the files with the `-db2` option, or all of the files without the `-db2` option.

Use the `-db2` option with `hdlImport` command to change the database format so that the sub-directories are flattened into files.

Support for Retaining Flip-Flop Signals

Use the `-keepGenFlipFlop` option with the `hdlSynthesize` command to keep the flip-flop signals defined in the `generate` block. By using this option, you can avoid the task of retaining flip-flop signals instead of using the `// quickturn pragma` statement, such as the following:

```
// quickturn preserve count_dly
```

Consider the following example:

```
generate
        if (DELAY == 0) begin : no_dly
                assign count = count_tmp;
        end
        else begin : dly
                reg [1:0] count_dly;
                // Not-Used: quickturn preserve count_dly
                always @(posedge clk)
                begin
                        if(rst)
                                count_dly <= 'h0;
                        else
                                count_dly <= count_tmp;
                end
                assign count = count_dly;
```


© 2021 All Rights Reserved.

```
        end
endgenerate
```

In the above example, HDL-ICE Compiler will not retain the flip-flop signal `count_dly` and will use an additional signal `count` as the flip-flop output. To retain the signal `count_dly` in the generated netlist, use the `-keepGenFlipFlop` option.

Support for Encryption in HDL-ICE Compiler

HDL-ICE Compiler supports RTL source files encrypted using the following encryption methods:

- Verilog-2001 standard encryption by `xmprotect`
- Verilog-XL encryption
- Verilog-2005 standard encryption by `xmprotect`

Irrespective of the encryption style used in the RTL, the gate-level netlist, generated by the HDL-ICE Compiler, will be encrypted using the `xmprotect` Verilog-2001 standard. You can import all files with any encryption style to the HDL-ICE Compiler together without using any additional option.

Debug Capability of Encrypted Source Code

You can use the `xmprotect` utility to encrypt a file with three levels of debug capability. For more information, refer to Encryption with Levels of Debug Capability in Chapter 16, “Compiling Designs for Simulation Acceleration Using IXCOM.”

xeDebug versus verigen

You can debug the following at xeDebug run time through probe and waveform display:

- `debugall`: Probe the signals, nets, and content of the instances.
- `portview`: Probe ports of the IP module.
- `none`: Default option. Cannot probe an IP module.

You can use the `verigen` program to dump the following from a design database:

- `debugall`: Stop at IP module, but you can see the module name and port name as the full path of waveform (cleartext). Cannot see instance contents and connectivity.

© 2021 All Rights Reserved.

- `portview`: Stop at IP module, but you can see the module name and port name as the full path of waveform (cleartext). Cannot see instance contents and connectivity.
- `none`: Stop at IP module and show the encrypted name of IP module (not cleartext).

Protection Right in HDL-ICE Compiler

A protection right is applied to a module based on the following conditions:

- When a module is enclosed within one protected region, the HDL-ICE Compiler gate-level output encrypts the entire module with the same protection right.
- When a module has both cleartext and multiple protected regions, the HDL-ICE Compiler splits the declared objects into cleartext objects and protected objects. In the gate-level netlist, all cleartext objects are declared in cleartext and the rest of the netlist is encrypted.
- When a module has multiple protected regions with inconsistent protection rights, the HDL-ICE Compiler encrypts the protected part of the netlist by using the strongest right. For example, if a region has `debugall` and another region has `none` in RTL, the protected regions in gate-level netlist will use `none`. If a region has `debugall` and another region has `portview` in RTL, the protected regions in gate-level netlist will use `portview`.

Importing Files by Using designImport

The import process, which is invoked with the `designImport` command, encrypts netlist entity names within the compiled database to respond to the protection rights assigned to a netlist source by a command-line option or an embedded pragma. Entity names derived from a protected source or a protected region within a source are, by default, encrypted.

If the rights to the protected source, or some region of the protected source, are specified as `debugall`, then the entity names derived from the specified source or region are stored as cleartext.

If the rights to the protected source, or some region of the protected source, are specified as `portview`, then the entity names of module, ports, and port-nets (nets connected to ports) within the specified source or region are cleartext, while the names of other entities are encrypted.

© 2021 All Rights Reserved.

Compiling the Design Files

Compiling the design places and routes the design in the emulator, and creates a compiled database that is used when running and debugging the design.

The compilation process starts on executing the `compile` XEL command. While the design is being compiled, messages about the status of each phase of design compilation are displayed, such as partitioning the design, and whether each phase was completed successfully or failed.

Note: The `compile` XEL command starts the process of core compilation only after successful execution of the `precompile` process.

For syntax details and more information on the `compile` XEL command, refer to the *Compiler Commands* chapter of the *VXE Command Reference Manual*.

Note: A design must be run on the same platform for which it was compiled. For example, a design compiled on the Linux platform must be debugged on the Linux platform.

This section includes the following topics:

- Using Parallel Partition Compiler for Large Designs
- Modular Compilation in ICE Mode
- Viewing Compile Results
- Using Compiler Options for Better Compile-Time Performance

Using Parallel Partition Compiler for Large Designs

Cadence recommends using Parallel Partition Compiler (PPC) to reduce compilation time for larger designs, that is, 500M gates and more.

The first step to enable parallel partition compiler is to manually partition the design. PARTITIONS directories are located within the `design` directory.

Next step is to assign an instance to a selected partition by using the following command:

```
partitionGroup -add {<instance>[.][<prer>][*]<partition>}
```

For syntax details and more information on the `partitionGroup` command, see *User Data Commands* chapter in *VXE Command Reference Manual*.

You can also leverage the PPC to generate multiple software FullVision partitions (`swfvDB`) and then generate a SAIF file for each partition. The state-dependent and path-dependent

© 2021 All Rights Reserved.

(SDPD) forward SAIF for backward SAIF generation is not supported with partial `swfvDB` files. For information on how to use the partial `swfvDB` files to accelerate the generation of SAIF files, refer to the Accelerating SAIF Generation Using FullVision Partitioning section.

Prerequisites for Using Parallel Partition Compiler (PPC):

- Ensure LSF is installed and available in the operating system path. To minimize the compilation time, the user should have enough available LSF hosts/cores to run all the partition jobs simultaneously. Job distribution through LSF is controlled by `LSF_options` file in the design directory. For this, user has to provide two lines in the `LSF_options` file. The first line is for the full design and the second line is for partitions.

  Note: For information about LSF wrappers supported for PPC, see section "Using a Job Scheduler Other Than LSF" on page 348

- Host memory requirements for partition jobs is 1GB for each million gates.
- Divide the design into sufficient number of partitions. However, more than 30 partitions is not recommended.
- Make partitions of roughly equal size to ensure the compiler does not have to wait for partition jobs in large partitions that take longer, while partition jobs in smaller partitions are done.
- Ensure partitions are not too small as it might limit design optimization and create a critical path jumping multiple times between partition-assigned boards and domains. Thereby reducing design performance (meaning reduced clock speed). Create large partitions by including more instances into lesser amount of partitions.
- The partitions should have limited interconnection. To ensure this, minimize the combinational paths crossing from one partition to another, especially if the path spans many partitions. Break the path or group the instances crossed by the path into lesser amount of partitions.
- For each partition, you can assign emulator boards and domains by using `partitionAssign` compiler command. This is similar to how users assign the emulator configuration to the entire design.

  For syntax details and more information on the `partitionAssign` command, see *User Data Commands* chapter in *VXE Command Reference Manual*.

  Try to maintain the contiguous space and minimize the hardware hierarchy crossing. If some or all partitions have no boards or domains assigned, compiler automatically assigns boards/domains to those partitions.

© 2021 All Rights Reserved.

Once all the conditions are met, use the following compiler option to enable parallel partition compiler:

```
compilerOption -add {distributedMulticore ON | OFF | <path>}
```

When `distributedMulticore` compiler option is turned `OFF` or is not specified, compilation runs in regular mode. When `distributedMulticore` compiler option is turned `ON`, compilation runs in parallel partition mode.

When `distributedMulticore` compiler option is turned `ON`, each partition is compiled in its own directory.

The directories are created in `PARTITIONS` subdirectory within the `design` directory. You can also specify a *<path>* where you want to create the directories for the partitions.

When using `compileFind` command, critical path information as well as other information for all trials are saved as `et3compile` log files and copied in `./tmp` directory as `et3compile.Trial-<i>.log` file.

Note: For the list of `compileFind` options supported for parallel partition compiler, see *Compile-Time Commands in ICE Mode* chapter in *VXE Command Reference Manual*.

Methods to Optimize Design and Improve Performance When Using Parallel Partition Compiler (PPC)

- Use `PostMergeOpt` compiler option to enable a certain amount of optimization after partition databases are merged into one. This involves netlist optimization across partition boundaries which can add to the compilation time but reduces the design size.

  For syntax details and more information on the `PostMergeOpt` compiler option, see *User Data Commands* chapter in *VXE Command Reference Manual*.

- Make large partitions by including more instances into a lesser amount of partitions as larger partition means more optimization is possible.

- When certain partitions do not get optimized enough, the amount of domains allocated to those partitions may not be adequate. In case of compilation failure or low design performance, check instruction usages in the compilation logs for each partition to see if its hardware utilization is too high (80% or more). If that is the case, assign more domain to those partitions, maybe at the expense of other partitions with lower instruction usage. Instruction usage can be seen in *<partition dir>*/*<partition ID>*/tmp/ `et3compile.msg or` *<same path>*/`et3pass1.msg` file.

- Reduce long critical cross-partition path by creating partitions with registers on their inputs/and or outputs. Try to reduce combinational cross-partition paths, including flip-flop clock and set/reset inputs. You can also reduce critical cross-partition path by using

© 2021 All Rights Reserved.

`PostMergeOpt` compiler option, or create large partitions by including more instances into lesser amount of partitions.

- Uneven distribution of design memory (by size or by the number of memory read/write ports) across partitions can result in compilation failure or in reduced design performance due to memory on critical path. You can resolve it by moving some of the memory-rich instances to different partitions, or by creating larger partitions by combining memory-rich instances with instances that have less amount of memory. It can also be resolved by increasing the number of domains assigned to memory-rich partitions.
- Too high concentration of design probes within certain partitions may increase the design step count and decrease design performance. To resolve this, try balancing memory read ports and primary inputs by moving memory-rich ports and input-output to other partitions, by enlarging partitions, and by adding more domains to memory-rich partitions.
- Opt for auto partitioning of designs in PPC flow so you can use the auto-partitioned design group for enhanced compile performance. Use the following QEL command to auto partition a design into partition groups:

  ```
  autoPart
  ```

  This command ensures that any partition grouping and partition assignments existing in the user data is overwritten by xeCompile when this command is invoked. It identifies suitable number of partitions of the design based on the resources and design stats and groups them into partition groups.

  From the emulator configuration, xeCompile determines the number of boards and domains. There must at least be four full boards or 32 domains in Palladium Z1 for using automatic partitioning. Also, the given boards must be well connected as the command does not check that. If you change the emulator configuration after partition grouping is complete, then you must re-run automatic partitioning although xeCompile does not check that. xeCompile also does not check whether the given configuration has enough resources. If you do not provide sufficient number of boards or if the boards are not well connected, compilation may fail later in the flow.
- Final grouping information is logged in `tmp/partitionGroup.qel` file which contains a series of `partitionGroup` commands that associate various instances of the design to their partitions. At this stage, any existing user data of `partitionGroup` is cleared and new user data related to partition grouping is created so that precompile can pick up the updated partition information in PPC flow. If the generated partition grouping is not suitable for the design and configuration, then users are expected to review the file generated and make the required changes. An easier way to make changes to the chosen partitioning is to copy the relevant `partitionGroup` commands from `tmp/`

© 2021 All Rights Reserved.

`partitionGroup.qel` file into the user script and reassign the partition to those instances. User data gets updated automatically when the script is run with the changes.

- In advanced auto-partitioning flow, the compiler aims to find a suitable number of partitions of a design based on the resources available and the netlist at the end of the precompiler after all the transformations are completed. Unlike `autoPart`, there is no partition group file created at the end of partitioning. Use the following compiler option to enable advanced auto-partitioning:

  ```
  compilerOption -add {advancedPart ON | OFF | <N>}
  ```

  For syntax details and more information on `advancedPart` compiler option, see *User Data Commands* chapter in *VXE Command Reference Manual*.

  Note that, when both `autoPart` and `advancedPart` options are used together, `advancedPart` overrides `autoPart`.

  Also, when `advancedPart` option is set to `ON`, there must at least be four full boards or 32 domains available in Palladium Z1.

If none of these methods improve performance, you must try compiling by turning off PPC.

Modular Compilation in ICE Mode

Modular Compilation (MC) enables distribution of the design compilation steps to a compute farm. The biggest advantage of using modular compilation is that it provides a scalable compile technology with sub-linear compile turn around time with respect to design size. Also, MC requires less memory on each host to support huge designs.

To use this type of distributed compilation, familiarity with the PPC (Parallel Partition Compiler) flow is a must. In modular compilation, a new command called `diCompile` command controls distributed compilation. This new command delegates all intensive tasks to a compute farm.

Note: See "Support for Compute Farm in Modular Compilation" section for more information.

In modular compilation, you can split your design compilation into compilation **buckets** in parallel by choosing design modules that must go into each bucket. Ensure you select design modules in a way (perhaps using gate count information) such that the resulting buckets are balanced in terms of their size. Each user-defined module is automatically considered as a bucket by the tool.

These user-defined modules (**buckets**) are internally numbered as `bucket 1...bucket N`. Note that, `bucket 0` is a unique bucket that contains design top-level module and any design component that is not part of any other bucket. Buckets constitute modules. Specify module names on separate lines in `bucketModules.lst` file.

© 2021 All Rights Reserved.

Note: You can only specify module names in `bucketModules.lst` file. No instance names are accepted.

Consider a design that has the following logical hierarchy:

```
DUT
  I1 (core1)
  I2 (core1)
  I3 (core2)
  I4 (core2)
```

In this example, there are 4 buckets that have been listed in `bucketModules.lst` file using the following module names:

```
core1
core2
```

Thus, the design has five buckets:

- Top bucket
- Two buckets for `core1` module
- Two buckets for `core2` module

Modular compilation in ICE mode starts from using the generated QTDB libraries. However, existing xeCompile script requires following modification to enable MC in ICE mode:

```
design <lib> <cell>
emulatorConfiguration -add …
compilerOption -add {mode ice}
compilerOption -add {distributedLibraries ON}
<Any userData setting required, provided they are supported by MC>
setModularCompile
set MC(computeFarm) auto
diCompile
```

The base configuration used to compile the whole design is given through `emulatorConfiguration` command.

If `Boards *` wildcard is specified, it enables automatic emulator configuration to compute the number of boards required.

Note: See Creating BP File for Designs with Target section for more information.

Ensure all `iScopedNets` commands have been applied before loading the design and switching to modular compilation.

Note: See Connecting Nets Using iScopedNets Program section for more information.

© 2021 All Rights Reserved.

Architecture of Design Directory for Enabling Modular Compilation

There is a directory named `DISTRIBUTED` that is created at the very start of modular compilation. This directory has a subdirectory for each bucket, named after its number. For example, `DISTRIBUTED/bucket 0`, `DISTRIBUTED/bucket 1` and so on). Note that, bucket numbering is not based on the order of the listed modules in `bucketModules.lst` file but varies based on the design. The QTDB directory at the top contains the `ud` directory with userdata as well as all design libraries generated by `designImport` command. The distributed compile driver (`diCompile`) links those libraries to each `DISTRIBUTED/x/QTDB` directory.

Support for Compute Farm in Modular Compilation

All distributed operations are run on a third-party managed compute farm software.Presently, there are two types of farm management software supported for modular compilation. They

© 2021 All Rights Reserved.

are LSF and NC. Other farms could also be supported through custom farm capability, but it needs to be provided and maintained by you.

Instead of relying on `LSF_options` file as is done in non-modular compilation, modular compilation relies on a JSON configuration file named `computeFarm.json` that must be accessible in the design directory.

Refer to Compute Farm Control Mechanism section for information about custom compute farm and examples of `computeFarm.json` configuration file created using the JSON syntax.

Modular compilation requires all bucket jobs to be able to start in the farm approximately at the same time because the jobs talk to one another to orchestrate the compile. To support this, the farm must have the capability to reserve as many hosts as buckets, and wait for all those hosts to be available before all the jobs start to ensure that there are no bucket jobs waiting to start. Thereby, you must provide the required farm options for buckets.

`diCompile` uses `xe.msg` logfile to dump log messages. Standard output (`stdout`) for each bucket job is available from the top directory in `tmp/reportFromBucket<N>.log`. Also, depending on farm capability, `diCompile` monitors jobs, searches for farm-related issues and reports them appropriately.

New Commands to Enable Modular Compilation (MC)

Following commands and option are supported to enable MC:

- `compilerOption -add {distributedLibraries ON}`: This compiler option enables modular compilation.
- `setModularCompile`: This command enables the MC specific switch. It disables the `compile` and `precompile` commands and enables the new `diCompile` (`di` stands for distributed) command. It also enables you to see errors prior to running `diCompile` command. Ensure you set all userdata commands before running this command in ICE flow.
- `diCompile`: This command enables distributed compilation as per the bucketing rule defined in `bucketModules.lst` file.

Note: Switching back to older flows cannot be supported unless all generated data is deleted by running `xeCompile -clean` command in the Linux shell.

Following are the prerequisites for running `diCompile` command:

- `setModularCompile` command must be invoked before `diCompile` command.
- User-defined bucketing rules must be provided through the `bucketModules.lst` file.

© 2021 All Rights Reserved.

- In ICE mode, the design must be fully loaded along with all the userdata.

For information on `diCompile` command options and the list of command changes or commands unsupported in modular compilation, see *VXE Command Reference Manual*.

Limitations of Modular Compilation in ICE Mode

- Modular compilation does not support compilation of designs in 1x mode.
- Designs with UPF, CPF, and Hardware-Assisted Weighted Toggle Count (HW-WTC) features are unsupported in modular compilation.
- Modular compilation does not support running `precompile` separately from `compile`.
- `compileFind` is unsupported in modular compilation.
- Modular compilation does not support the commands listed in the Database Commands chapter.
- `partitionGroup`, `partitionAssign` commands are disabled in modular compilation as MC uses automatic PPC partitioning.
- Compilation aborts if combinational loops involving more than one bucket are detected before scheduling. Such cases are reported in advance, so that the loop can be broken manually or the bucketing strategy modified as required.

  Note: User must avoid creating cross-module loops when specifying bucket modules. Such loops typically occur due to combinational assignments/forces between I/Os of different modules. In some cases, the presence of the loop between bucket modules may not be obvious until the issue is notified by the ET Compiler. In which case, `getLoopDesignNetNames` command must be used to gather a detailed report from the compiler message.
- Modular compilation does not support designs with ATB constructs.
- Modular compilation does not support RTL database.
- Modular compilation does not support external-netlist-flow-based designs.

For information on using modular compilation in IXCOM mode, refer to Executing Non-Synthesizable Code in the Emulator section in Compiling Designs for Simulation Acceleration Using IXCOM chapter.

To debug modular compiled designs, refer to Debugging Designs with Modular Compilation section in Design Debugging Processes chapter.

© 2021 All Rights Reserved.

Viewing Compile Results

xeCompile compiles a design, and creates the download files to be used by xeDebug. These files configure the Palladium Z1 system for the emulation modes (Vector Debug or ICE/STB).

Also, when a compile session is started, xeCompile makes a main log named `xeCompile.log` in the design directory. This log file contains an exact copy of the terminal or console window. This file is incremental, that is, multiple compile sessions get sequentially recorded within the `xeCompile.log` file.

xeCompile also generates a log file named `xe.msg` that supports recursive logging, that is, nested command calls are also logged in the file. The `xe.msg` file contains a complete log of all previous compile and run-time sessions. This file includes the output displayed on the screen (same as `xeCompile.log`), output from the internal calls that are not displayed on the screen, and logging from the XEL commands that otherwise are not recorded in the console log file.

The `xe.msg` file is a superset of `xeCompile.log`. It is also incremental like the `xeCompile.log` file. Therefore, many commands are logged twice; first as an echo to the screen and second from inside the commands. Echoes are shown with a prompt in upper case, like on the screen, and logging from inside commands are shown with the prompt in lower case.

xeCompile generates another log file named `xeCompile.key` that contains all the commands typed by user. This file contains only the commands. It is a record of the compile session, and if sourced, produces exactly the same results as the original session. However, the `xeCompile.key` log file cannot record correctly some inherent limitations of the technology, such as effects of timing.

Using Compiler Options for Better Compile-Time Performance

Consider the following possible user-actions to ensure a better compile-time performance:

- User can use the `compilerOption -add {compilerEffort LOW|HIGH}` command to fine-tune the compile. The default option is `HIGH`, which means that compiler produces the best quality results (lower step count). If `compilerEffort` is set to `LOW`, the compilation process is faster; however, the emulation might run slower or might require a larger hardware configuration.

  For a very few unusual designs, the `compilerEffort HIGH` setting could actually cause the design to require more domains than the `compilerEffort LOW` setting. There is no easy way to characterize designs that experience this problem. Compiling a design several times, using both the `compilerEffort LOW` and `compilerEffort`

© 2021 All Rights Reserved.

HIGH settings for compilerOption can be used to determine if a specific design is one of these exceptional cases.

- Use the compileFind XEL command with different options (such as -first, -best, -opt, -all, and -lsf) to specify the conditions for finding the most appropriate compile run. For detailed information about each option, refer to the description of compileFind command in *Compile-time Commands in ICE Mode* chapter of the *VXE Command Reference Manual*.
  - The -first option tries to find the first passing compile, using the specified parameters, such as -max_trials <*n*> and -max_steps <*n*>.
  - The -best option tries to find the best compile, using the specified parameters. Log messages for critical path information are placed in the file ./tmp/et3compile.msg
  - The -opt option enables to explore the effects of changing the optimization parameters on the compile success rate, capacity, and step count. This option runs a compilation for the given number of trials for each pair of values defined by you for optimizationEffort and placementEffort.
  - The -all command runs multiple compiles with sequential seeds and logs the results. By using extra parameters, you can vary the number of compiles and seed to use, set step limits, and so on. The critical path information for the ith (passed) trial for the -all option is located in the ./tmp/et3pass2.<*i*>.msg file. For this option, the information is provided for each passed trial.
  - When using LSF,
    - The -lsf option is a fast parallel alternative to the -first, -best, and -all options performing trials sequentially. The -lsf option performs the required number of trials with different seeds on different machines in parallel using LSF and chooses the best trial (the one with the minimum step count).
    - When both -opt and -lsf options are present, it is a fast parallel alternative to the -opt command that performs the trials sequentially. It performs the required number of trials with different seeds and different combinations of optimization effort and placement effort on different machines in parallel using LSF and chooses the best trial (the one with the minimum step count).

    For detailed description on use of LSF in VXE, refer to Using LSF to Distribute Compiles with Different Seeds among Different Computers.
  - As manual fine-tuning to find the best value for memory utilization can be cumbersome, use the -mem option of the compileFind command. This enables

© 2021 All Rights Reserved.

you to explore the effects of changing the memory utilization on results such as the compile success rate, capacity, and step count.

The `-mem` option executes a number of compiles for each memory utilization value, using user-defined decrements, starting from a user-defined maximum value (100, by default), down to a minimum value (50, by default). Each trial is recorded in the logfile. For each memory utilization value, the record will contain the value, number of the trial, seed value, compilation result (pass or fail), and step count. It will also report the success rate, average step count, and gate count, for each memory utilization value. It will also log a summary report similar to that given by running the `-all` option of the `compileFind` command. For each trial <*n*>, this summary report will report each memory utilization value in a file named `tmp/et3pass2.util`<*U*>`.`<*n*>`.msg`.

Note: Running the `-mem` option can be very time-consuming. Therefore, be aware of this trade-off.

- When a compile fails, user can repeat the command using different seed values until a successful compile is achieved.

  Seed value is a number between 0 and 2^{32} -1 used as input to the hardware compile. The seed value influences logic partitioning and affects the success rate of compilation. Several seed options are available. The default is to use a random seed, but user can also specify to the compiler to use a specific number as the seed value. This is generally useful if you have already run a compile, and want to tune the results.

- For all compiles subsequent to a successful compile, user can use the same seed value of the last passing run.

- Use parallel partition compiler to reduce the compilation time by partitioning the design into groups of module instances and running the parallelized phase of compilation as a set of parallel jobs, one per partition.

Using LSF to Distribute Compiles with Different Seeds among Different Computers

As design size increases, the time taken to execute the `compileFind` command with options such as `-first`, `-best`, `-all`, and `-opt` might become prohibitively long. To mitigate this problem, use the `-lsf` option of the `compileFind` command. This command option distributes the compiles with different seeds among different computers using LSF, a job scheduler.

Note: You should have LSF software installed. Before running these commands, run the setup script required for LSF. For details, you must consult the LSF administrator.

© 2021 All Rights Reserved.

To use a job scheduler other than LSF, you must implement the LSF commands required for the `compileFind -lsf` command as scripts. For details, refer to Using a Job Scheduler Other Than LSF.

When the `-lsf` option is executed, first, if necessary, the precompile process is performed in the design directory. Then, the first part of the back-end compile independent of the seed value is performed in the design directory. After that, the second part of the back-end compile dependent on the seed value is performed in parallel on different machines with different seed values.

Parallel trials are performed in subdirectories labeled as `ITER_1`, `ITER_2`,... of the working directory. The corresponding LSF outputs are placed in files named such as `report_from_trial_1`, `report_from_trial_2`,... in the working directory. The previous trial subdirectories, `ITER_<n>`, and LSF output files, `report_from_trial_<n>`, are moved to the `PREV_RESULTS` subdirectory.

By default, `N` random seeds S_1,... S_n,... S_N are generated. The default value of `N` is `30`.

For each `n` (that is, from `1` to `N`) the following command is synthesized and submitted to LSF:

```
bsub [<LSF_options>] -o report_from_trial_<n> remote_compile.sh <n> <design>
<parallel_compile_options> -msg et3pass2.<n>
```

where,

- `bsub` is a LSF command.
- *<LSF_options>* is a string retrieved from the `LSF_options` file that contains the options to the `bsub` command. The `LSF_options` file is located in the design directory. The contents of this file are described in LSF_options File.

  Note: The `LSF_options` file can also be saved in a location other than the design directory. To do so, before starting the compile process, define the full path to the `LSF_options` file using the `CDS_LSF_OPTIONS` environment variable. When you execute the `compileFind -lsf` command, the design directory is searched for the `LSF_options` file. If the file is not found in the design directory, the full path specified with the `CDS_LSF_OPTIONS` environment variable is searched for an `LSF_options` file.
- `-o report_from_trial_<n>` specifies to LSF to redirect the standard output of the job (together with error output) to the `report_from_trial_<n>` file.
- `remote_compile.sh` is a shell script (provided by Cadence).
- *<n>* is a trial number.
- *<design> ::= <top_lib_name> <top_cell_name>*.
- *<parallel_compile_options>* are described in Parallel Compile Options.

© 2021 All Rights Reserved.

- `-msg et3pass2.<n>` specifies to the Compiler to write messages to the `ITER_1/tmp/-msg et3pass2.<n>` file.

LSF will send the jobs to the appropriate machines.

When all trials have finished and at least one trial has succeeded, the best trial (the one with the minimum step count) is chosen. Consider the best trial to be `n`. Then, the following is done:

1. Files created under from `ITER_<n>/dbFiles` are moved to `dbFiles`, like this:

   ```
   mv ITER_<n>/dbFiles/<file> dbFiles
   ```

2. Message files (`tmp/*msg`) are from each ith subdirectory to the design directory:

   ```
   mv ITER_<n>/tmp/et3pass2.<n>.msg tmp
   mv ITER_<n>/tmp/qtet3pass2.<n>.msg tmp
   ```

3. The `xe.msg` file from each ith subdirectory is attached to the `xe.msg` file in the design directory:

   ```
   cat ITER_1/xe.msg >> xe.msg
   ```

4. In the design directory, `dbgPrepare` and `bitGen` are executed.

5. All the temporary subdirectories and files are removed:

   ```
   rm -rf ITER_1 ITER_2...
   rm -rf ITER_1.bak ITER_2.bak...
   rm report_from_trial_1 report_from_trial_2...
   rm report_from_trial_1.bak report_from_trial_2.bak...
   ```

LSF_options File

You can put options to the `bsub` LSF command in the `LSF_options` file. All the options must be defined in one line.

Note: The specified options are passed to LSF without any checking. This file can also be used for distributed synthesis. For details, refer to Executing Distributed Synthesis during RTL Synthesis.

To use an LSF queue for distributed synthesis, provide an `LSF_options` file of your choice using a command like `-q my_queue`.

Following are few usage examples:

- The following options instruct LSF to submit jobs only to machines with Linux operating system version `6.0` or higher and with memory at least `16000Mb`:

  ```
  -R "OSREL==6 && OSNAME==lnxOS && mem > 16000"
  ```


© 2021 All Rights Reserved.

- Specifying the following command line in the `LSF_options` file indicates that jobs should be submitted only to the machines (defined by the LSF administrator) having more than `1000MB` of available swap memory (`swp > 1000`), CPU utilization of less than 50% (`ut < 50`), and memory more than `16000MB` (`mem > 16000`):

  ```
  -R "swp > 1000 && ut < 50 && mem > 16000"
  ```

- The following options instruct LSF to submit jobs only to machines defined (by the LSF administrator) as having type `DEFAULT` and with memory at least `8000Mb`:

  ```
  -R "type == DEFAULT && mem > 8000"
  ```

Parallel Compile Options

The parallel compile options available with the `-lsf` option of the `compileFind` command are:

```
compileFind -lsf [-max_trials <number>] [-seed <number>] [-max_steps <number>] \
                 [-min_steps <number>] [-crit_paths <crit_paths>] [-etOnly] \
                 [-first] [-1pass] [-keep_all] [-rerun_best] [-compactDCC]
                 [-no_export] [-names_crit] [-link <filename>] [-verify] \
                 [-file <file>]
```

Options `[-1pass]`,`[-rerun_best]`,`[-compactDCC]`,`[-link <filename>]`,`[-file <file>]` are not supported for PPC.

The following parallel compile options are available when the `-lsf` option is specified with the `-opt` option:

```
compileFind -opt -lsf [-max_trials_each <number>] [-seed <number>] \
                      [-oE {<values>}] [-pE {<values>}] \
                      [-max_steps <number>] [-min_steps <number>] \
                      [-crit_paths <crit_paths>] [-etOnly] [-first] \
                      [-keep_all] [-names_crit] [-rerun_best] [-no_export] \
                      [-logfile <logfile>] [-verify] [-compactDCC] \
                      [-link <filename>] [-file <file>]
```

Though `-opt` option is supported for PPC, `[-first]`, `[-rerun_best]`, `[-logfile <logfile>]`, `[-compactDCC]`, `[-link <filename>]`, and `[-file <file>]` options are not supported with `-opt` option for PPC.

For detailed description of the syntax of both the commands, refer to the *Compiler Commands* chapter of the *VXE Command Reference Manual*.

Support for Network Computer Natively for Compilation

NC is natively supported in addition to LSF as LSF wrapper-based solution might not provide adequate performance on a Network Computer (NC) farm.

© 2021 All Rights Reserved.

Depending on the flow (regular or Parallel Partition Compiler) and the command (`compile` or `compileFind`), there are different ways to switch to NC mode. In regular compile flow, NC impacts `compileFind` only as the new *-nc* option is used instead of the *-lsf* option with the `compileFind` command. While in PPC flow, NC impacts both `compile` and `compileFind` commands.

To switch to NC mode in PPC flow, use the following PPC setting before starting `compile` or `compileFind`:

```
set PPC(useNativeNC) true
```

Note that in the PPC flow, even if `compileFind -lsf` is provided, `compileFind` natively uses NC if the PPC setting is set to *true*. Also if you do not provide the *-lsf* option, PPC flow uses NC natively for partition jobs.

When using `compileFind`, invoke `dbgPrepare` internally through `compile -dbgPrepareOnly`.

Note: `LSF_options` file works the same way for NC. Only when NC is unable to check the validity of directory paths for all hosts in the network, the *-D* option must be added to the `LSF_options` file.

Using a Job Scheduler Other Than LSF

Normally, the Parallel Partition Compiler (PPC) and `compileFind-lsf` command uses the `bsub`, `bjobs`, and `bkill` commands provided by LSF. PPC uses two additional commands namely, `bhist` and `lsid`. You can cause `compileFind -lsf` to use a job scheduler other than LSF. To do so, you must provide scripts that implement the functionality of `bsub`, `bjobs`, `bkill` in non-PPC and PPC mode, `bhist` and `lsid` in PPC mode only using an alternative job scheduler of your choice. Before starting xeCompile, set your UNIX search path so that your scripts are found. The compiler will then use the commands that you provide, instead of using the commands that LSF provides.

The scripts for the `bsub`, `bjobs`, and `bkill` commands that you provide must implement the equivalent LSF commands in the following ways:

- bsub

  The script for the `bsub` command should send a job to the job scheduler.

  xeCompile will call the script with the following arguments:

  ```
  bsub <LSF_options> -o report_from_trial_<n> remote_compile.sh
       <more_arguments>
  ```

  where *<LSF_options>* is a single-line content of the `LSF_options` file in the design directory (if that file exists).

© 2021 All Rights Reserved.

The script needs to do the following:

- Tell the job scheduler to run the `$QTHOME/share/vxe/bin/remote_compile.sh` script with *<more_arguments>* as arguments to the script.

  The script can assume `QTHOME` is already set correctly in the UNIX environment.

- Tell the job scheduler to send any messages for the job submission to the `report_from_trial_<n>` file.

- If `bsub` is called with *<LSF_options>*, pass these options to the job scheduler, using whatever format the job scheduler expects.

  Note: If the design directory contains a file named `LSF_options`, then xeCompile will pass the contents of this file (which should be a single line) as arguments to `bsub` each time `bsub` is called. The `LSF_options` file can contain options in the format expected by the alternative job scheduler, provided these options are on a single line.

- Get a unique number *<JobID>* from the job scheduler to identify the job.

- Echo a message similar to the following to `stdout`:

  ```
  Job <841258> is submitted.
  ```

  where, `841258` is the *<JobID>* you got from the job scheduler.

  Note: The message echoed to the `stdout` must include the *<JobID>* as a string of digits, surrounded immediately by `< >` without any spaces.

When using Parallel Partition Compiler, xeCompile will call the `bsub` script with the following arguments:

```
bsub <LSF_options> -o <path_to_output_file> <a_script_to_run>
<more_arguments>
```

Additional `bsub` command options are required for using Parallel Partition Compiler:

- `-o <logfile>`: Indicates output that must be sent to a file named *<logfile>*.

- `-E <script>`: Here, *<script>* is a shell script which must be executed on the host chosen by the farm scheduler before the actual job starts. If the script status is "0", the scheduler can start the job on the host, otherwise, it has to find another host. This pre-execution feature is used to avoid hosts which may cause problem (that is, where the compilation might fail).

- `-q <queue>`: Indicates bsub must use the queue given as argument instead of the default one.

© 2021 All Rights Reserved.

- `-Is`: Indicates this is an interactive job and `bsub` should wait for the command to finish. In addition, `bsub` result should be the output generated by the command.

Expected Output:

Except when `-Is` is used, `bsub` is expected to echo on `stdout` something similar to `Job <jobId>`.

- bjobs

The script for the `bjobs` command should report the status of the submitted jobs.

xeCompile will call the script periodically (every five seconds) with no arguments. The `bjobs` command should echo on `stdout` a table like the following:

```
JOBID USER STAT
206150 NA DONE
206156 NA RUN
```

The table has a heading line (which is ignored), followed by status lines that give the status of jobs. Each status line should have three items separated by whitespace. The items are: `JobID`, a second item (which is ignored), and a status field whose value is `PEND`, `RUN`, `DONE`, `EXIT`, or `ZOMBI`.

For the `bsub` provided by LSF, the meaning of the status field is as follows:

- `PEND`: The job has not started yet.
- `RUN`: The job is running.
- `DONE`: From the perspective of the job scheduler, the job has completed successfully (even if the compile trial failed).
- `EXIT` and `ZOMBI`: The job scheduler encountered a problem trying to run the job.

The only essential requirement for `bjobs` is that when a job has not finished yet, `bjobs` must not show the job as `DONE`, `EXIT`, or `ZOMBI`, and when a job has finished, at least one of the subsequent periodic calls to `bjobs` must show the status of that job as `DONE`, `EXIT`, or `ZOMBI`.

xeCompile does not distinguish between the `DONE`, `EXIT`, and `ZOMBI` status, except that it prints a slightly different message for a trial that ends with `DONE`, as compared to `EXIT` or `ZOMBI`. xeCompile ignores status lines of any jobs not submitted by the current compile.

Additional `bjobs` command options supported for using Parallel Partition Compiler:

- `-noheader`: Option used to indicate job heading must not be shown.

© 2021 All Rights Reserved.

- `-o <stat |exec_host | submit_time | start_time | finish_time | run_time>`: Select which of the following information must be returned by `bjob`.

  `stat` stands for status.

  `exec_host` stands for the host where the job is/was running.

  `submit_time`, `start_time`, `finish_time` stands for the time of day when the job was submitted, the job started, the job finished.

Note: Time of day format should have 3 fields, such as “Apr 11 10:43”.

  `run_time` stands for the elapsed time in seconds taken by the job.

- `<jobId>`: Option used to denote the job ID returned by `bsub`.

Expected Output:

We expect one line with the following fields:

First field is the recognized status field: `PEND`, `RUN`, `DONE`, `EXIT`. The meaning is the same as that for non-PPC `compileFind`.

Second field is for the host name: Any string should be fine.

Third, fourth, and fifth field: A single line with time of the day information for the given job.

Sixth field: Any string of digit indicating the elapsed time in seconds used by the job.

Example:

```
DONE my_host Jun  4 16:23 Jun  4 16:24 Jun  4 16:24 1 second(s)
```

- bhist

The script for the `bhist` command displays some information about the jobs given as argument. This command is only supported with PPC.

Following `bhist` command options are supported for Parallel Partition Compiler:

`<list of jobIds>`: Option used to denote the job IDs that were returned by `bsub`.

Expected Output:

The second line must show job headers, with one of them being `PEND`. Then for each job, there should be a line with a `jobID` and a list of information. In the column identified by `PEND` we expect to find the pending time for each job. Each item should be separated by white spaces.

Example:

© 2021 All Rights Reserved.

```
Summary of time in seconds spent in various states:
JOBID   USER    JOB_NAME  PEND   TOTAL
9757751 aUser whoami    1       6
9757752 aUser *TIONS/1  5       6
```

- bkill

  The script for the `bkill` command should kill the jobs specified by its arguments.

  The `compileFind -lsf` command generates a file named `killAllLSFjobs` that contains a line similar to the following:

  ```
  bkill 287926 287927 287928 287929 287931
  ```

  If the user (of xeDebug) executes the `killAllLSFjobs` file, `killAllLSFjobs` will call `bkill` with a sequence of zero or more *<JobIDs>* as arguments. The script should tell the job scheduler to kill the specified jobs.

  Following `bkill` command options are supported for Parallel Partition Compiler:

  <jobId>: Option used to denote the job ID that was returned by `bsub`.

  There is no expected output but the job should get killed.

- `lsid`

  The script for `lsid` command provides information about the LSF system. This command is only supported with PPC.The output does not matter as long as the command status is ”0” to indicate the farm is up and running. Otherwise, an error message would be issued by the command which would be copied to `xeCompile` log before aborting the compilation.

Improving Compilation Time

`dbgPrepare` is a process invoked during compilation. It has two sub-processes: pass1 and pass2. pass1 is run in parallel with the compiler. pass2 is run after pass1 finishes execution. The processes are run as follows:

- pass 1 is run with multiple threads: The pass1 process consumes longer time as compared to pass2 and creating swfvDB takes significant time in pass1. Therefore, pass1 is run with multiple threads, which enhances the performance of dbgPrepare processing. This behavior is enabled by default and makes compilation faster. You can also enable running of pass1 without multiple threads, if required, using the following environment variable:

  ```
  setenv dbgPre_ST
  ```

- pass 1 and pass 2 are run concurrently where pass2 is started earlier without waiting for pass1 to end. You can disable this behavior using the following environment variable:

© 2021 All Rights Reserved.

```
setenv XE_DBGP_PARALLEL FALSE
```

For information on various settings for the environment variable, refer to the Environment Variables Required in Specific Cases in chapter Setting Up the Environment for Palladium Z1.

Limitations:

Running pass1 and pass2 simultaneously requires more memory. Both pass1 and pass2 need to open Palladium database. If LSF is available and pass1 runs in another host, memory usage does not increase in the main host. In the absence of LSF, however, the main host needs more memory to run both pass1 and pass2.

Using Multithreading Feature for Compilation

The following table lists the different multithreaded programs available in different phases of compilation:

Program	Default maximum number of threads	Local	Network	Ways to Control
`hdlSynthesize` command	1	Yes	Yes	For network-level multiprocessing, specify the `hdlSynthesize -parallel` command to use the `parallel.host` file. For details, refer to Using parallel.host File. For local-level multiprocessing, specify the `hdlSynthesize -parallel<number>` command. For details, refer to Using Multiple CPUs of the Same Workstation.

© 2021 All Rights Reserved.

Program	Default maximum number of threads	Local	Network	Ways to Control
Design import	32 **Note:** The default number can be changed using the `IMPORT_MAX_PARALLEL` environment variable that has an absolute maximum limit of 512 threads. If a larger number is specified, it is reduced to 512.	Yes	Yes	Use the `importOption {parallel on}` command. For details, refer to Using Parallel Import. **Note:** In addition, the following special-purpose option can be used to disable automatic creation of BCUs: `importOption {parallel [on|off]}` To script a set of parallel import operations acting on the same database (QTDB), and more than one such parallel invocation tries to create the same library (as can happen with automatic BCU creation), things do not work correctly. This option is obsolete in all except such legacy application.
`compileFind -lsf` command	No restrictions on the number of the processes	No	Yes	For details, refer to Using LSF to Distribute Compiles with Different Seeds among Different Computers.

© 2021 All Rights Reserved.

Program	Default maximum number of threads	Local	Network	Ways to Control
`compileFind -opt -lsf` command	No restrictions on the number of the processes	No	Yes	For details, refer to Using LSF to Distribute Compiles with Different Seeds among Different Computers.

Synthesizing and Compiling ALUs

The Palladium Z1 hardware contains Arithmetic Logic Units (ALUs) that the HDL-ICE Compiler can use to synthesize computation-intensive parts of the design such as, the following operations:

- Arithmetic operations, such as addition, subtraction, negation, increment, decrement, multiplication, and division with both signed and unsigned numbers
- Comparison operations, such as `A > B`, `A < B`, and `A==B`

Most operations are supported with unlimited operand width. But the maximum operand width is 256 for multiplication, 64 for signed division, and 63 for unsigned division.

Synthesizing a certain part of the netlist on to ALUs can potentially reduce the step count and also improve the emulator capacity. For example, when a 32-bit adder is mapped to an ALU, only one primitive is required, as compared to requiring about 64 primitives when implemented as boxes or look-up tables. However, the number of ALUs in the hardware is limited, and if too many arithmetic operations are mapped to ALUs, the performance might be worse than if ALUs had not been used at all. Always use an option to apply ALU synthesis selectively, by operand size or by module.

To switch on ALU synthesis, use the `-alu` option:

- `vaelab` syntax:

  ```
  vaelab -alu
  ```

- `hdlSynthesize` syntax:

  ```
  hdlSynthesize -alu
  ```

In the gate-level netlist, each ALU instruction appears as an instance of a `qtref` cell `Q_ALU_I`. A sequence of ALU instructions using the same ALU, passing intermediate values

© 2021 All Rights Reserved.

through registers, are linked together using `SYNC_IN/SYNC_OUT` ports. This is called an ALU chain.

For each ALU chain, the HDL-ICE Compiler also produces a Verilog simulation model, along with instances of `Q_ALU_I`, so that the behavior of the netlists can be verified in a simulator, such as Xcelium, or in a formal verification tool, such as Conformal.

Controlling ALU Utilization During Synthesis

There are only 384 physical ALUs on a Palladium Z1 board. Each ALU has a limited number of registers and routing resources. Thus, having a large number of ALU primitives in a netlist can become a performance bottleneck and slow down the entire emulation cycle.

Controlling ALU Synthesis by Operand Size

Use the synthesis option `-MinALUSize <arg>` with the `vaelab` command or `-alu<arg>` option with the `hdlSynthesize` command to control ALU utilization by operand size.

- `vaelab` Syntax:

  ```
  vaelab -MinALUSize <arg>
  ```

- `hdlsynthesize` Syntax:

  ```
  hdlSynthesize -alu<arg>
  ```

The `<arg>` values specify the threshold values for the ALU mapping. For example, `-MinALUSize 40` option instructs the HDL-ICE Compiler to synthesize only those additions to the ALUs that have all the 40 bits non-constant for both of the operands; an addition smaller than 40 bits is flattened into gates without using ALUs. Meanwhile, `-MinALUSize 10` encourages the HDL-ICE Compiler to map more additions, including small adders like a 10-bit adder, to the ALUs.

Note the following points about the value of `<arg>`:

- The value of `<arg>` should be between `2` to `1000`.
- Fewer operations are mapped to ALU as `<arg>` becomes bigger.
- The value of `<arg>` represents the minimum threshold of the average number of non-constant inputs when an addition operation is mapped to ALU. Default threshold value is `23`.
- Different operations can interpret this threshold in different ways.

© 2021 All Rights Reserved.

By default, ALU mapping is switched off. Using either of the `-alu` or `-MinALUSize` command options enables ALU mapping. Thus, when the `-MinALUSize <arg>` option is present with the `vaelab` command, `-alu` option is optional for `vaelab`.

The compiler maps the ALU operations to physical ALUs. If there are not enough physical ALUs, the compiler can convert some ALU operations to gates.

Controlling ALU Synthesis by Module

To enable ALU synthesis for modules whose names match the specified patterns, use one of the following commands:

```
vaelab -skipModule "+enableALU <pattern1>...<patternN>"
hdlSkipModule +enableALU <pattern1>...<patternN>
```

This specification is in addition to the enabling of ALU synthesis by other options. Do not use the `-alu` option of the `vaelab` command or the `hdlSynthesize` command together with the `+enableALU` option, because the `-alu` option enables ALU synthesis in all modules, making `+enableALU` redundant.

Consider the following examples for controlling ALU synthesis by module. Note that the output netlist in all examples is written to `top.vg`.

Example 1

To synthesize the design with top-level module `top`, enabling ALU synthesis in `mod1` module and in all modules whose names start with `alu2`, such as `alu2` and `alu2a`, use:

Either the following Unix command:

```
vaelab top -skipModule "+enableALU mod1 mod2*" -OutputVlog top.vg
```

Or the following `xeCompile` commands:

```
hdlOutputFile -add -f verilog top.vg
hdlSkipModule +enableALU mod1 mod2*
hdlSynthesize top
```

To selectively disable ALU synthesis for modules whose names match the specified patterns, and for which ALU synthesis has been enabled by other options, use:

Either the following `vaelab` command:

```
vaelab -skipModule "+disableALU <pattern1>...<patternN>"
```

Or the following `xeCompile` command:

```
hdlSkipModule +disableALU <pattern1> ... <patternN>
```

Example 2

© 2021 All Rights Reserved.

To synthesize the design with top-level module `top` for which ALU synthesis is enabled for all modules in the design except `mod1` and the modules whose name starts with `mod2`, such as `mod2` or `mod2a`, use:

Either the following Unix command:

```
vaelab top -alu -skipModule "+disableALU mod1 mod2*" -OutputVlog top.vg
```

Or the following `xeCompile` commands:

```
hdlOutputFile -add -f verilog top.vg
hdlSkipModule +disableALU mod1 mod2*
hdlSynthesize -alu top
```

Example 3

To synthesize the design with top-level module `top` for which ALU synthesis is enabled for operations that meet the size threshold `64`, which can be set by the `-MinALUSize` or `-alu64`, in all modules except `mod1` and the modules whose name starts with `mod2`, use:

Either the following Unix command:

```
vaelab top -MinALUSize 64 -skipModule "+disableALU mod1 mod2*" \
           -OutputVlog top.vg
```

Or the following `xeCompile` commands:

```
hdlOutputFile -add -f verilog top.vg
hdlSkipModule +disableALU mod1 mod2
hdlSynthesize -alu64 top
```

Controlling ALU-to-Gates Transformations at Compile Time

During synthesis, the chains of ALU instructions represented by `ALU_I` primitives are synthesized. During compilation, the compiler schedules the ALU chains to physical ALUs. When feasible, multiple ALU chains can be scheduled to the same physical ALU. Most of the ALU transformations are performed automatically, without any user intervention.

One of the possible ALU transformations is conversion of some ALU chains to gates. It can be necessary because HDL synthesis is performed on the hierarchical design representation when the design capacity and hardware configuration are not known.

There are only 48 physical ALUs on a Palladium Z1 module. As a result, too many ALU primitives in a netlist can cause a performance bottleneck, and slow down the entire schedule or even exceed the capacity of the available Palladium Z1 modules. The compiler can map ALU instructions to gates to reduce the contention on ALU resources.

By default, based on factors, such as hardware configuration and design capacity, the compiler automatically decides how many and which ALU chains should be transformed to

© 2021 All Rights Reserved.

gates. You can modify this ALU-to-gates transformation by using the `aluTransform` User Data command.

Note: ALU-to-gates transformations are performed before the netlist-optimization step. Therefore, 100% utilization before netlist optimization might become 80% utilization (or even lesser) after netlist optimization.

Following are the various options for the `aluTransform` command that help you control ALU-to-gates transformations:

Command	Description
`-add{* alu2gates OFF}`	No ALU-to-gates transformations are performed.
`-add{* alu2gates ON}`	Restores the default behavior for ALU-to-gates transformations.
`-rm{* alu2gates}`	Restores the default behavior for ALU-to-gates transformations.
`-add{* aluMaxUtilization <number>}` where `<number>` is a percentage	Modifies the default maximum utilization percentage (100). Minimum value for `<number>` is `0` (no transformation is performed), and maximum value is `100`.
`-rm {* aluMaxUtilization}`	Restores the default value for `aluMaxUtilization`.

Note: The asterisk `*` in these commands is a placeholder. This command structure is similar to the structure of the `memoryTransform` command. Refer to the *User Data Commands* chapter of the *VXE Command Reference Manual* for more details.

Caution

User Data from the above commands is stored and read from the `dbFiles/<top>.aluRules` ***file. Do not edit this file.***

Information on the available and used physical ALU can be accessed from the back-end compilation log file (for example, `tmp/et3compile.msg`) and in the file `dbFiles/<topcell>.memplacement`.

© 2021 All Rights Reserved.

Connecting Hierarchical Nets

You can connect the hierarchical nets to either power or ground, or to another net using the `hierAssign` command. The `hierAssign` command inserts buffers at a specified hierarchical location and removes all drivers of the net before connecting the net to either power or ground, or to another net.

For complete syntax of the `hierAssign` command, refer to the description of the `hierAssign` command in the *User Data Commands* chapter of the *VXE Command Reference Manual*.

The following example shows how the buffers are inserted to separate the nets.

`top.A`, `top.u1.A`, and `top.u2.A` are three names for the same net, and the net's only driver is the outside instance `top.u2`, as shown in the diagram below:

To tie `top.u2.A` to ground, while leaving `top.A` and `top.u1.A` unaffected cannot be done using the `tieNet` command. A `tieNet` command with any of these names affects `top.A`, `top.u1.A` and `top.u2.A`, since these are three names for the same net.

© 2021 All Rights Reserved.

The command `hierAssign -add {top.u2.A 0}` first inserts a buffer separating `top.u2.A` from the rest of the net as shown in the following diagram:

The buffer is then disconnected from `top.u2.A`, and `top.u2.A` is tied to ground as shown in the following diagram:

The tying of `top.u2.A` to the ground does not affect `top.A` or `top.u1.A`.

© 2021
All Rights Reserved.

In general, the command `hierAssign -add {<instance>.<net> <value> | <source_net>}` inserts buffers as follows:

- If <*instance*>.<*net*> connects to a port of *<instance>*, and there is a driver on only one side of the port (that is, either inside *<instance>* or outside *<instance>*), a buffer is inserted at the port, pointing from the side where the driver is located, towards the other side. If the port is driven both inside and outside the specified *<instance>*, the port is considered bidirectional and no buffer is inserted at the port.
- If *<instance>*.*<net>* connects to a port of a sub-instance of *<instance>* (for example, it connects to *<instance>*.u1.A), and the driver is only on one side of the port (that is, either inside *<instance>*.u1 or outside *<instance>*.u1), then a buffer is inserted at the port, pointing from the side where the driver is located, towards the other side. If the port is driven both inside and outside *<instance>*.u1, the port is considered bidirectional and no buffer is inserted at the port.

Finding Faults Using ModelChecker

If you suspect that a design has run-time issues, it is recommended that you use the Compiler Model Checker, or the ModelChecker tool. ModelChecker can help in quickly finding issues, thereby, saving time in debugging the design. At the end of the compilation process, a final <*design*>.proto file is generated. The .proto file, basically, contains a netlist of boxes. Each box has an associated logic function and the inputs and outputs of each box are partitioned to various emulation processors data storage (emulation process + step). The compiler needs to properly schedule the logic function of the box at a specific step in a specific emulation processor. Scheduling of the logic function is quite complex, and can have some errors. These errors or faults might not be detected at run time if that portion of the model is not exercised, or is not sensitive to the data stimulus. But, certain faults might lead to problems during simulation.

For such situations, it is recommended that you use the ModelChecker tool to detect the compiler faults before these cause run-time failures. The ModelChecker is a static model checker that runs a series of tests to compare the final .proto file against the instruction sequences of the emulation processors. The ModelChecker identifies the inconsistencies between the model created by the compiler by comparing a statically known set of conditions and properties of the current emulator configuration. ModelChecker cannot prove that the model is correct, but it is close, assuming a good set of tests are covered. ModelChecker detects the problems, and provides the signal information to cross reference it to the run-time problems. The following sections describe how ModelChecker can be invoked, the kind of tests that are performed, and how the information can be interpreted.

© 2021 All Rights Reserved.

Invoking the ModelChecker

The ModelChecker program can be invoked in two different ways – either as a standalone executable that can be invoked on the final design database, or automatically from the compiler. Both methods have the same semantics (arguments).

- Invoking `ModelChecker` as a standalone executable:

  Type the following command on the UNIX prompt:

  ```
  ModelChecker <design_name> [options]
  ```

  Here, the `<design_name>` argument refers to the top cell name used for naming the `.proto` file.

  The `[options]` argument can have values `-write_pgm`, `-write_fnets`, or `-omit [<check_type>]`.

- Invoking the ModelChecker program as a part of the compilation process:

  Specify the option with the `compilerOption` command:

  ```
  compilerOption -add {ModelChecker OFF | ON}
  ```

  Corresponding to the `[options]` argument that you specify with the `ModelChecker` command as a standalone utility, you can specify the required options with the `compilerOption` command:

  ```
  compilerOption -add {write_pgm OFF | ON}
  compilerOption -add {write_fnets OFF | ON}
  compilerOption -add {ModelCheckOmit "LOGIC LATCH XBIO DBIO MEM"}
  ```

For command description and syntactical details, refer to the *VXE Command Reference Manual*: *Programs and Utilities* chapter for `ModelChecker` command, and *User Data Commands* chapter for `compilerOption` command.

Important

For large designs, ModelChecker run-time amounts to a fraction of the compilation time. But for small designs, invoking the ModelChecker program at compile time using the `compilerOption` command options could almost double the running time for small designs. Therefore, it is recommended that ModelChecker should be invoked as a standalone application to avoid increasing compilation time.

Types of Checks Performed by ModelChecker

This section outlines the following basic ModelChecker check types performed:

- LOGIC Check Type

© 2021 All Rights Reserved.

- LATCH Check Type
- XBIO Check Type
- DBIO Check Type
- MEM Check Type

Most of the check types are based on the BOXes found in the final `<design_name>.proto` file. By default, all check types are executed. However, you can turn off the check types not required by using either of the following commands:

- `ModelChecker <design_name> -omit [<check_type>]`
- `compilerOption -add {ModelCheckOmit "[<check_type>] [<check_type>] ..."}`

For all check types, when ModelChecker finds a problem, a message of the following format is displayed to stderr and recorded in the `./tmp/ModelChecker.log` file:

```
WARNING (ModelChecker): <check_type> failed for BOX <box number>: <User
Information>, INTERNAL: <Engineering Information>
```

Here, `<User Information>` includes a net name found in the post-synthesized netlist, such as signal name or memory instance name (depending on the check type).

LOGIC Check Type

It compares the actual and expected operands and logic functions. Ensures that the operands are properly sequenced to be computed in the same cycle, not the previous cycle.

This check compares the expected operands and logic functions to the actual operands and logic functions. The expected operands and logic functions are obtained by traversing the BOX netlist (agnostic of the actual emulation hardware or instructions). The actual operands and logic functions are obtained by traversing the emulation processor instructions.

LATCH Check Type

It is similar to the LOGIC checks, except that with LATCH check type, the *d*-pin data is scheduled in the previous major cycle.

Note: When the LATCH check type is specified, all types of latches including regular latches, combo-cycle (CC) updating latches, and scattered probes chain latches are checked.

© 2021 All Rights Reserved.

XBIO Check Type

It performs the checks on XBIB and XBOB box types. The XBIO boxes are the primary inputs and outputs of the design that drive or load target interfaces.

The expected operand of the XBOB box is the location of the data input. When traversing the instructions to compute the expected operands, ensure that the operand is computed in the current cycle.

DBIO Check Type

It performs the checks on DBIB, DBOB, and HCB box types for SA interface. These checks are identical to XBIO checks. The DBIO box checks ensure that primary inputs are properly stabilized and routed to their corresponding destination (load).

MEM Check Type

It performs the checks on XMPR and XMPW box types for both INTRAM and EXTRAM.

XMPW checks make sure that all address, data, and byte_enable operands arrive at the same time as the memory is enabled in the same major cycle. Proper slicing and constant propagation (for smaller memories) are also checked on the INTRAM and EXTRAM interfaces. In addition to regular memory writes, assembly-mode-store operations are checked as well.

XMPR checks are identical to XMPW checks with respect to setting up the address bus. However, a latency must be considered on the data read bus. XMPR checks also cover disassembly read operations.

For all types of RAM checks, proper spacing is also checked to make sure that operations to the same bank are not violating pending or previous requests.

Output Logs of ModelChecker

ModelChecker warnings are displayed to the standard error as well as logged in the `./tmp/ModelChecker.log` file.

Note: These log files are maintained separately because each pass through the ModelChecker program operates on a completely different compiled `<design_name>.proto` file.

When the option to write `fnets` is specified, the `et3fnets` names are displayed in the standard output console and written to the `./tmp/et3compile.msg` file.

© 2021 All Rights Reserved.

When the option to write `pgm` is specified, the `et6wrpgm` processor program is saved to the `./dbFiles/<design_name>.et6pgm` file.

The `et3fnet` and `et6wrpgm` header information is also displayed in the standard output console. This header information contains the version and launch notification of the applications.

If the `compilerOption` userdata command for invoking the ModelChecker program is specified in an iterative compile process, such as while finding the best step count, the ModelChecker program is still invoked only once on the final compiled database that is generated after the compilation process is completed.

© 2021 All Rights Reserved.

10

Compiling and Running Designs with Memories

A memory is defined as a one-dimensional unpacked array of a variable with a one-dimensional packed datatype. In netlists, memory instances typically refer to elements that are not completely defined in either the netlist or the ASIC Library. This lack of definition leads to a blackbox description of these elements. Generally, a memory compiler generates ASIC memories, rather than providing a library definition for every possible memory configuration. Typically, memories are mapped into the xeCompile database during netlist import. While compiling designs with memories, ROM and RAM cell definitions created by the Netlist Importer can be generated either using the automatic memory process or manually.

Note: For the handling of memories in SA mode using IXCOM, also refer to Direct Access to DUT Memory Arrays and Registers in Chapter 16, “Compiling Designs for Simulation Acceleration Using IXCOM.”

ASIC vendors generally use a memory compiler to generate ASIC memories, rather than providing a library definition for every possible memory configuration. The same method is used in adding memory cells to the Palladium Z1 database.

Memories can be described in the following ways:

- Memory RTL files

  Memory RTL files are usually the best method for describing memories. Create a Memory RTL file written in Verilog or VHDL, and synthesize it using xeCompile. xeCompile automatically maps the memories to the corresponding Memory Primitives. xeCompile memory synthesis process is described in the Synthesizing Designs with Memories of Chapter 9, “Compiling Designs for In-Circuit Emulation with xeCompile.”

  You can use xeCompile to synthesize Memory RTL files even if you do not use it to synthesize the design itself.

- Memory Primitives

  The memory primitives are the lowest-level way to describe memories. These are also known as MPR (Memory Port Read) and MPW (Memory Port Write), and represent

© 2021 All Rights Reserved.

memory read and write ports, respectively. This approach works well for legacy designs that already describe memories using MPR and MPW primitives.

Note: Any memory which is not modeled as gates is actually modeled with MPR/MPW memory primitives. These primitives can be combined to create a memory with any number of read and write ports. However, these primitives can only be accelerated, not simulated, so a congruent simulation is not possible. Consult with the assigned Cadence AE before using these primitives.

In ICE mode, a maximum of 63 address bits are supported at compile time and at run time the limit of memory support is defined by the available emulator resources. In SA mode, the limit is 30 address bits.

Supported Memory File Formats

In emulation, the supported file formats are as follows:

- For ASCII:
  - PD ASCII (bin, oct, dec, hex)
  - readmemb and readmemh
  - NC_SIM
  - Denali memory format
- For binary:
  - memtran
  - raw
  - raw2

Transforming Memories

`et3compile` can transform design memories. The goals of memory transformations are:

- Fit all design memories in emulator memory for the given hardware configuration.
- Increase the probability of successful compilation.
- Decrease the emulation step count.

The following types of memory transformations can be performed:

© 2021 All Rights Reserved.

- Transform shallow memory instances into gate level models.
- Merge memory instances.
- Split memory instances with multiple read ports.
- Squeeze memory instances with constant address pins.
- Perform memory compaction.
- Perform sparse memory modeling for memory instances that you have requested.

You can force transformation of memory instances and/or prevent transformation of other memory instances by use of the following `memoryTransform` command:

```
memoryTransform -add {<memory> <parameter_name> <parameter_value>}
```

To remove the memory instance name from User Data, use the following command:

```
memoryTransform -rm { <memory> [<parameter_name>] }
```

A list of transformed memories can be found in the `dbFiles/<topcell>.memTransformList` file.

Some memory transformations are compatible (that is, can be performed on the same memory instance), and some are not. There are the following rules:

- A shallow memory instance represented as a gate array cannot (naturally) participate in other memory transformations.
- A memory instance that you have defined as sparse cannot participate in other memory transformations.
- Memory splitting, memory squeezing and memory compaction are fully compatible transformations. For example, memory M can first be split into 3 chains (M=>{M1, M2, M3}), then each chain can be squeezed (M1=>M1.1, M2=>M2.1, M3=>M3.1), and finally chains M1.1 and M3.1 can be compacted.

Transforming Shallow Memory Instances into Gate Arrays

Some designs can contain big numbers of “shallow” memory instances. Transformation of shallow memories into gate-level models can be beneficial due to the following two reasons:

- It can allow to fit a design in the given hardware configuration if the bottleneck was a lack of memory capacity rather than gate capacity.
- It can decrease the step count.

© 2021 All Rights Reserved.

The decision on which memory instances should be transformed is based on capacity considerations and on pursuit of the minimal estimated emulation memory access.

Memory instances that you have explicitly defined and instrumentation memory instances are not transformed.

To prohibit transformation of a shallow memory into gate array, use the following memoryTransform command:

```
memoryTransform -add {<memory> SHALLOW off}
```

To restore the default behavior, use the following command:

```
memoryTransform -add {<memory> SHALLOW on}
```

To remove the shallow memory transformation of the specified memory instance, use the following command:

```
memoryTransform -rm {<memory> SHALLOW}
```

To prohibit transformation of **all** shallow memories into gate arrays, use the following command:

```
memoryTransform -add {* SHALLOW off}
```

To restore default behavior for **all** shallow memories, use the following command:

```
memoryTransform -add {* SHALLOW on}
```

The number of shallow memory instances transformed into gates depends on the *shallow memory utilization factor* discussed below.

Setting the Shallow Memory Utilization Factor

By default, transformation of shallow memory into gates starts from the smallest memory instances and continues while gate utilization (in percents) is below the threshold value (80% by default). This value can be changed using the following command:

```
memoryTransform -add {* shallowMemoryMaxUtilization <number>}
```

where `<number>` is a non-negative integer between 0 and 100.

Merging Memory Instances

If two memory ports have the same type (read or write), the same depth, the same address bus and they are independent of each other, they are called mergeable ports.

© 2021 All Rights Reserved.

If several memory instances M1, M2, ..., Mk have the same number of ports and their corresponding ports are mergeable, they can be merged into memory instance M by placing their data signals side-by-side.

To prohibit merging of memory instances, use the following memoryTransform command:

```
memoryTransform -add {<memory> MERGE off}
```

To restore the default behavior, use the following command:

```
memoryTransform -add {<memory> MERGE on}
```

To remove the merge memory transformation of the specified memory instance, use the following command:

```
memoryTransform -rm {<memory> MERGE}
```

Splitting Memory Read Ports

Each memory instance is represented in the data base as a chain of Read (MPR) and Write (MPW) ports. Multi-port memory instances can be responsible for high step count. If a memory instance has more than one MPR, it is possible to decrease its chain length by using N identical memory storages (instead of the original one) and splitting the given chain into N chains as follows. Each chain contains the same MPW as in the original chain. In other words, each original Write port is “represented” in each chain. Each Read port (MPR) is represented in only one chain. The port order in each chain will be the same as in the original chain. Pins A, DI, BS, WE, DO of each “new” port will be connected to the same nets as the corresponding pins of the corresponding original port.

Such a splitting can decrease the step count for a design. The trade-off is an increase in the required hardware memory size.

Currently splitting can apply only to memory instances “destined” to be mapped to EXTRAM and only if such a splitting would decrease the estimated maximum (among the physical EXTRAM) number of accesses to a physical EXTRAM.

To prohibit split memory transformation, use the following memoryTransform command:

```
memoryTransform -add {<memory> SPLIT off}
```

To restore the default behavior, use the following command:

```
memoryTransform -add {<memory> SPLIT on}
```

To remove the split memory transformation of the specified memory instance, use the following command:

```
memoryTransform -rm {<memory> SPLIT}
```


© 2021 All Rights Reserved.

Squeezing Memory Instances with Constant Address Pins

If a memory instance has an address input pin tied (no matter whether the pin is tied up or down, but it is tied the same way for each memory port), such a pin can be removed. Removal of one address pin reduces the memory size two times, removal of two address pins reduces it four times, and so on (in general, removal of N address pins reduces the memory size 2^N times.) Such a “squeezing” can allow for designs to save considerable resources of physical memory.

To prohibit memory squeezing, use the following memoryTransform command:

```
memoryTransform -add {<memory> SQUEEZE off}
```

To restore the default behavior, use the following command:

```
memoryTransform -add {<memory> SQUEEZE on}
```

To remove the memory squeeze transformation of the specified memory instance, use the following command:

```
memoryTransform -rm {<memory> SQUEEZE}
```

Compacting Memories

If a design memory instance with data width not equal to a multiple of 32 is mapped on emulator memory, a part of emulator memory space would be lost. Memory compaction transforms the design memory so that the emulator memory space can be used effectively.

Following types of memories cannot be compacted in this way:

- Memory instances with address bus width equal to 6 or less (that is, with depth equal to 64 or less)
- Memory instances with data bus width equal to a multiple of 64 (64, 128, 192, and so on)

The default behavior for memory compaction is as follows. The process browses the design database and collects memory instances. It then calculates the required DRAM size (D) and compares it with the available DRAM size (A). If D<=A, no compaction is performed. Otherwise, it finds a minimum subset of memory instances, the compaction of which would reduce D to Dc where Dc<=A. (If this is unachievable, ALL “compactible” memory instances will be transformed).

To prohibit memory compaction, use the following memoryTransform command:

```
memoryTransform -add {<memory> COMPACT off}
```

To force memory compaction, use the following command:

```
memoryTransform -add {<memory> COMPACT on}
```


© 2021 All Rights Reserved.

To remove the memory compaction of the specified memory instance, use the following command:

```
memoryTransform -rm {<memory> COMPACT}
```

The number of memory instances that can be compacted depends on the *memory utilization factor* discussed below.

Setting the Utilization Factor

To change the number of compacted memory instances, use the following command:

```
memoryTransform -add {* memoryUtilization <number>}
```

where `<number>` is a non-negative integer representing the utilization factor (U), given in percent for available emulator memory size.

Setting U < 100 will force more memory instances to be compacted; if U = 0, all compactible memory instances will be compacted because $A_m = 0$.

To remove the utilization factor from User Data, use the following command:

```
memoryTransform -rm {* memoryUtilization}
```

Handling Sparse Memories

Memory instances that have the following characteristics are referred to as sparsely used memories or sparse memories:

- A large/deep address range in their declaration
- Only a small percentage of the address range accessed during simulation or emulation

In conventional memory mapping, if very deep sparse memories are present, a design could require a very large amount of hardware resources. When the design with sparse memories, is accelerated in the emulator, for efficient use of emulator resources, instead of implementing the entire/declared address range for the sparse memory, only the percentage accessed during simulation should be implemented.

To handle sparse memories, the VXE software includes a strategy known as a *page model*. The page model requires much less memory space, compared to a conventional memory model. Refer to Handling Sparse Memories through the Page Model for details.

© 2021 All Rights Reserved.

Handling Sparse Memories through the Page Model

The VXE software uses the Page Model to implement sparsely-used memories. This model is used to dynamically map only the accessed pages of a sparsely-used memory onto the emulation (hardware) memory. The fraction of the accessed pages should be small.

To understand the page model, consider that a sparsely-used memory `M` has `aw` address bits, that is, its depth is equal to 2^{aw} words. Consider the address space of `M` as a sequence of $K=2^k$ pages (blocks) of size 2^p each. Then, `aw=k+p`. These 2^k pages are called *virtual pages*.

Note: '*Words*' mean addressable words of the corresponding memory instance.

The page model consists of a pool memory which contains $N=2^n$ physical pages and a catalogue that maintains the mapping from virtual pages to physical pages. If a virtual page is not mapped to any physical page, then that virtual page is considered to have the default value.

Conditions When Sparse Memory Implementation Does Not Improve Capacity

It is possible that declaring a design memory as sparse can increase the amount of emulator memory required. In such a situation, the compiler will issue an error or warning:

- If the sparse memory implementation with the parameters provided consumes the same or more memory resources than the 'normal' implementation, an ERROR message is issued.

If the sparse memory implementation consumes less memory than 'normal' implementation but more than in the case of memory compaction, a WARNING message is issued.

Modeling Sparsely-Used Memories at Compile Time in ICE Mode

During design emulation, some addresses of each memory instance are accessed while others are not. If the percentage of accessed addresses for a memory instance is very low, the memory is said to be sparsely used.

Some designs contain very deep sparsely-used memories. The conventional way of mapping such memories onto emulator's memory can require huge hardware resources and become the bottleneck for emulation.

In ICE mode, if you know that certain memories in your design are sparsely used, you can use the `memoryTransform` command at compile time to declare these memories as

© 2021 All Rights Reserved.

"*sparse*". For each sparsely-used memory, you specify a number of "pages" to be implemented physically in the emulator. If at run time, a test writes to more than this number of pages, the emulation stops with an error. (In LA mode, the emulation issues an error message but does not stop.) You can use the `memory -reset` command at run time to reset a sparsely-used memory, setting the memory contents to zero, and freeing all of its pages.

For detailed information on access to sparsely-used memories at run time, refer to Accessing Sparsely-Used Memories at Run Time in ICE Mode.

Declaring Sparsely-Used Memories in ICE Mode

Sparsely-used memories must be declared as sparse using the following XEL command:

```
memoryTransform -add {<name> SPARSE <n> <p>}
```

where,

<*name*> is a hierarchical name of the memory

<*n*> is used to calculate the number of physical pages, that is, Number of physical pages = 2^n

<*p*> is used to calculate the size of each page, that is, Number of words per page = 2^p

This command specifies that a memory instance with name <*name*> should be treated as a sparsely-used memory which can use at most $N=2^n$ physical pages, and each page should contain 2^p words.

For example, the following command specifies that memory instance `top.M123` should be modeled as a sparsely-used memory with `2M` (2^{21}) physical pages, and each page should be `64` (2^6) addresses deep (that is, contain 64 words):

```
memoryTransform -add {top.M123 SPARSE 21 6}
```

The `memoryTransform` command should precede the `compile` command in the compiler script.

<*n*> and <*p*> should be positive numbers such that `n+p<`*aw*, where *aw* is the number of address bits of the sparsely-used memory. If these rules are violated, a corresponding error message is issued. There are no default values for <*n*> and <*p*>. You must choose these values based on the knowledge of the design and understanding that the sparse model consists of two memory instances of the following sizes instead of the original memory size, 2^{aw}x*dw*:

- 2^{aw-p}x*cw* (catalogue)

  Note: The catalogue width `cw` is either `32` or `64`, depending on the implementation.

© 2021 All Rights Reserved.

- 2^{n+p}x*dw* (pool)

For example, if a sparsely-used memory instance has size 2^{32}x64 and you have defined p=9 (page size is 512 words) and n=16 (has 64K physical pages), then without 'sparse implementation', this instance will consume 4G of 64-bit wide words of the emulator memory. The sparse model will require 32M of 64-bit wide words for pool memory, and 8M of 64 or 32-bit wide words for catalogue memory.

The emulator's memories are at least 32-bits wide. When necessary, the compiler compacts the narrow memories, replacing them with shallower memories of width 32 or 64. However, the compiler cannot compact the catalogue or pool of a sparsely-used memory. Keep this fact in mind when choosingthe values for <*n*> and <*p*>.

After compiling, the dbFiles/<*top*>_mpart file shows the depth and width of the memories as actually implemented.

Removing Sparsely-Used Memories in ICE Mode

You can use the following XEL command to remove a specific sparsely-used memory:

```
memoryTransform -rm {<name> SPARSE}
```

Where, <*name*> is a hierarchical name of the memory.

Accessing Sparsely-Used Memories at Run Time in ICE Mode

nce designs containing sparsely-used memories have been compiled, those memories can be accessed at run time by using either of the following methods:

- Using the *Memory* tab in the xeDebug GUI
- Executing the memory -load and memory -dump XEL commands on the command line

For detailed information about handling sparsely-used memories in a design at compile time, refer to Modeling Sparsely-Used Memories at Compile Time in ICE Mode.

While accessing sparsely-used memories at run time, you must keep the following points in mind:

- For memory -load command, remember that the memory was defined as sparse, and therefore, only part(s) of its address space (not exceeding the POOL size) can be loaded.
- If a sparse memory was dumped with memory -dump command, and then loaded with memory -load command, the memory content will be the same as before dumping.

© 2021 All Rights Reserved.

- `memory -reset` command sets all virtual pages to have the default value 'all zeroes'. In the other words, it 'frees up' the memory.
- `memory -set` command performs a similar operation as `memory -reset` command. However, it sets all virtual pages to have the default value of 'all ones' instead of 'all zeroes'.
- If during the emulation, one of the sparse memories overflows, the emulation stops and an error message (containing the name of the overflowed memory instance) is issued. In this case, if required, create a script to catch this error, reset the overflowed sparse memory, and continue emulation.

You can use the following XEL commands to obtain information about the sparsely-used memories:

- `memory -sparse_mem_info [-v] <mem_name>`

  Provides information such as, page size, number of virtual pages, number of physical pages, and number of used physical pages for the specified sparse memory. Using the `-v` option displays the information in a more user-friendly format.

- `memory -sparse_mem_info -usedpages [-v] <mem_name>`

  Provides a list of page numbers for all used virtual pages, that is, virtual pages that do not have the default value for the specified sparse memory. Note that the list can be very long. Using the `-v` option displays the information in a more user-friendly format.

- `memory -sparse_mem_info -statistic [-v] [-byusage]`

  Provides statistic information for all sparse memories. The output of this command lists the percentage of used physical pages for each memory name. The list is sorted by memory name. If you specify the `-byusage` option, the list is sorted by usage.

- `memory -sparse_mem_info -overflow`

  Provides a list of overflown sparse memory instances.

Important

In SA mode, the above-listed `memory` command options for accessing sparsely-used memories at run time are not supported.

For more information on the `memory` XEL command, refer to the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Also, note that the SA-specific `readmem` memory-access command is not supported for sparsely-used memories.

© 2021 All Rights Reserved.

Examples

Listing Sparse Memories

```
XE> memory -sparse_mem_info -statistic
18% top.ABin0.A1write.A1nfbuf.A1nfbuf_ram.mem
```

Getting Information about a Sparse Memory

```
XE> memory -sparse_mem_info
top.ABin0.A1write.A1nfbuf.A1nfbuf_ram.mem
32 3216 7
XE> memory -sparse_mem_info -v top.ABin0.A1write.A1nfbuf.A1nfbuf_ram.mem
page size 32, 32 virtual pages, 16 physical pages, 7 used physical pages
XE> memory -sparse_mem_info -usedpages -v
top.ABin0.A1write.A1nfbuf.A1nfbuf_ram.mem
7 virtual pages are used:
0 1 2 3 4 5 6
```

Clearing a Sparse Memory (Reset)

```
XE> memory -sparse_mem_info -statistic
43% top.ABin0.A1write.A1nfbuf.A1nfbuf_ram.mem
XE> memory -reset top.ABin0.A1write.A1nfbuf.A1nfbuf_ram.mem
XE> memory -sparse_mem_info -statistic
0% top.ABin0.A1write.A1nfbuf.A1nfbuf_ram.mem
```

Support for Sparse Memories in SA Mode

In ICE mode, a memory can be identified and implemented as a sparse memory by using the `memoryTransform` command. However, the functionality provided by this command is not sufficient for SA mode. The `memoryTransform` command implements the specified memory as sparse only in the hardware emulator; it does not have the ability to do the same in the software simulator. In SA mode, both the software simulator and the hardware emulator need to have the same sparse memory implementation model to implement key features like hot-swap.

A separate implementation for sparse memories is required in SA mode.

> **Note:** *xrun* supports the `/* sparse */` pragma, which can be used to declare a sparse memory for simulation purposes. The IXCOM compiler recognizes this pragma in your HDL, and passes it on to *xrun* by dumping it in the HDL that it generates for *xrun* consumption. The IXCOM support for the *xrun* "`/* sparse */`" pragma is orthogonal (that is, it has no correlation) to the sparse memory support in IXCOM mode.

This section describes how to specify a memory to be implemented as a sparse memory in SA mode, compile these sparse memories, and access these memories at run time. This

© 2021 All Rights Reserved.

section also describes the limitations of this sparse memory implementation. This section also provides a comparison between the sparse memory implementations in ICE and SA modes.

The following sections describe the processes for:

- Specifying Verilog Memory as Sparse Memory in SA Mode
- Specifying VHDL Memories as Sparse Memories in SA Mode

Specifying Verilog Memory as Sparse Memory in SA Mode

You can identify a Verilog memory to be implemented as a sparse memory in SA mode by using the following `$ixc_ctrl` system task:

```
$ixc_ctrl("sparse_mem", <memory_name>, <num_pages>, <page_size>);
```

Here:

- <*memory_name*> is the reference to memory to be implemented as sparse.
- <*num_pages*> is the number of pages of memory accessed by the testcase.
- <*page_size*> is the size of each memory page (unit: memory word / address).

Note: <*num_pages*> and <*page_size*> are rounded up the nearest power of 2.

Only a subset of the actual memory declaration, which is controlled by the <*page_size*> and <*num_pages*> parameters is implemented in both the software simulator and the hardware emulator. To be specific, only the <*num_pages*> pages of the memory are implemented with each page containing <*page_size*> addresses. This sparse memory implementation model involves implementing only a subset of the actual memory declaration This implementation does NOT have a cache-memory concept, where the memory is implemented as declared in the software simulator, and only the needed subset is "*cached*" into the hardware emulator.

The page boundaries are calculated solely on the basis of the <*page_size*> parameter, and are independent of the memory declaration. So if the <*page_size*> parameter is `1024`, the page boundaries are 0-1023, 1024-2047, 2048-3071, and so on.

The <*num_pages*> pages of memory accessed by the testcase/simulation do not have to be contiguous; these can be any located anywhere within the memory declaration.

The first *write* access to any memory address within a page will mark that page as *used* or *allocated*, and count it against the <*num_pages*> total. *Read* access to a memory address does not mark the page containing it as *used* or *allocated*.

© 2021 All Rights Reserved.

Read access to a memory address, which has not yet been written to, results in a default value being returned ('`x`' for four-state, `0` for two-state).

An attempt to access more memory pages than were allocated during compilation (via the <*num_pages*> parameter results in a run-time error.

The sparse memory implementation checks for the following in the `$ixc_ctrl` statement used to specify sparse memory.

```
$ixc_ctrl("sparse_mem", <memory_name>, <num_pages>, <page_size>);
```

The second argument (<*memory_name*>) must be a reference to a valid memory instance (one-dimensional unpacked array of a one-dimensional packed array).

The third and fourth arguments (<*num_pages*> and <*page_size*>) must be positive integers.

The ratio of the size of memory in the sparse memory implementation to the size of the memory in the original declaration must be less than 0.5 (that is, 50%). This ratio is calculated using the formula:

<*num_pages*> * <*page_size*> /
<*num_addresses_in_the_original_declaration*>

The values used for <*num_pages*> and <*page_size*> in this calculation are the ones after these have been rounded up to the nearest power of 2.

Violation of any of these conditions result in a compile-time error.

- For every reference to a sparse memory that occurs in an unsupported context as outlined above, the `UNSUP_SPMEM_REF` (Unsupported Sparse Memory Reference) error is issued.
- For every reference (in an LHS context) to a sparse memory that occurs in an unsupported LHS context as outlined above, the `UNSUP_SPMEM_WRITE_REF` (Unsupported Sparse Memory Write Reference) error is issued.
- For every XMR reference to a sparse memory that occurs in an unsupported context as outlined above, the `UNSUP_SPMEM_XMR_REF` (Unsupported Sparse Memory Out Of Module Reference) error is issued.

Example of Verilog Sparse Memory Implementation in SA Mode

Assume that a design includes a 1G address Verilog memory, and the testcase or simulation only accesses 16M addresses of that memory during simulation:

```
reg [63:0] mem[(2**30 -1):0];   // 1G memory decl
```


© 2021 All Rights Reserved.

One way to specify sparse memory implementation for this memory is as follows:

```
initial $ixc_ctrl("sparse_mem", mem, 16, 2**20);
```

The above syntax implements `mem` as a sparse memory of `16` pages, with each page containing `2**20` addresses.

The page boundaries will be:

`0M to (1M -1)`, `1M to (2M -1)`, ...and so on.

The testcase or simulation can do a *write* access to addresses within any 16 such pages within the memory declaration, and a *read* access to any address within the entire memory declaration.

The 16 pages with *write* accesses do not have to be contiguous. These can be any 16 of the 1024 pages (1G divided by 1M) located within the memory declaration.

Attempts to do a *write* access to a 17th page will result in a run-time error.

Note that page boundaries are dictated solely by the <*page_size*> parameter, and the address range specified in the memory declaration has no effect on the page boundaries. This will not have any impact in most cases because memory declarations usually have `0` as the lower address bound.

Assume a sparse memory declaration is as follows:

```
reg [63:0] mem [258:321];
initial $ixc_ctrl("sparse_mem", mem, 2, 16);
```

For this memory, the page boundaries that straddle the memory declaration are:

256 - 271

272 - 287

288 - 303

304 - 319

320 - 335

So *write* access to address 271 and 272 will count as accessing two separate pages, even though they are within the first 16 memory addresses as per the memory declaration (258 - 273). On the flip side address 320 and 321 will be considered to be part of the same page as (258 - 271).

© 2021 All Rights Reserved.

Accessing Verilog Sparse Memories at Run Time in SA Mode

The Verilog sparse memory implementation instruments all (legal) access to a sparse memory in the HDL. However at run time, care must be taken to ensure that attempts to access a sparse memory go through the sparse memory implementation. All references to the original memory in the HDL are instrumented at compile time, and the original memory declaration cannot be accessed at run time via the GUI.

The various ways to access a Verilog memory at run time are discussed below:

- **xeDebug commands:**

  Only the `memory` command can be used to access DUT sparse memories via xeDebug. All other xeDebug commands that access memory (`value`, `deposit`, `$readmem`, `$writemem`, and so on) access the original memory declaration and not the corresponding sparse memory declaration. The `memory` command (for a DUT memory) calls the equivalent `xc memory` command. The `xc memory` command is executed by IXCOM run time and uses the sparse memory declaration instead of the original memory declaration. All other xeDebug commands that enable memory access are not routed through IXCOM run time, and hence cannot be updated to use the sparse memory implementation instead of the original declaration.

- **xrun commands:**

  Just like xeDebug commands, access to sparse memory is supported only via the `xc memory` tcl command. Using any other `xrun` command to access the sparse memory accesses the original declaration and not the sparse memory implementation.

- **DPI, VPI, and other interfaces:**

  Access to sparse memories is only allowed via the `marg` programming interface. Access through DPI, VPI, or other means accesses the original memory declaration.

Limitations with Handling of Verilog Sparse Memories in SA Mode

In Verilog sparse memory implementation, all *`read`* accesses of the sparse memory are replaced with a function call and all *`write`* accesses of the sparse memory are replaced with a task call.

- Sparse memory references are supported only in the constructs listed below:
  - `initial` and `always` processes
  - Primitives and continuous assignments
  - Tasks and functions

© 2021 All Rights Reserved.

- Property and Sequence declarations
- Assertions

Note: Only covergroup-related constructs are included.

- *Read* access to a sparse memory is allowed in all instances in which a function call expression is legal (within the supported constructs listed above).
- Access to a sparse memory via `$readmem*` and `$writemem*` system tasks is supported.
- Access to a sparse memory via the "`marg`" interface (both the C and SystemVerilog incarnations) is supported.
- OOMRs to sparse memory as an argument to the `$readmem*`/`$writemem*`/`marg` system tasks are supported.
- OOMRs to DUT sparse memories are supported in the testbench partition. OOMR access to DUT memories in the testbench partition gets converted to corresponding "`marg`" interface calls. Access to sparse memory through the "`marg`" interface is supported. Therefore, OOMRs to DUT sparse memories in the testbench partition are also supported.

Errors in Verilog Sparse Memory Implementation in SA Mode

Compile-time errors in are generated in the following cases:

- Reference to a sparse memory in a construct other than the ones listed above.
- Any forms of *write* access to a sparse memory except those in the LHS of a procedural assignment.
- An expression that accesses more than one memory word (like slice select) or an expression that accesses parts of a memory word (like bit/part select on a memory word). Both *read* and *write* access to a sparse memory should be on a memory word basis.
- Access to a sparse memory via any other `$systf` except `$readmem*` and `$writemem*` system tasks.
- OOMRs (Out Of Module References) to DUT sparse memories in the hardware partition.

At run time, the Verilog sparse memory run-time implementation issues the following error message when the testcase or simulation tries to access more pages within the sparse memory than were allocated for its implementation at compile time:

© 2021 All Rights Reserved.

```
** IXCOM Runtime ERROR **
For sparse memory (mem) in module (dut)
Attempt to write to address 37, results in
number of blocks accessed during simulation exceeding the
number of blocks allocated at compile time (2)
```

This error will also result in the simulation being stopped, that is, a `$stop` will be executed immediately after this statement is issued.

Specifying VHDL Memories as Sparse Memories in SA Mode

This section outlines the support for VHDL sparse memories in SA mode. It describes the syntax for identifying a VHDL memory to be implemented as a sparse memory, and the parameters that you can use for controlling the sparse memory implementation. It also lists the limitations in this support, and the errors that you might encounter during compilation or at run time.

Here, a memory is defined as a one-dimensional array of std_logic_vector, bit_vector, signed, unsigned types declared in VHDL. Sparse memories, or sparsely-used memories are defined as memory instances that have:

- A large or deep address range in their declaration

  And,

- Only a small percentage of the address range accessed during run time

Syntax for Specifying VHDL Memories as Sparse Memories

You can identify a memory to be implemented as a sparse memory in VHDL by specifying the following pragma either just before or after the memory declaration:

```
--pragma SPARSE_MEM <memory_name> <num_pages> <page_size>
```

Here:

<*memory_name*> is a string and reference to the memory to be implemented as a sparse memory. It must be a reference to a valid memory instance, that is it must be a one-dimensional unpacked array of a one-dimensional packed array.

<*num_pages*> is a positive integer that specifies the number of pages of memory accessed by the testbench.

<*page_size*> is a positive integer that specifies the size of each memory page (unit: memory word / address).

© 2021 All Rights Reserved.

The <*num_pages*> and <*page_size*> parameters are rounded up to the nearest power of 2.

The ratio of the size of memory in the sparse memory implementation to the size of the memory in the original declaration must be less than 0.5, or 50%. This ratio is calculated using the following formula:

```
<num_pages> * <page_size> / <num_addresses in the original
declaration>
```

If there are any errors in the pragma statement or memory calculation, a compile-time error is generated.

The sparse memory implementation is only a subset of the actual memory declaration, and is controlled by the <*page_size*> and <*num_pages*> parameters. The <*num_pages*> pages of the memory are implemented with each page containing <*page_size*> addresses. Thus, the total addresses implemented are <*num_pages*> `*` <*page_size*>.

The page boundaries are calculated only on the basis of the <*page_size*> parameter, and are independent of the memory declaration. So if the <*page_size*> parameter is 1024, the page boundaries are `0 - 1023, 1024 - 2047, 2048 - 3071`, and so on.

The <*num_pages*> pages of memory accessed by the testbench do not have to be contiguous; these can be located anywhere within the memory declaration.

The first *write* access to a memory address within a page marks that page as *used* or *allocated*. Read access to a memory address does not mark the page containing it as as *used* or *allocated*.

If you attempt to read a memory address that has not yet been written to, a default value, `U` for `std_logic` and `0` for `bit`, is returned. If you attempt to access more memory pages than the number you specified for <*num_pages*> parameter, during compilation a run-time error is generated. Every write access to a sparse memory is transformed to the corresponding sparse-memory-write procedure call, and every read access is transformed into corresponding sparse-memory-read function call.

VHDL sparse memories can be specified only if these exist in the DUT portion of the design. If you attempt to specify a memory in the testbench portion as a sparse memory by using the `SPARSE_MEM` pragma, a warning message is generated. Testbench memories are processed by Xcelium. However, if the size of the memory is greater than that permitted by Xcelium, Xcelium generates an error. In this case, you need to explicitly specify the required memory size by passing the `-vhdlsparsearray` <*size*> argument with the `-xmelabarags` option of the `ixcom` command. IXCOM passes this option to Xcelium by dumping it in the `xmelab.args` file.

© 2021 All Rights Reserved.

Following is an example of specifying VHDL sparse memories. Assume, that a 1G address memory exists in the design as follows:

```
type arrType is array((2**30 - 1) downto 0) of std_logic_vector(63 downto 0);
--pragma SPARSE_MEM mem 16 1048576
signal mem : arrType;
```

In this example, only 16M addresses of this memory are accessed during simulation.

The above pragma statement implements `mem` as a sparse memory of `16` pages, with each page containing `2**20 (1048576)` addresses. So, out of the total 1G addresses, only 16M addresses are implemented. The page boundaries will be `0M to (1M -1)`, `1M to (2M - 1)`, and so on.

At run time, you can write to any of the addresses located within any of 16 such pages within the memory declaration, and you can read any of the addresses within the entire memory declaration.

The 16 pages with write access need not be contiguous. These can be any 16 pages of the 1024 pages (1G divided by 1M) located within the memory declaration. If you attempt to write to a 17th page, an error will be generated.

Note: Page boundaries are set according to the <*`page_size`*> parameter, and the address range specified in the memory declaration has no effect on the page boundaries.

Limitations in the Support for the VHDL Sparse Memories

For sparse memories, all read accesses of the sparse memory are replaced with a function call and all write accesses of the sparse memory are replaced with a procedure call. These replacements impose limit the access to sparse memories.

- Sparse Memory references are supported only in the following constructs:
  - Process blocks - variable and sequential signal assignment statements (in LHS and RHS of the assignment) only. It is not supported in function calls or procedure calls.
  - Concurrent signal assignment statements (in LHS and RHS of the assignment).

  Note: Reference to a sparse memory in a construct other than the ones listed above results in a compile-time error.
- Write access to a sparse memory is allowed only on the LHS of an assignment. Else, a compile-time error is generated.
- Read access to a sparse memory is allowed only on the RHS of an assignment. For signal assignments, the sparse memory can appear only in the waveform expression. Any other forms of read access to a sparse memory result in a compile-time error.

© 2021 All Rights Reserved.

- Both read and write access to a sparse memory should be on a memory-word basis. This means that an expression that accesses more than one memory word, such as slice-select, or an expression that accesses parts of a memory word, such as bit- or part-select on a memory word results in a compile-time error.
- Out-Of-Module References (OOMRs) to DUT sparse memories are not supported in the DUT-Testbench partition, and result in a compile-time error.
- Only one-dimensional arrays of vectors of `std_logic`, `std_ulogic`, and `bit` types are supported for memories to be implemented as sparse memories.
- User types are not supported.
- Ports cannot be declared as sparse memories.
- Only signals within architectural scope can be declared as sparse memories. Variables or signals in another scope, declared as sparse memories are not supported.
- Sparse memories cannot be declared or accessed within function or procedure calls.
- Sparse memories cannot be specified in the sensitivity list of the process, or as an argument to a function or procedure call.
- Use only integers to specify the number of pages and page size. Do not use expressions, like (`2**20 - 1`); Instead, specify the expanded value.
- Sparse memories are not supported within records.

Comparison of Sparse Memory Support between SA and ICE Modes

Following are the differences in the support for sparse memories in SA and ICE modes.

Syntactical Differences

- SA: (HDL in the module containing the sparse memory)

```
$ixc_ctrl("sparse_mem", <mem_name>, <num_pages>, <page_size>)
```

- ICE: (XEL command to xeCompile)

```
memoryTransform -add {<mem_name>, SPARSE, <log2_num_pages>, \
                <log2_page_size>}
```


© 2021 All Rights Reserved.

The difference is in <*num_pages*> and <*page_size*> parameter specification. Assume the implementation is for 16 pages of 1024 addresses each. The parameter values would be as follows:

	Required	IXCOM	ICE
num_pages	16	16	4
page_size	1024	1024	10

If an XEL command is used to specify a sparse memory in SA mode, an error message is issued.

In SA mode, access to more pages than are allocated at compilation results in the simulation being stopped with the invocation of `$stop`, and an error message is issued.

Run-Time Differences for Sparse Memories support in SA and ICE Modes

In the IXCOM mode, the following run-time commands are not supported for sparse memories:

- `memory -set | reset`

  Use the above command to reset the sparse memory implementation, that is, to free or unallocate all the used or allocated pages.

- `memory -sparse_mem_info`

  Use the above command to view a report containing information related to sparse memory usage.

If you attempt to access the original memory declaration, it can lead to errors. All (legal) accesses to sparse memories (in the HDL) are instrumented by the IXCOM compiler to use the sparse memory implementation instead of the original memory declaration. All illegal accesses to sparse memories (in the HDL) result in a compile-time error.

Similarly at run time, (legal) accesses to sparse memory through either the `marg_interface` or the `xc memory` command are executed by to access the spare memory implementation instead of the original memory declaration. However at run time, the original memory declaration can be accessed through DPI, VPI, and GUI and so on.

© 2021 All Rights Reserved.

Memory Slicing

Memory slicing is useful for logical distribution of large memories into physical resources of Palladium Z1. Memory slicing is done automatically as part of the compilation process, without requiring user control.

Palladium Z1 systems support the following types of memory slicing:

- Vertical Slicing: This refers to slicing of every memory instance of a design memory into multiple chunks along the data width. In this type of slicing, the physical memory of Palladium Z1 is sliced into 32, 64 bits chunks for internal memory and 64, 128, 256, 512 bits chunk for external memory.
- Horizontal Slicing: This refers to slicing of every memory instance of a design memory into multiple chunks along the address depth. For example, the physical boundary of internal and external memory is 17 and 30 address bits.

Enabling Memory Assignment and Memory Slicing

You can assign a target for the memory. This memory target assignment is honored if possible. Based on the target assignment, appropriate horizontal and vertical slicing is performed. Memories are first sliced horizontally, and then vertically.

During horizontal slicing, all memories with address buses deeper than 24 bits are assigned to EXTRAM.

The `memTarget` command helps to assign all memory slices to a particular type of resource as per memory target assignment specified by you. The type of resource can be either INTRAM or EXTRAM. The slicing operation first looks into user request for target assignment, then assigns all memory slices to INTRAM, if possible. If that is not possible, then the deepest memories are assigned to EXTRAM.

Default Setting of Memory Assignment and Memory Slicing

By default,every memory (including the shallowest of memory) in the design is implemented in INTRAM, which is faster than EXTRAM. This ensures that every memory is sliced in such a way that its vertical and horizontal dimensions are not more than 64 and 17 bits, respectively. The presence of bit-enable memories is an exception to this logic because the memories are vertically sliced into 32-bit chunks and the rest of the 32 bits are used for bit-enable signals.

When the INTRAM resources are completely utilized, the design memories are implemented in EXTRAM. At this stage, by default all 24-bit deep memories are implemented in EXTRAM

© 2021 All Rights Reserved.

unless you force these sliced memories to be implemented in INTRAM and have enough space in INTRAM to accommodate the memory slices.

Note: Bit-enabled features are not supported for EXTRAM.

You can override the above slicing process, and define the memory location by using the `memTarget` command.

Considerations for Memory Slicing

Following are the considerations to be kept in mind for memory slicing:

- The internal RAM (INTRAM) of the domain can only accommodate 17 bits of address lines; while the external RAM (EXTRAM) can handle upto 30 bits of address lines. By default, all memories with equal or more than 24 bits of address lines use EXTRAM. You can control memory target placement by running `memTarget` command and then recompiling the design.
- The internal domain supports bit-enable feature (BE), enabling user memories to be vertically sliced up to 32-bit chunks. However, the rest of the 32 bits are used for bit-enable signals, such that memory slicing is possible at bit level, instead of byte level.
- For Palladium Z1 systems, irrespective of whether you use the `memTarget` command, memory slicing takes place.

  Use the `memTarget` command as follows:

  ```
  memTarget -add {<memory_instance_name> INTRAM | EXTRAM}
  ```

  `memTarget` command specifies memory instances to be placed on either EXTRAM or INTRAM.

  The first argument <*memory_instance_name*> can take "*" specifying all the memories in a given design to be placed in a particular memory target.

  `memTarget` command will ensure that there is a specific location for the placement of a design memory. Depending on the data, address bus width, and the geometry of selected target, appropriate memory slicing occurs.

  Refer to memTarget command in *VXE Command Reference Manual* to view the syntax of this command.
- Final slicing might occur during the universal partitioning (UPART) phase because UPART module determines memory assignment.

© 2021 All Rights Reserved.

Using Memory Primitives

A memory is created with memory primitives by creating an MPR for each read port and an MPW for each write port. The address and data ports are connected as appropriate. The active-high write enable should be connected so that the data and address for the write port is valid during the entire period that the write enable is active. If the address or data changes at the time that the write enable becomes active, the write operation uses the new value for the address and data lines. Since there is no hold requirement, the address and data lines can change at the same time the write enable becomes inactive.

The SYNC_IN and SYNC_OUT pins are used to tie the memory ports together into a single memory. The sync pins also control the order of evaluation for the memory ports. The first memory port needs to have its SYNC_IN pin tied to ground. The remaining ports need to be connected SYNC_OUT to SYNC_IN with the last port's SYNC_OUT pin left unconnected. The write ports should generally be placed before the read ports. This will cause a read from the same location as is being written to return the value currently being written. This ordering mechanism can also control which write port will have the precedence when there are multiple write ports. The instance name of the first port is the name used as the instance name of the memory for reading and writing.

Probing Restrictions

SYNC_IN and SYNC_OUT nets are not real nets. They are sequencing signals introduced to model port ordering in a multi-port memory. The nets will not exist in the design being emulated. As such, it is invalid to specify them as probe or keepnet.

Building Memory Primitives

The memory primitive module names are Memory Port Read (MPR) and Memory Port Write (MPW). Multiple MPRs and MPWs tie together to form a single memory with multiple ports that access that memory.

The designImport recognizes instances of modules with names of the form `[MPR | MPW MPW_BS]<depth>X<width>` and creates the module definitions as needed. There is no need to predefine MPRs, MPWs, and MPW_BSs.

For depths of 1K to 512K, the depth is referred to with K notation. For depths of 1M to 512M, the depth is referred to with M notation. For depths of 1G to 512G, the depth is referred to with G notation. For depths of 1T to 512T, the depth is referred to with T notation. For depths of 1P to 512P, the depth is referred to with P notation. For depths of 1E to 8E, the depth is referred with E notation. 8E (63 address bits) is the maximum supported depth.

© 2021 All Rights Reserved.

The pins used in the modules must be in a particular order, but you can choose the pin names. The pin names are then referenced by the modules that instantiate these MPR and MPWs. The following is an example of an 8-deep 2-bit wide MPR and MPW:

```
module MPW8X2 (WR_ADDR2, WR_ADDR1, WR_ADDR0, DATAIN1, DATAIN0, WR_ENBL, SYNC_IN,
SYNC_OUT);
input WR_ADDR2, WR_ADDR1, WR_ADDR0;
input WR_ENBL;
input DATAIN1, DATAIN0;
input SYNC_IN;
output SYNC_OUT;
endmodule
```

```
module MPR8X2 (RD_ADDR2, RD_ADDR1, RD_ADDR0 SYNC_IN, DATAOUT1, DATAOUT0, SYNC_OUT);
input RD_ADDR2, RD_ADDR1, RD_ADDR0;
input SYNC_IN;
output DATAOUT1, DATAOUT0;
output SYNC_OUT;
endmodule
```

In the example, the address and data pins are ordered from most significant bit to least significant bit. The SYNC_IN and SYNC_OUT signals are the special signals described in Using Memory Primitives. The write enable (WR_ENBL) signal is an active high signal. There are no read enable signals because the read port is always active.

Enforcing Design Rules for Memory Primitives

Memory primitive cell names are:

```
- MPR<size>                 // read port
- MPW<size>                 // write port
- MPW_BS<size>              // write port with Byte-select signals
<size>   ::= <depth>X<width> | <depth>x<width>
<depth>  ::= 1<suffix> | <number>[<suffix>]
<number> ::= 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512
<suffix> ::= K | M | G | T | P | E
EXAMPLES. MPW8X17; MPR128GX36; MPW_BS1EX1
NOTE: K stands for kilo (2 power 10); M stands for mega (2 power 20);
      G stands for giga (2 power 30); T stands for tera (2 power 40);
      P stands for peta (2 power 50); E stands for exa  (2 power 60).
      (See www.en.wikipedia.org/wiki/Binary_prefix)
NOTE. The minimum depth is 2 (aw = 1). The maximum depth is 8E (aw = 63).
```

A memory primitive cell name implicitly defines the number of terminals.

- For MPR:

```
#inputs = 1 + #addr_inputs
#outputs = 1 + <width>
```

- For MPW:

```
#inputs = 2 + #addr_inputs + <width>
#outputs = 1
```


© 2021 All Rights Reserved.

- For MPW_BS:

```
#inputs = 2 + #addr_inputs + <width> + <width>/8
#outputs = 1
```

Note: The <*width*> of MPW_BS must be divisible by 8 (such as 8, 16, 24, ...).

`addr_inputs` can be calculated as follows:

```
 <depth1>             2   4   8   16   32   64   128   256   512
 <suffix>
-------------------------------------------------------------------------------
     -                1   2   3    4    5    6    7     8     9
#addr_inputs          1   2   3    4    5    6    7     8     9
 <depth2>K        1   2   4   8  16  32  64 128 256 512
-------------------------------------------------------
#addr_inputs     10  11  12  13  14  15  16  17  18  19
 <depth3>M        1   2   4   8  16  32  64 128 256 512
-------------------------------------------------------
#addr_inputs     20  21  22  23  24  25  26  27  28  29
 <depth4>G        1   2   4   8  16  32  64 128 256 512
-------------------------------------------------------
#addr_inputs     30  31  32  33  34  35  36  37  38  39
 <depth5>T        1   2   4   8  16  32  64 128 256 512
-------------------------------------------------------
#addr_inputs     40  41  42  43  44  45  46  47  48  49
 <depth6>P        1   2   4   8  16  32  64 128 256 512
-------------------------------------------------------
#addr_inputs     50  51  52  53  54  55  56  57  58  59
 <depth7>E        1   2   4   8  16  32  64 128 256 512
-------------------------------------------------------
#addr_inputs     60  61  62  63
```

Note: K stands for kilo (2 power 10); M stands for mega (2 power 20); G stands for Giga (2 power 30); T stands for Tera (2 power 40); P stands for Peta (2 power 50); E stands for Exa (2 power 60);

The minimum depth is 2 (aw = 1). The maximum depth is 8E (aw = 63).

Examples:

```
MPR16Kx4  has 15 (1+14)   inputs and 5 (1+4) outputs.
MPW8X32   has 37 (2+3+32) inputs and 1 output.
MPR512Mx8 has 30 (1+29)   inputs and 9 (1+8) outputs.
```


© 2021 All Rights Reserved.

Running Designs with Memories

This section describes the detailed information on loading and dumping memories.

Both memory loads and memory dumps can be run in background. Use the `memory` command option `-bg` to run the memories in background. The operation is added to a task queue and the command is returned immediately without waiting for the memory operation to finish. It can also be added to a group of memory load and dump commands, which can improve performance.

If a memory dump or load is run without using the background option, it is a foreground operation. Use the `memory -fg` command to run in foreground, and wait for all background tasks to finish. If an error is found in the background tasks, an error message is generated.

Global Memory Options

User-defined options applied to all subsequent memory operations are known as global options. Global options append to other options. Global options that conflict with separate memory commands can be overridden by specifying the desired memory option as a separate memory command, on its own line.

The following options are global options:

- `-append`
- `-clearAddrReg`
- `-full`
- `-skipzero`
- `-z`
- `-dir`
- `-background (or -bg)`

The following command options can be used to manipulate the global options:

- `memory -opt <option_specifications>`
- `memory -optrm <option_specifications>`
- `memory -optclear`

© 2021 All Rights Reserved.

The `memory -opt` command with no specifications returns a string showing the current settings of the global options. With option specifications, it adds the specified options to the existing set of options. Use the `memory -optrm <option_specifications>` command to remove specified options, or `memory -optclear` to clear all options.

Global stack options enable you to maintain several sets of global options settings. Each set of global option settings is one level on the global options stack. The level of the top of the stack is the current global options. The following commands manipulate the memory option stack:

- `memory -optstack`

  This command shows the values of all levels of global options. Each level is marked with a number, and the top level is marked as number 1.

- `memory -optpush`

  Create a new set of global options. This command option pushes a new level of the stack top, a duplicate of the current stack top.

- `memory -optpop`

  Pop a level of global options. This command option removes a set of global options. Removes the top level and the second level becomes the new top level.

- `memory -optuse <n>`

  Bring level `<n>` to the top of the memory stack. The order of all other levels will be maintained. Thus, the command `memory -optuse 3` will bring level 3 to the top of the stack, but levels 1 and 2 will retain their order.

The Memory Address Register

The memory address register is the register that drives the write address of a memory instance. Various options can be used to specify the range of memory addresses to be dumped from the address register. Some memory instances are used to capture data starting from address 0.

Loading Memories

The system tasks `$readmemb` and `$readmemh` are used to read and load data from text files into memory. As the file is read into memory, each number is assigned a specific word element in the memory. Addressing is controlled by specifying start and/or finish address in the system task invocation as well as by specifying addresses in the data file.

© 2021 All Rights Reserved.

Addresses in the data file appear as an @ character, followed by a hexadecimal number. Both uppercase and lowercase is allowed. When the system encounters and address specification, it loads the subsequent data, starting at that memory address. If no addressing information is specified, and no address specifications are found in the data file, the default start address becomes the left-hand address in the declaration of the memory. Consecutive words are read until either the memory is full or the data file is completely read. If the start address is specified in the task without a finish address, loading will start at the specified start address and continue toward the right-hand address given in the declaration of the memory.

If both start and finish addresses are specified as task parameters, loading begins at the start address and continues to the finish address, regardless of how addresses are specified in the memory declarations.

When addressing information is specified in both the system task and the data file, the addresses in the data file must be within the address range specified by the system task parameters. Otherwise, an error message is generated and the load operation is terminated. A warning message is issued if the number of data words in the file differs from the number of files in the range implied by the start through finish addresses.

For more information, refer to IEEE Standard 1364-2001.

The `memory -load` command loads the memory into the specified file. In XEL, it uses the following syntax:

```
memory -load {[<file_format>] <memory_name> -file <file_name> [-retainValue] \
             [-retainValueOffset]} \
             [-start <start_number>] [-end <end_number>] \
             [-fileoffset <offset_number> | continue] [-dir <directory_name>] \
             [-background|-bg] [-nofill] [-nochecksize]
```

Loading/Dumping Memory

In addition to using the `memory` command with the `-load` or `-dump` option to load or dump memories, you can use the options provided in the *Memory* tab of the xeDebug GUI to load and dump the required memories. For details about this tab and its options, refer to Memory Tab in SA Mode, Memory Tab in Vector Debug Mode, and Memory Tab in STB and LA Modes sections in the Using the xeDebug GUI chapter.

© 2021 All Rights Reserved.

Figure 10-1 Memory Tab in the xeDebug GU
I

Select the AutoView option in the *Memory Action Control* group box in the *Memory* tab of the xeDebug GUI to launch the *Memory Content* window that lists all the memories in the design. The contents of this window can be edited and then written back to memory by using the *Save* button in the *Memory Content* window.

Figure 10-2 displays the *Memory Content* window.

© 2021 All Rights Reserved.

Figure 10-2 Memory Content Window

Dumping Memories

The `memory -dump` command reads data from the memory instance and dumps it to the specified file. In XEL, it uses the following syntax:

```
memory -dump {<file_format> <memory_name> -file <file_name>} [-append] \
       [-start <start_number>] [-end <end_number>] [-nooverwrite] [-skipzero] \
       [-addrReg <register_name>] [-clearAddrReg] [-full <number>] \
       [-addrRadix <radix>] [-dataRadix <radix>] [-z] [-gz] \
       [-dir <directory_name>] [-background | -bg]
```

Optional starting and ending addresses can be specified for memory data to be dumped. The `-skipzero` option will ignore all lines whose data consists of zeros. The `memory -dump` command must specify the file format. Some memory options can only be used with specific formats. For more information, refer to the *VXE Command Reference Manual.*

As not all memory data is useful, it might be desirable to use options to specify the range of memory desired, using the additional `memory -dump` command options `-addrReg <register_name>`, `-clearAddrReg`, and `-full <number>`. The `-addrReg` option defines the address range to dump, from address 0 to the current value of the specified register. It cannot be used in connection with the `-start` and `-end` options. The `-full` option can be combined with `-addrReg` to indicate memories that should be dumped if they

© 2021 All Rights Reserved.

are more than <*number*> percent full. This can be calculated using the current value of the address register and the memory depth. Use the `-clearAddrReg` option to clear the register after dumping the memory.

The following example shows the use of these options:

```
memory -dump %pd_raw Top.mem1 -file mem1.dump -addrReg Top.mem1_wa[19:0]
       -clearAddrReg -full 90
```

This command dumps Top.mem1 in raw format. The memory address range to be dumped is from 0 to register Top.mem1sa[19:0]. This memory is not to be dumped unless the address register has a value greater than 9437184, which is 90% of the memory depth. The memory is to be cleared after dumping.

Specify the `-z` option to save the data file in compressed format, as generated by the UNIX compress command. It applies only to the raw or raw2 formats. Contents are written to a file with the `.Z` suffix. The file can be uncompressed through UNIX `uncompress` or `zcat` commands. Use of `-z` cannot be combined with the `-append` or `-fileoffset` raw2 format option.

The output of the `memory` command can be difficult to read if the design has a large number of memories. To make reading memory names easier, execute the following Tcl commands:

```
set memlist [memory]
foreach item $memlist {
puts $item
```

Using Memory Streaming

Memory streaming simplifies the use of large data stream files in the in-circuit emulation. A data stream file can be used either as an input or output to the emulation.

If the data stream file is used as an input to the emulation, data is transferred from the file to the specified design memory in *chunks*.

Typically, one *chunk* is enough to fill the memory. When the design has finished processing that *chunk*, another *chunk* is transferred, filling the memory again. The *chunks* are taken in sequence from a file that contains enough data to fill the memory many times.

If the data stream file is an output from the emulation, data is transferred from a design memory to the file in *chunks*. Each *chunk* is appended to the file, so that eventually the file contains a stream of data much larger than the size of the memory itself.

The XEL commands to use memory streaming are as detailed in the following sections.

© 2021 All Rights Reserved.

Using XEL to Write Memories

The following command shows the syntax for writing memories.

```
memory -load %pd_raw2 <inst> -file <file> -fileOffset <offset> [-nofill]
       [-start <start>] [-end <end>]
```

To use the `memory -fileOffset <offset>` or `-getFileOffset` options, or the `-nofill` option the memory contents file must be in `raw2` format (`%pd_raw2`). Otherwise, using the `-fileOffset` and `-getFileOffset` options will generate errors.

The file can be created using `memTran` as described later in this section.

`<offset>` can be either one of the following:

- `<number>` - Takes data from the file starting from byte <number>, where a byte is 8 bits. When memTran creates a file in raw2 format, if the width specified in the memTran command line is not a multiple of 8 bits, memTran automatically pads the data so that each word takes a whole number of bytes in the file.
- `continue` - Takes data from the file, starting at the byte immediately after the last byte in the file taken in a previous operation.

If `-nofill` is specified, the address range from end of the `%pd_raw2` file to end of memory is not changed. Otherwise this range will be filled with 0s.

If the `-start` and `-end` parameters are specified, the command only takes enough bytes from the file to fill the section of the memory specified by `-start` or `-end`. Unlike `<offset>`, the start/end is measured in words, where the word width is determined by the memory width.

The `-getFileOffset` parameter returns the `<offset>` in the file from which the next data will be taken. If no data remains in the file, -1 is returned. Usually, a script uses `-getFileOffset` only to test whether any data remains in the file.

If the file does not contain enough data to fill the memory, the command fills part of the memory with data from the file, then fills the rest of the memory with zeros. To fill only part of the memory, use `-start` and `-end`.

The software remembers one file offset for each file being written, rather than one for each (instance,file) pair. Therefore, to use `-fileOffset continue` to load two memories independently from the same file, you must make two copies of the file (although one can be a soft link).

Using XEL to Read Memories

The following command shows a sample of the syntax for reading memories.

© 2021 All Rights Reserved.

```
memory -dump %pd_raw2 <inst> -file <file> [-append]
       [-start <start>] [-end <end>]
```

The `-append` parameter gives an error if the format is not raw2. Using `-append`, the file is appended. Without `-append`, the file is overwritten.

The following shows a section of a run-time script that repeatedly loads data from file `in_file.raw2` to memory `in_mem`, runs, and dumps data from memory `out_mem` to file `out_file.raw2`. The process repeats until there is no more data in `in_file.raw2`.

```
# This part of the script assumes the logic analyzer has been
# set up to trigger each time the design finishes processing
# the data from in_mem. For example, we could trigger on in_mem
# read address having a particular value and clock having a rising edge.
rm -f out_file.raw2
while {[memory -getFileOffset in_raw2]!=-1} {
        memory -load %pd_raw2 in_mem -file in.raw2 -fileOffset continue
        run
        waitWhileBusy
    memory -dump %pd_raw2 out_mem -file out_file.raw2 -append
}
```

Initializing and Masking Memories

You can initialize a range of memory instances, or all the memory instances to a given value using the following command:

```
memory -setvalue {-all | <list_of_instances>...} [-start <addr1>] [-end <addr2>]
       -value {<value> | all0 | all1}
```

The value can be set to `0`, `1`, or any other value. To set a value other than `0` and `1`, ensure that the value is specified in a format that follows Verilog, VHDL, or C++ (0x hex only) convention.

The memory instances can also be initilized with random numbers, based on the integer value specified as the seed number using the `-random <seed> option`.

```
memory -setvalue {-all | <list_of_instances>...} [-start <addr1>] [-end <addr2>]
       -random <seed>
```

You can also overwrite the bit position of the memory words by specifying the `-mask` option with the `memory -setvalue` command. The `-mask` option specifies a mask value that indicates whether that bit position of a memory word should be overwritten by the corresponding bit as specified by the `-value` option..

```
memory -setvalue -value {<value> | all0 | all1} [-mask <value>]
```

Translating Memory Format Using the memTran Utility

By using the `memTran` utility, you can translate between different file formats.

© 2021 All Rights Reserved.

The command syntax is:

```
memTran -translate [in_format] <in_file> [out_format] <out_file> \
        [<memory_dimension_options>] [<format_specific_options>]
```

The format string is identical to the format string accepted by the `memory` command. As some formats (such as ASCII, memtran, raw, and Xcelium) can identify themselves by file format, the input format is optional. Input format must be different from the output format.

Options for Specifying Memory Dimensions

Memory dimension options specify the width and depth of the memory instance and address ranges whose data are specified in the memory content file. This information is included in some file formats.

If the output format requires information that the input format does not contain, either default values are used, or you must specify the appropriate command-line option. If no option is specified, and a default value is not available, an error message is generated. The following table lists the options and available file formats for conversion, as well as whether the file of that format contains the information, and if a default value is available. If “No” is indicated, the file format does not contain the information, and cannot provide a default value.

	memtran	**raw**	**raw2**	**PD ASCII**	**readmemb/ readmemh**	**Xcelium**
-width	Yes	Yes	No	No	No	No
-depth	Yes	Yes	end-start+1	end-start+1	end-start+1	end-start+1
-start	Yes	Yes	0	Yes	0	0
-end	Yes	Yes	Last addr in file	Yes	Last addr in file	Last addr in file

© 2021 All Rights Reserved.

Using Format-Specific Options

The following format-specific options are available:

Option	Format	Use
`-p <passwd>`	%pd_memtran	Specifies password for file encryption. If the input file is encrypted, a correct password must be specified to convert the input file to another format. If the output file is in %pd_memtran format and a password is specified, the output file is encrypted. To specify a password for both the input and output file, use two `-p <passwd>` options, one for the input file and one for the output file.
`-skipzero`	PD ASCII	Filters out words containing all-zero data. Used with output files in PD ASCII (`%pd_b`, `%pd_o`, `%pd_d`, `%pd_h`).

Comparing File Formats

To compare file formats, use the command-line syntax:

```
memTran -compare [format1] <file1> [format2] <file2> [-summary] [-width <n>]
        [<format_specific_options>]
```

If the file can identify its format type, format specification is optional. If neither file being compared contains width information, you just use the `-width` parameter to specify memory width.

If the `-summary` option is specified, the utility will only print a one line summary message, for example:

```
file1 and file2 same
```

Or

```
file1 and file2 differ
```

Otherwise, a full report of the differences is printed, as shown in the example below:

```
00000000:  05 06              06 00
00000007:  15 16              16 00
0000000ff: 25 26              26 00
```


© 2021 All Rights Reserved.

Each line describes a single word. It shows the address, content of the memory word in the first file, in hexadecimal, and content of the memory word in the second file in hexadecimal. Whether or not the `-summary` option is specified, you can check the return code of the `memTran` program. In the C shell, it is $?. A 0 means there is no difference in the files.

Using memTran to Translate to and from raw2 File Formats

You can use the `memTran` program to convert a file to raw2 file formats. Following is the syntax of the memTran command:

```
memTran -translate <input_format>] <in_file> <%pd_raw2> <out_file>
        [<memory_dimension_options>] [<format_specific_options> ...]
```

A file in `readmemh` format uses one line of `hex` data for each memory word. A file with 12-bit-wide words would contain lines like the following:

```
000
123
456
789
abc
def
```

The width of the words in the file should match the width of the memory to be written; but the depth of the file need not match the memory depth.

Suppose the `readmemh` file is named `in_file.readmemh`, and the words are 12-bits wide. To create the raw2 file named `in_file.raw2`, use the following command:

```
memTran -translate %readmemh in_file.readmemh %pd_raw2 in_file.raw2 -width 12
```

To convert a raw2 file to readmemh format, use a command such as the following:

```
memTran -translate %pd_raw2 out_file.raw2 %readmemh out_file.readmemh -width 12
```

Files in raw2 format do not contain memory-width information (nor any other header information). You must specify the correct `-width` information to the `memTran` command, and must also, when using `memory -load`, ensure that the file has the correct width for the memory being written.

To convert from a Palladium Z1 text file to raw2 format, use the following command line:

```
memTran -translate ascii_file %pd_raw2 bin_file -width 12
```

Use the following command to compare raw2 format files:

```
memTran -compare %pd_raw2 bin_file1 %pd_raw2 bin_file2 -width12
```

Note: The `.bin` extension is no longer supported.

© 2021 All Rights Reserved.

Emulating TCAM Memories with Palladium Z1

Ternary Content-Addressable Memory or TCAM needs special handling for emulation. If the compiler treats TCAM as a general case of a multi-port memory, the TCAM implementation is quite inefficient, both in terms of the required number of domains and by the performance. The Palladium Z1 emulator supports TCAM and facilitates an efficient implementation of TCAM.

This section describes how users can represent the TCAM part of the design to take advantage of the TCAM-aware compilation, thereby increasing the design performance and reducing the required configuration.

Representing TCAM to the Compiler

The compiler supports TCAM with two memory instances:

- `tcam_mem_X`
- `tcam_mem_Y`

Use a pair of bits `tcam_mem_X[i][j]` and `tcam_mem_Y[i][j]` to encode a ternary value stored in TCAM row `i` and column `j`.

> **Note:** The names `tcam_mem_X` and `tcam_mem_Y` are just examples; users can choose any other names.

TCAM performs the search operation by comparing a given pattern under a given mask against all the TCAM words. The pattern and the mask are the TCAM inputs.

TCAM has a set of *match* outputs, one output per row. Each output is set to `1` if the corresponding row value matches the pattern and mask. If the value does not match, the output is set to `0`.

Both `tcam_mem_X` and `tcam_mem_Y` have one read or write port to read or write these by address. It is possible to execute the TCAM search operations, and either read memory by address or write memory by address, within the same emulation cycle.

TCAM width and depth can be provided as Verilog parameters.

Defining and Instantiating TCAM

To specify a TCAM as a part of the design, define TCAM as an empty module with the following Verilog integer parameters:

© 2021 All Rights Reserved.

- *DATA_WIDTH* - TCAM data width
- *ADDR_WIDTH* - number of TCAM address bits
- *DEPTH* - number of TCAM words, if the size of TCAM is less than `2** ADDR_WIDTH`. For instance, `ADDR_WIDTH` might be 10 bits, but TCAM might take only 768 words out of 1024.

  When searching for a match, TCAM performs a logic function of four arguments:

  - Memory *X* bit
  - Memory *Y* bit
  - Pattern word *pattern* (TCAM inputs)
  - Mask word *mask* (TCAM inputs)

  The function is the following:

```
match [i]= (~X[i] & ~Y[i]) | (pattern & ~X[i] & Y[i]) |
           (~pattern & X[i] & ~Y[i]) | ~mask
```

The `match` is `1` for the TCAM word (the word matches the pattern) when the function generates `1` for all the bits of the word.

To modify the semantic of pattern and mask bits, define a wrapper around the TCAM cell, and invert pattern and/or mask inputs within the wrapper.

The inputs of the module (the order is essential) are:

- Pattern to search for
- Mask to compare under
- Read/write address for memory *X*
- Read/write address for memory *Y*
- Write data for memory *X*
- Write data for memory *Y*
- Write enables for memories *X* and *Y*: *WE_X* and *WE_Y*

The outputs of the module (the order is essential) are:

- Match bits, one per TCAM word
- Read data from memory *X*
- Read data from memory *Y*

© 2021 All Rights Reserved.

Inside the module, you must include the following pragmas to pass the TCAM parameters to the compiler:

```
// quickturn write_parameters
// pragma CVASTRPROP MODULE HDL OBJECT_MODULE "TCAM"
// pragma CVASTRPROP MODULE HDL SAVE_RESTORE_OK "1"
// pragma CVASTRPROP MODULE HDL WE_TERM "WE_X WE_Y"
// pragma CVASTRPROP MODULE HDL memory1 "tcam_mem_X ADDR_WIDTH DATA_WIDTH"
// pragma CVASTRPROP MODULE HDL memory2 "tcam_mem_Y ADDR_WIDTH DATA_WIDTH"
```

Here, you can substitute *tcam_mem_X* and *tcam_mem_Y* with any of the user-defined names. Do NOT modify any of the other pragma text.

Example of a TCAM Cell

```
module tcam #(
        parameter integer DATA_WIDTH = 18,
        parameter integer ADDR_WIDTH = 10,
        parameter integer DEPTH = 2**ADDR_WIDTH
    )
        // quickturn write_parameters
        // pragma CVASTRPROP MODULE HDL OBJECT_MODULE "TCAM"
        // pragma CVASTRPROP MODULE HDL SAVE_RESTORE_OK "1"
        // pragma CVASTRPROP MODULE HDL WE_TERM "WE_X WE_Y"
        // pragma CVASTRPROP MODULE HDL memory1 "tcam_mem_X ADDR_WIDTH DATA_WIDTH"
        // pragma CVASTRPROP MODULE HDL memory2 "tcam_mem_Y ADDR_WIDTH DATA_WIDTH"
    (
        input [DATA_WIDTH-1:0] pattern,
        input [DATA_WIDTH-1:0] mask,
        input [ADDR_WIDTH-1:0] addr_X,
        input [ADDR_WIDTH-1:0] addr_Y,
        input [DATA_WIDTH-1:0] write_data_X,
        input [DATA_WIDTH-1:0] write_data_Y,
        input WE_X,
        input WE_Y,
        output [2**ADDR_WIDTH-1:0] match,
        output [DATA_WIDTH-1:0] read_data_X,
        output [DATA_WIDTH-1:0] read_data_Y
    );
endmodule
```

Multiple modules might be defined with different TCAM width or depth. Instantiate the module in which the TCAM instance with the specified width and depth is required, and accordingly connect the inputs and outputs.

Using TCAM in SA Mode

To use the TCAM cell in a design that needs to be compiled using IXCOM for SA mode, use the `ixc_embed_hw` pragma to enable the IXCOM compiler to pass the TCAM cell to xeCompile.

© 2021 All Rights Reserved.

Typically, for TCAM usage in SA mode, using the pragmas as follows is sufficient:

```
//pragma ixc_embed_hw footer
            <TCAM_cell_specification_for_xeCompile>
//pragma ixc_embed_hw endfooter
```

To build a complete TCAM module with software simulation and hot swap, while using the TCAM cell as the basic building block, you need to provide both simulation model and hot-swap links.

Refer to the example of hot-swap-capable `ixc_tcam` model defined in `${IXCOM_INSTALL_HOME}/etc/ixcom/IXCclkgen.sv` that is built from the hardware TCAM cell as listed below:

```
module ixc_tcam (
      pattern, mask, addr_X, addr_Y, write_data_X, write_data_Y, WE_X, WE_Y, match,
                read_data_X, read_data_Y);

   parameter integer DATA_WIDTH = 64;
   parameter integer ADDR_WIDTH = 7;
   parameter integer DEPTH = 2**ADDR_WIDTH;

   input [DATA_WIDTH-1:0] pattern;
   input [DATA_WIDTH-1:0]  mask;
   input [ADDR_WIDTH-1:0]  addr_X;
   input [ADDR_WIDTH-1:0]  addr_Y;
   input [DATA_WIDTH-1:0]  write_data_X;
   input [DATA_WIDTH-1:0]  write_data_Y;
   input                   WE_X;
   input                   WE_Y;
   output [DATA_WIDTH-1:0] read_data_X;
   output [DATA_WIDTH-1:0] read_data_Y;
   output [DEPTH-1:0] match;

   bit [DATA_WIDTH-1: 0] tcam_mem_X[0:DEPTH-1];
   bit [DATA_WIDTH-1: 0] tcam_mem_Y[0:DEPTH-1];

   // pragma ixc_embed_hw footer
   // quickturn write_parameters
   // pragma CVASTRPROP MODULE HDL OBJECT_MODULE "TCAM"
   // pragma CVASTRPROP MODULE HDL SAVE_RESTORE_OK "1"
   // pragma CVASTRPROP MODULE HDL WE_TERM "WE_X WE_Y"
   // pragma CVASTRPROP MODULE HDL memory1 "tcam_mem_X ADDR_WIDTH DATA_WIDTH"
   // pragma CVASTRPROP MODULE HDL memory2 "tcam_mem_Y ADDR_WIDTH DATA_WIDTH"
   // pragma ixc_embed_hw endfooter

   //pragma ixc_embed_sw footer

   bit [DEPTH-1:0] bmatch = {DEPTH{1'b1}};
   bit [DATA_WIDTH-1:0] bpattern;
   bit [DATA_WIDTH-1:0] bmask;
   always@(*)
     bpattern = pattern;
   always@(*)
     bmask = mask;
   assign match = bmatch;
   assign read_data_X = tcam_mem_X[addr_X];
   assign read_data_Y = tcam_mem_Y[addr_Y];

   always@(*)
     if(WE_X)
       tcam_mem_X[addr_X] = write_data_X;
```


© 2021 All Rights Reserved.

```
   always@(*)
     if(WE_Y)
       tcam_mem_Y[addr_Y] = write_data_Y;

   genvar i;
   generate for(i = 0; i < DEPTH; i++)
     always@(*)
     bmatch[i] = (~((bpattern&~tcam_mem_X[i])|(~bpattern&~tcam_mem_Y[i])|~bmask)
                  == 0);
   endgenerate

//pragma ixc_embed_sw endfooter

`ifdef AXIS
   initial $ixc_ctrl("hwonly_mem", "tcam_mem_X", DATA_WIDTH, 0, DEPTH-1);
   initial $ixc_ctrl("hwonly_mem", "tcam_mem_Y", DATA_WIDTH, 0, DEPTH-1);
`endif

endmodule // ixc_tcam
```


© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

11

Modeling Clocks in ICE Mode

This appendix provides the guidelines for modeling clocks in ICE mode. For details about modeling clocks in SA mode, refer to Modeling Clocks in SA Mode.

Supported Clock Technologies

In ICE mode, VXE supports two technologies for generating clocks within the emulator:

- CAKE (Clock Averaging with Ko-incident Edges)

  For a given set of clock frequencies, the CAKE technology gives the best-possible emulation speed. Therefore, it is, usually, the best technology to use, and preferred by most users.

- SCM (Simulation Control Module)

  SCM technology, usually, gives slower emulation speed than CAKE because it has more FCLKs per design clock cycle. But, the advantage of using SCM is that it preserves the same edge relationships (that is, coinciding or not coinciding edges), which the clocks of the specified frequencies would have had in simulation, if the duty cycles of all clocks were exactly 50% and the initial rising edges of all clocks coincided.

To specify the technology, use the `clockOption` XEL command:

```
clockOption -add {technology CAKE | technology SCM}
```

For details, refer to the description of the `clockOption` command in the *User Data Commands* chapter of the *VXE Command Reference Manual*.

Specifying Oversampling Ratio for CAKE Technology

When using CAKE technology, you must specify an oversampling ratio. The oversampling ratio is the number of emulator cycles (FCLK cycles) per cycle of the fastest design clock. This ratio can be set to `0.5`, `1`, or any even number. Oversampling ratio setting of 0.5, 1, 2, and so on is also referred to as CAKE 0.5X mode, CAKE 1X mode, and CAKE 2X mode, respectively.

© 2021 All Rights Reserved.

The smaller the number is, the faster the design runs. However, a small oversampling ratio can have disadvantages.

The oversampling ratio of `0.5` is good for designs in which the fastest design clock, clocks only a small fraction of the design and certain additional restrictions on clocking are met.

The oversampling ratio of `1` can result in a large capacity cost for some designs. You can try to reduce this capacity cost by providing extra information (shadowsData) to the compiler.

Some designs contain asynchronous loops. In such designs, an oversampling ratio higher than `1`, such as 2 or 4 might be needed to make the design function correctly.

For more information about CAKE and SCM technologies, see the description of the `clockOption` command in the *VXE Command Reference Manual*.

Using the 1X Mode

1X mode is a Palladium Z1 compilation mode that allows the fastest design clock to run at the same frequency as the FCLK of the emulator. Without the use of 1X mode, the fastest design clock would be running at a speed at least two times slower than FCLK.

In most designs, compared to the 2X mode, using the 1X mode doubles the emulation speed and logic analyzer upload speed, with little or no capacity cost. However, in some of these designs, you must provide extra information about internal clock signals to the compiler to avoid the capacity cost for 1X mode.

How the 1X Mode Works

Palladium Z1 emulators operate by periodically evaluating (or, computing a value for) each net in a design. The value of each net is computed once per cycle of an (imaginary) clock called FCLK. The FCLK frequency depends on the design.

Except for certain target I/O signals, the Palladium Z1 emulator evaluates each net in its input netlist only once per FCLK cycle. However, in 1X mode, the fastest design clock, and possibly other signals, must change twice per FCLK cycle.

To make this possible, in 1X mode, the precompile step of compilation transforms the design as follows, when creating the netlist for the back-end compiler:

For each net in the design, the precompiler determines the edges of the fastest design clock at which the net can change value. If a net can potentially change value at both rising and falling edges of the fastest design clock, two nets (called "net" and "shadow net") are created. These nets represent the value of the net in the high and low (or "active" and "passive")

© 2021 All Rights Reserved.

phases of the fastest clock. The "net" and "shadow net" are evaluated in parallel, once per FCLK cycle.

The Palladium Z1 run-time software (xeDebug, waveform viewer, etc.) hides the existence of shadow nets. Waveforms for all nets in the design are displayed correctly, as though running in 2X mode. The waveform shows two time points for each FCLK cycle.

The CAKE 2X mode or 2X mode (or oversampling ratio set to 2) means that there are two FCLK cycles per cycle of the fastest design clock.

For each net that has a shadow net, at precompile stage, a separate logic must be created to compute the net and the shadow net. If a large fraction of the nets in a design have shadow nets, the capacity cost for 1X mode will be substantial. However, for most designs, only a small fraction of design nets have shadow nets because, typically, the values of only a few design nets change on the falling edge of the fastest design clock. As a result, the capacity cost for 1X mode is usually less than 10%. Most often, it is less than 1%.

There are situations in which shadow nets are created at precompile stage, even though you have not specified CAKE 1X mode. For example, if you have enabled multisampled I/O and declared at least one multisampled input signal, during precompile stage, shadow nets are created irrespective of whether or not you have specified CAKE 1X mode. Whenever shadow nets are created at precompile stage, the waveform automatically displays two time points per FCLK; otherwise it displays one time point per FCLK cycle.

In this appendix, generically, the "1X mode" term is used to describe a compilation in which shadow nets are created, and the "2X mode" term is used for a compilation in which the shadow nets are not created. The term "CAKE 1X mode" or "CAKE 2X mode" is used to mean that you have explicitly declared CAKE 1X or 2X mode.

By default, the sampling of design inputs connected to a target, and updates to the design outputs connected to a target, happen only once per FCLK cycle. Exceptions to this rule are the Multipulse and Pulse outputs to a target specified by using the `-edgeInfo` and `-pulse` options of the `terminalTiming` command, respectively. However, if you enable the multisampled I/O feature, inputs from a target can be sampled, and outputs updated, twice per FCLK cycle. When this feature is enabled, the compiler divides the FCLK cycle into two parts. For a multisampled input, the physical input from the target is sampled once during each of the two parts of the FCLK cycle, and the sampled values are transferred to net and shadow respectively. For a multisampled output, the physical output to the target is updated once during each part of the FCLK cycle, using the values of net and shadow, respectively.

A special CAKE 0.5X compilation mode is available, which allows the fastest design clock to run at double the frequency of FCLK. Usually, for this mode to be useful, the highest-frequency clock of the design must clock only a small fraction of the design, and the

© 2021 All Rights Reserved.

design must use only one edge of the highest-frequency clock. CAKE 0.5X mode is described in Using the CAKE 0.5X Mode.

When to Use 1X Mode

1X mode can be used for most designs having the following characteristics:

- The design has multiple clock sources with different, arbitrary, frequencies.
- The design uses complex multiplexing, gating and dividing circuits to generate internal clocks from the clock sources.
- The design uses both edges of its clocks.
- Input and output signals connected to an external target change at every cycle of corresponding design clocks (including, possibly, the fastest design clock), or even twice per cycle.
- Some design clocks (or other design inputs) are driven by sources external to the emulator, asynchronous with FCLK.

When Not to Use 1X Mode

Obstacles to using 1X mode can include the following:

- For a specific design, 1X mode might result in too much capacity cost. You can reduce this cost by using the techniques described in 1X Mode Capacity Cost.
- When a design contains combinational loops, precompile breaks the loops by inserting delays. Sometimes, the loopbreaking delay prevents a design from working correctly unless the oversampling ratio (number of FCLK cycles per design clock cycle) is increased. Such a design might not run correctly in 1X mode. Avoiding the Need for Oversampling in the Appendix 12, “Loop-Breaking” describes techniques to avoid problems caused by loopbreaking delay.
- A *1X-type clock source* should be generated using Palladium Z1's CAKE clock generation technology. (A *1X-type clock source* refers to a clock source that changes value more than once per FCLK.) If the design and target share common 1X-type clock sources, the emulator, not the target, should drive these clock sources. By using the multisampled I/O feature, it is possible to have a 1X-type clock driven by a source external to the emulator, asynchronous with FCLK. However, there will be a significant capacity cost (probably double) for any logic associated with this clock source in the emulated design.

© 2021 All Rights Reserved.

- If a design interfaces with an external target, for 1X mode, you might need to specify I/O timing constraints, which are not required in 2X mode. This is not really an obstacle; it is merely an extra task required for 1X mode. If a 1X-mode design interfaces with an external target, and the clocks associated with the target interface are at least two times slower than the fastest clock, then extra I/O timing constraints for 1X mode are usually not required. Details are provided in Specifying a 1X-Mode Target Interface.

When emulating a design for the first time, Cadence recommends that you use the CAKE 2X (or higher) mode initially, to enable easy bring-up of the design.

Clocks in CAKE 1X Mode

A design compiled in CAKE 1X mode can have multiple clock sources with arbitrary frequencies, driven automatically using the CAKE clock-generation technology. The following diagram shows waveforms of CAKE 1X mode clock sources named CK6, CK5, and CK3, with relative frequencies 6, 5, and 3, respectively.

A clock source like CK5 with frequency less than, but more than half of, the fastest clock frequency, has a waveform that coincides with that of the fastest clock in some FCLK cycles, and is flat low in the remaining FCLK cycles. Since every rising edge of CK5 coincides with a rising edge of CK6, the output of a flip-flop clocked by rising edge of CK5 does not need a shadow net.

A clock source like CK3 with frequency no more than half that of the fastest clock is a "2X-type" clock source. Neither CK3, nor outputs of flip-flops clocked by it, need shadow nets.

Upgrading from 2X to 1X Mode

For most designs, the steps to upgrade from 2X mode to 1X mode are as follows:

1. Change the compile script to specify CAKE 1X mode. (See Specifying CAKE 1X Mode.)

© 2021 All Rights Reserved.

2. If the design connects to an external target, you might need to change the compile script to specify a 1X-mode target interface. (See Specifying a 1X-Mode Target Interface.)

3. Run and debug the design.

   If the design works correctly in 2X mode but not in 1X mode, perform the following steps:

   - Run `precompileOption -add breakHalfCycleAll` command. Remove `precompileOption -add noReset1X`, if you are using `noReset1X`. Performing these steps might adversely affect emulator capacity and FCLK speed. If these steps enable correct function but the cost is too much, try each step individually.
   - Ensure you have specified the target interface correctly. Refer to "Specifying a 1X-Mode Target Interface" section for more information.
   - If a run script contains the command `run` <*n*>, and the `timeUnit` is the ICE-mode default of `fclk`, then while moving from 2X mode to 1X mode, the value of <*n*> should be decreased by a factor of 2. For example, `run 100` should be changed to `run 50`. If the `timeUnit` is something other than `fclk` (for example, `dclk` or `ns`), the value of <*n*> should not be changed when moving from 2X mode to 1X mode.

   If the steps above do not enable the design to work correctly in 1X Mode, then you must use FullVision to isolate the problem, just like you would resolve design problems.

4. Improve the compile results (capacity and step count), if necessary. (See 1X Mode Capacity Cost.)

   Steps 1, 2, and 4 are described in more detail in the following sections. Steps 3 and 4 are usually independent, and can be performed in any order.

Specifying CAKE 1X Mode

The following XEL commands specify CAKE 1X mode for an example design with three clocks named CK6, CK5, and CK3. These commands are given before precompile.

```
clockOption -add {technology CAKE 1}
clockOption -add {activeEdge positive}
clockOption -add {ignoreOutput on}
clockOption -add {ignoreEvent on}
clockFrequency -add {CK6 6}
clockFrequency -add {CK5 5}
clockFrequency -add {CK3 3}
```

The `{technology CAKE 1}` option specifies CAKE 1X mode. In CAKE 1X mode, the fastest design clock has one cycle in each FCLK cycle.

The `{activeEdge positive}` option is appropriate for nearly all designs. The `{activeEdge negative}` option would be appropriate (that is, would give a smaller 1X mode capacity cost) for an unusual design in which most design signals changed on the

© 2021 All Rights Reserved.

falling edge of the fastest clock. If you do not specify the active edge, it is, by default, taken as `positive`.

The `{ignoreOutput on}` option is used in this example, but it is only needed in certain designs. When using CAKE 1X mode, if you enable the multisampled I/O feature (See Using the Multisampled I/O Feature), then you do not need to specify the `{ignoreOutput on}` option.

If you do not enable the multisampled I/O feature, the following rules apply.

- Outputs to an external target (except multipulse and pulse outputs) can change at most once per FCLK cycle. Inputs from an external target are sampled once per FCLK cycle, and do not have shadow nets.
- If a design output (even one not connected to an external target) has a shadow net, precompile will fail with an error unless you specify the `{ignoreOutput on}` option. An output that is not connected to a target still causes an error because precompile does not check whether or not an output is connected to a target. If you specify the `{ignoreOutput on}` option, and a design output with a shadow net is connected to a target, the net (not the shadow net) will be used as the output to the target. The waveform of such an output still shows two samples per FCLK, showing the values of the net and shadow net.

In the example above, there are three CAKE-generated clocks CK6, CK5, and CK3, with relative frequencies 6, 5, and 3 respectively. Any design logic driving these CAKE clock signals is automatically disconnected, and the signals are driven by CAKE clock-generation logic instead.

1X-mode Terminology: Active and Passive Phases

Precompile messages in 1X mode use the terms “primary clock, “active phase” and “passive phase”.

If CAKE 1X mode is specified, the primary clock is the highest-frequency clockSource.

One cycle of the primary clock (which in 1X mode, takes place during one FCLK cycle) consists of, by definition, the active phase followed by the passive phase. With activeEdge positive, the active and passive phases are the high and low phases of the fastest clock, respectively. With activeEdge negative, the active and passive phases are the low and high phases. If a user net can have different values in the active and passive phases, it must have a shadow.

The terms “active phase” and “passive phase” always refer to the phases of the primary clock.

© 2021 All Rights Reserved.

Specifying a 1X-Mode Target Interface

When a design compiled in 1X mode has input or output signals (or both) connected to an external target, and some of these signals change more than once per cycle of the primary clock, the target interface is said to run in 1X mode. If the primary clock itself connects to the target, then (since the primary clock changes twice per cycle of the primary clock) the target interface runs in 1X mode. When the target interface runs in 1X mode, additional terminal timing commands are needed, that are usually not needed otherwise.

The following changes might be required:

- Specifying Clock Outputs
- Specifying `terminalTiming` constraints on inputs, to avoid output-to-input delays (Constraining Inputs)
- Specifying multisampled I/O, if needed (Using the Multisampled I/O Feature)

These changes are described in the following sections.

Specifying Clock Outputs

If the primary clock, or any other clock with frequency faster than half that of the primary clock, is an output to the target, user must implement each such clock output using one of the following methods:

- Implement the clock output as a multipulse output. Since a multipulse clock (as seen at the target) has the same waveform in every FCLK, this method can implement the primary clock but cannot implement other clocks which are flat low in some FCLK cycles.

  The following command specifies `<signal>` as a multipulse output with rising edge at step 0, falling edge at step 128, and low phase duration at least 25 steps.

  ```
  terminalTiming -add {<signal> -edgeInfo 0 25 0 128}
  ```

  For more information, see the description of the `terminalTiming` command in the *VXE Commands Reference Manual*.

  One advantage of using a multipulse clock is that the rising edge can be scheduled at step 0. Scheduling the clock output rising edge as early as possible is desirable, because it allows design inputs that change in response to the clock output to be scheduled relatively early. This can improve overall step count in a design where a design input is on the critical path.

- Implement the clock output using the multisampled I/O feature, which is described later in this chapter. If the emulator is Palladium Z1, and the design has other multisampled I/O signals which change on both edges of the clock output, then user should use the

© 2021 All Rights Reserved.

multisampled I/O feature to implement the clock output as well as the other multisampled signals. This allows you to reliably constrain the timing of the other signals relative to the clock output.

- A third method (which is rarely used) is to implement the clock as a (positive or negative) pulse output. A positive pulse output can implement a clock that goes high, then low, in some FCLK cycles, and is flat low in other FCLK cycles. For more information, see the description of the `terminalTiming` command in the *VXE Commands Reference Manual*.

Constraining Non-Clock Outputs

Almost always, non-clock outputs must be constrained to change at a later step within the FCLK cycle than for corresponding clock outputs. This requirement is exactly the same in 1X mode and 2X mode. For more information, see the example "Design with clock outputs" in the description of the `terminalTiming` command in the *VXE Commands Reference Manual*.

Constraining Inputs

For a target interface where the clock is generated by Palladium Z1, transitions of inputs to Palladium Z1 are generally caused by a transition of the clock output. If you do not specify the input signals as multisampled terminals and do not specify timing constraints for them, the inputs will usually be scheduled early in the FCLK cycle. As a result, there will be a one-FCLK delay between the clock output transition and the input transition as seen by the design in Palladium Z1.

For a 1X mode design in which the target interface clock frequency is more than half the design's fastest clock frequency, a one-FCLK delay from clock output to input is usually not tolerated by the design. For such a design, you must constrain the inputs and the input components of bidirectional signals to be scheduled long enough after the clock output transition to eliminate the clock-output-to-input delay. For more information, refer to the *Avoiding output-to-input delay* and *Avoiding output-to-input delay for bidirectional signals* sections in the description of `terminalTiming` command, *User Data Commands* chapter, *VXE Commands Reference Manual*.

Using the Multisampled I/O Feature

If any input or output other than the primary clock can change more than once per cycle of the fastest clock, use the multisampled I/O feature. If user enables the multisampled I/O feature, the emulator can sample inputs from an external target twice per FCLK cycle, and can change outputs to an external target twice per FCLK cycle.

© 2021 All Rights Reserved.

In some designs, using the multisampled I/O feature could increase the step count with which the design is compiled, thus reducing the FCLK frequency. Use the feature only when it is needed.

To use the multisampled I/O feature, do the following, before the precompile step:

1. Enable the multisampled I/O feature by giving the XEL command

   `compilerOption -add {enableMultiSampledIO <mode>}`

   where `<mode>` is 1 or 2. Specifying a `<mode>` of `0` disables the feature.

   The `<mode>` setting (1 or 2) determines how precompile handles design outputs, including output and enable components of bidirectional signals. If `<mode>` is `1`, any output that has a shadow is multisampled. If `<mode>` is `2`, only those outputs specified with `multiSampledTerminal` are multisampled; if any other outputs have shadows, user must specify the `ignoreOutput on` command, or precompile will fail.

2. For each input signal which must be multisampled, and each bidirectional signal whose input component must be multisampled, declare the signal as a multisampled terminal using the command

   `multiSampledTerminal -add {<signal>}`

   A tristate output signal is considered to be bidirectional, and the waveform shows its input component. When the `enableMultiSampledIO <mode>` setting is `1`, if a bidirectional output or enable component has a shadow (and is therefore multisampled), then the input component is automatically multisampled as well. When the <mode> setting is `2`, declaring a bidirectional signal as a `multiSampledTerminal` makes the output, enable and input components multisampled.

When the multisampled I/O feature is enabled, the compiler divides the FCLK cycle into two parts. In Palladium Z1, the step number where the second part begins is decided by the compiler, so as to minimize total step count. For a multisampled input, the physical input from the target is sampled once during each of the two parts of the FCLK cycle, and the sampled values are transferred to net and shadow respectively. For a multisampled output, the physical output to the target is updated once during each part of the FCLK cycle, using the values of net and shadow, respectively. For either a multisampled input or multisampled output, one can say that the non-shadow signal is scheduled during the first part of the FCLK cycle, and the shadow is scheduled during second part of the FCLK cycle.

Non-multisampled inputs are sampled only once per FCLK cycle, even though the multisampled I/O feature is enabled. Non-multisampled outputs change only once per FCLK cycle.

The `dbFiles/<top_cell>.ctl` file provides the information about which primary I/O signals are multisampled.

© 2021 All Rights Reserved.

If a multisampled signal is scheduled at a certain step N in the first part of the FCLK cycle, its shadow is not necessarily scheduled the same number of steps, N, after the beginning of the second part. Timing constraints apply to multisampled signals in the following way.

- Timing constraints can specify relationships between multisampled and non-multisampled signals, between different multisampled signals, or between multisampled signals and fixed step numbers.
- Timing constraints between different multisampled I/O signals, or between multisampled I/O signals and fixed step numbers, apply to the scheduling of shadows (in the second part of FCLK) as well as to the scheduling of non-shadows (in the first part of FCLK). For example, if A and B are multisampled I/O signals,
  - The commands

    ```
    delayBox -add {after_A 10 A}
    terminalTiming -add {B after_A}
    ```

    constrain B to be scheduled at least 10 steps after A, and constrain B's shadow to be scheduled at least 10 steps after A's shadow.
  - The commands

    ```
    delayBox -add {after_begin 10}
    terminalTiming -add {B after_begin}
    ```

    constrain B to be scheduled at least 10 steps after step 0, and constrain B's shadow to be scheduled at least 10 steps after the beginning of the second part of the FCLK cycle.
- Timing constraints between multisampled and non-multisampled signals apply to the scheduling of non-shadow signals but have no direct effect on the scheduling of shadow signals. For example, if A is a multisampled signal and C is non-multisampled,
  - The following commands constrain C to be scheduled at least 10 steps after the non-shadow A is scheduled, but allow C to be scheduled at any time relative to A's shadow:

    ```
    delayBox -add {after_A 10 A}
    terminalTiming -add {C after_A}
    ```
  - The following commands constrain the non-shadow A to be scheduled at least 10 steps after C is scheduled:

    ```
    delayBox -add {after_C 10 A}
    terminalTiming -add {A after_C}
    ```

    These commands have no direct effect on A's shadow (but A's shadow must be scheduled in the second part of the FCLK cycle, which starts at a step later than the latest non-shadow multisampled signal, including A, is scheduled.)

© 2021 All Rights Reserved.

There is a minimum of one TIF (eight steps) interval between the step at which the last I/O signal is scheduled in the first part of the FCLK cycle, and the step at which the first I/O signal is scheduled in the second part. Likewise, there is a minimum of one TIF (eight steps) interval between the step at which the last I/O signal is scheduled in the second part of the FCLK cycle, and the step at which the first I/O signal is scheduled in the next FCLK cycle.

Using 1X Mode with the multiplexCable Feature

Some Cadence Speedbridges use the multiplexCable feature, which assigns multiple design signals to a single target cable pin. For such a Speedbridge, Cadence provides a set of `delayBox` and `terminalTiming` commands that determine the scheduling order of signals on each pin. The provided commands assume that the signals connected to the Speedbridge do not have shadows.

If the dumpShadows labels of the clock or reset inputs of the speedbridge wrapper module include `p=both`, the signals connected to the Speedbridge will have shadows and compile will fail in `et3compile` with a message saying that the scheduling order of the signals is not fully determined. To avoid this problem, analyze the reason for the `p=both` label on an input of the wrapper module, and modify the design or apply shadowsData so that the label does not include `p=both`.

1X Mode Capacity Cost

In some designs, compiling in 1X mode immediately doubles emulation speed with little or no capacity cost compared to 2X mode.

However, in many designs, there is initially a significant capacity cost for 1X mode. The capacity cost can cause a degradation in compilation step count, so that emulation speed in 1X mode is much less than double the speed in 2X mode.

Most designs do not heavily use the falling edges of their clocks. If a design does not heavily use the falling edges of its clocks but still has a significant capacity cost for 1X mode, a likely reason is that most signals do not change value on falling edge of the fastest clock, but precompile is not able to prove it, so precompile must give these signals shadows. In such a design, user might be able to reduce the 1X mode capacity cost by providing precompile extra information about internal clocks, enabling precompile to infer that outputs of storage elements clocked by these clocks do not need shadows.

The process of reducing 1X mode capacity cost is usually an iterative one involving the following steps:

1. Choose, at random within the dumpShadows report, a storage element "E" whose output has a shadow.

© 2021 All Rights Reserved.

2. Provide precompile additional information to eliminate the shadow on E. Usually this information will eliminate shadows on other storage elements as well. Recompile the design, and verify that storage element E no longer appears in the dumpShadows report.
3. If the capacity cost has been reduced to an acceptable level, stop. Otherwise, repeat the process, starting at step 1.

Details of this process are given in the following sections. Do not give up if the first few iterations make little progress. Sometimes the first few iterations make little progress, and then suddenly an iteration makes excellent progress.

Analyzing 1X Mode Capacity Cost

In the file `tmp/<LIB>.<TOP>.MultiClock`, precompile reports both the gate count for the design, and the number of shadow nets in a 1X mode compile:

```
Size of design in 2-input gates: 4635760
INFO (MultiClock#38396): [host,22369] Created 184636 shadow nets.
```

Usually, the percent capacity cost for 1X mode is roughly three times the number of shadow nets divided by the number of gates, expressed as a percent.

You can identify the design nets that have shadows, and the reasons for this from the `dumpShadows` report. Run precompile with the `-dumpShadows` option (along with any other options normally used). Precompile will create a `dumpShadows` report at `./tmp/precompile.dumpShadows`.

For each net that has a shadow, the dumpShadows report shows the gate (or storage element) driving the net, along with the nets connected to the pins of this gate. CAKE clock sources are an exception: no driving gate is shown for a CAKE clock source. Buffers and inverters are not shown in the report; inversion on an input or output pin is shown by a "`*`" character. Common gate and storage element types include:

FDRSE	D flip-flop
LDRSE	D latch
A2	two-input AND
O2	two-input OR
XO2	two-input XOR
MX2	two-input MUX

© 2021 All Rights Reserved.

For example, for the following circuit, where ck is the fastest CAKE clock source:

Here is the dumpShadows report:

```
A2 U1
<- O gck (sh) p=gnd
-> I1 ck (sh) a=pwr p=gnd
-> I0 ena1
---
O2 U2
<- O gck2 (sh) (pos)
->*I1 ena2
-> I0 gck (sh) p=gnd
---
FDRSE U5
<- Q q5 (sh) (both)
-> D d5
->*CK gck2 (sh) (pos)
-> R
-> S
---
```

In the `dumpShadows` report, any net that has a shadow has the label "(sh)", and also has exactly one of the following five labels:

p=gnd or p=pwr	The net is always low (or always high, respectively) during the passive phase. A net can also be labeled a=pwr or a=gnd if it has a known value during the active phase. However, active-phase labels are not important in analyzing shadows.

© 2021 All Rights Reserved.

(pos) or (neg)	The net is "positive" or "negative". "Positive" or "negative" means this net will never have a rising (or falling, respectively) edge at the transition from active to passive phase. A net with p=gnd (or p=pwr) is always positive (or negative, respectively) but the (pos) or (neg) label is not shown.
(both)	The net is "both". "Both" means during precompile stage, it is not possible to determine if the net is positive or negative.

A power or ground net appearing in the report has label `pwr` or `gnd`, respectively.

The output of a Q_FDP0 flip-flop clocked by rising edge of a p=gnd or (pos) clock can never change at the passive phase, and thus needs no shadow. Likewise, the output of a Q_FDP0 flip-flop clocked by falling edge of a p=pwr or (neg) clock needs no shadow. Thus, from capacity point of view, clocks with labels p=gnd, p=pwr, (pos), or (neg) are desirable clocks.

The output of a flip-flop clocked by any edge of a clock labeled (both) must have a shadow, which will also be labeled (both). Moreover, any nets downstream from this flip-flop will also have shadows and be labeled (both). Thus, a clock with label (both) is always undesirable.

To understand why a given net has a shadow, use the following method. Find, in the `dumpShadows` report, the gate or storage element driving this net. If the driver is a combinational gate, a shadow on the output must be caused by a shadow on one of the inputs. If the driver is a flip-flop, a shadow on the output is usually caused by a (both) label on the clock input, or by a shadow on the set or reset input. (For a flip-flop, there are other possible causes, such as (sh) (pos) on the clock net connected to the flip-flop with "`*`" indicating inversion.)

The `dumpShadows` report is usually a very large file, but you can use the "vim" text editor to view it. Searching the file in the editor is very slow, so to find the driver of a net NNN it is usually better to use the `grep -n` command in a separate window to get a line number, and then go to that line number in the editor.

```
grep -n '<.* NNN' precompile.dumpShadows
```

A good alternative to viewing the dumpShadows report directly is to use the Schematic Viewer. You can use the script `<install_dir>/share/vxe/gift/misc/shadows.qel` to generate a schematic for the cone of logic generating a given net, with the shadow labels represented as different colors on the schematic. The file `<install_dir>/share/vxe/gift/misc/shadows_howto.txt` gives more information about using `shadows.xel`.

Note: In most situations, it is easier to create a `shadows.request` file, run precompile, and analyze the resulting report.

© 2021 All Rights Reserved.

The precompiler generates the `shadows.request` file automatically with the following text:

```
#####################################################
# File name: shadows.request
# Use this file to analyze the sources of 1X mode logic duplication (shadows).
# Report will be printed to tmp/precompile.shadows_request file.
# ---------------------------------------------------------------------------
# shadows.request should be edited by the user according to the following rules.
#
# A line which starts with # is ignored.
#
# Case 1. shadows.request may contain hierarchical net names, line by line.
# tmp/precompile.shadows_request would contain cones of logic explaining
# why each of these nets has a shadow. A summary report would be printed
# at the bottom of tmp/precompile.shadows_request.
#
# Case 2. Top line in shadows.request may contain the following:
# -random <number_of_shadows>
# xeCompile would randomly pick <number_of_shadows> nets with shadows
# and generate tmp/precompile.shadows_request similar to case 1.
#
# Case 3. Top line in shadows.request may contain the following:
# -logic
# xeCompile would analyze all "shadow generators" in the design.
# tmp/precompile.shadows_request would contain one representative
# for a set of logically equivalent shadow generators,
# with the size of this set.
#
# Case 4. Top line in shadows.request may contain the following:
# -module
# This is similar to case 3, but report in tmp/precompile.shadows_request
# is split by module, i.e. shadow generators are considered equivalent
# if they are logically equivalent as in case 3, and belong to the
# instances of the same module.
#####################################################
```

The most common use of `shadows.request` file are cases 1 and 2:

Case 1: To analyze why specific nets have shadows, create a `shadows.request` file containing the net names, one name per line, and run precompile. Each line must have any space characters that are part of the name, with no extra space characters that are not part of the name.

Case 2: To analyze a random sample of nets with shadows, create a `shadows.request` file containing just the line `-random <N>` where `<N>` is a number such as 100, and run precompile.

At the end of the report, a Summary section lists "shadow generators" that were encountered during tracing, with the number of traces that encountered each shadow generator. A shadow

© 2021 All Rights Reserved.

generator is a gate whose output net is "worse" than any of its input nets, in terms of shadows. Shadow generators may be considered to be the "root cause" of shadows.

Providing Extra Information to Precompile

This section describes methods of providing extra information to precompile to reduce the number of shadow nets in a design. All the methods except `precompile -moreExclusive` can alter design functionality. Use these methods with caution.

Declaring Extra Clock Sources

In some early releases, declaring internal nets as clock sources was a popular method of reducing 1X capacity cost. For example, one design had clock generation logic as in the diagram below:

Originally, the user declared 10 outputs of the PLL block as clock sources. Compiling in 1X mode required almost double the capacity required for 2X mode. However, when the user additionally declared 40 outputs of `CLK_CTRL` as clock sources, the 1X-mode capacity cost was reduced to about 1%.

Each clock source itself has a capacity cost. If the number of internal clock sources is large, it is better to use the `clockAssign` XEL command to connect internal nets directly to original clock sources.

The `shadowsData` XEL command provides a less drastic and more flexible way of providing clock information, so that you need not declare internal nets as clock sources to improve 1X-mode capacity.

© 2021 All Rights Reserved.

Using shadowsData

You can use the `shadowsData` XEL command to specify behavior of particular clock, reset, or other nets in the design. Specifying this behavior can eliminate shadows on logic downstream from these nets.

To specify `shadowsData` for a net, use the command:

```
XEC> shadowsData -add {<net> <keyword>}
```

The following keywords are supported: `p=gnd`, `p=pwr`, `pos`, `neg`, `none`. For simplicity, the description below assumes you have not specified activeEdge negative.

`p=gnd` or `p=pwr`	Forces <*net*> to be low (or high, respectively) during the passive phase, regardless of the value driven by the design at that time. A net declared as `p=gnd` or `p=pwr` in `shadowsData` becomes `p=gnd` or `p=pwr` in the `dumpShadows` report.
`pos` or `neg`	In a given FCLK cycle, if <*net*> is low in the active phase, the `pos` keyword in `shadowsData` forces it to be low in the following passive phase, regardless of the value driven by the design at that time. Similarly, if the <*net*> is high in the active phase, the `neg` keyword forces it to be high in the following passive phase, regardless of the value driven by the design at that time. The `pos` or `neg` keywords affect functionality of the <*net*> only when the <*net*> would have had a rising (or falling, respectively) edge at the passive phase of the primary clock. A net declared as `pos` or `neg` in `shadowsData` becomes (`pos`) or (`neg`) in the `dumpShadows` report because the net cannot have a rising (or falling, respectively) edge at the passive phase.
`none`	Forces the <*net*> to have the same value in the following passive phase as it has in the prrior active phase. If <*net*> would have changed at the passive phase, the change is delayed to the next active phase, or lost entirely if <*net*> would have changed again at that active phase. Declaring a net as `none` removes the shadow on this net in the dumpShadows report.

© 2021 All Rights Reserved.

Declaring a net as `p=gnd` or `pos` in `shadowsData` has a less drastic effect on the functionality than declaring it a clock source. If the net is a gated version of the fastest clock that is low when disabled, declaring the net as `p=gnd` in `shadowsData` allows the behavior to be preserved, whereas declaring it a clock source requires it to toggle always. Declaring an internal clock net as `pos` is even less drastic because it allows this clock to remain constantly high or constantly low when disabled, and allows it to switch between high and low frequencies.

Another advantage of using `shadowsData` over declaring clock sources is that you need not specify frequencies for the given clocks.

Detecting shadowsData Violations

Specifying `shadowsData` for a design net enforces an assumption about the behavior of the net. For example, `shadowsData none` means that the net never changes at the falling edge of the primary clock. If `shadowsData` incorrectly describes the natural behavior of a net, the `shadowsData` will alter the behavior of the net. Such a situation is called a `shadowsData` violation.

If you specify `shadowsData`, precompile automatically creates the following instrumentation nets that you can use to detect `shadowsData` violations:

- The `CVA_UA_VERIFY_SHADOWS_DATA` net is high only in those FCLK cycles in which a `shadowsData` violation occurs. This net (as well as the other instrumentation nets described below) can be used in SDL.

  A typical use is as given below:

  ```
  if (CVA_UA_VERIFY_SHADOWS_DATA == 1) trigger;
  ```

- For each shadowsData net `x.y.z`, precompile creates the following three instrumentation nets at the design top level (names are escaped):

  - `\x.y.z@shadowsDataOriginal` behaves as the original `x.y.z` would have behaved without shadowsData.

  - `\x.y.z@shadowsDataFail` is the individual violation signal for `x.y.z`. It is high in just those FCLK cycles in which `x.y.z` has a violation.

    The global violation signal `CVA_UA_VERIFY_SHADOWS_DATA` is an `OR` of the individual violation signals.

  - `\x.y.z@shadowsDataEnable` is the enable signal for violation on `x.y.z`, which you have to control. Normally, it is `high`. You can force it to `low` to disconnect `\x.y.z@shadowsDataFail` from the `OR` tree that controls `CVA_UA_VERIFY_SHADOWS_DATA`. In this case, `\x.y.z@shadowsDataFail` is

© 2021 All Rights Reserved.

still `high` when `x.y.z` has a violation, but does not cause `CVA_UA_VERIFY_SHADOWS_DATA` to be `high`.

You can add all these nets to the waveforms by using the `probe` command. Ensure that the escaped name of the signal, including the backslash at the beginning and the space at the end, are enclosed in curly braces. For example:

```
probe -add {\x.y.z@shadowsDataOriginal }
```

If you apply shadowsData to a net having name that contains a space character, such as `top.\genblk.abc .clk`", the spaces in that name are replaced by the **!** character in the names of the instrumentation nets, for example as follows:

```
\top.\genblk.abc!.clk@shadowsDataOriginal
```

If you apply shadowsData to a net having multiple equivalent names, the `tmp/*MultiClock` file might contain a message like the following:

```
shadowsData net 'top.u1.gclk' was substituted with its equivalent net 'top.gclk'
```

This message indicates that shadowsData was applied to the `top.u1.gclk` net. But the `top.gclk` name is used instead in the names of the instrumentation nets. One of the names would be `\top.gclk@shadowsDataOriginal`.

Tying Mux-Selects or Other Signals to Reduce Capacity Cost of 1X Mode

In one design, by tracing back the reason for shadows on several randomly chosen flip-flop outputs, it was found that most of the 1X capacity cost was caused by a certain clock which was the output of a mux select. The waveform of this clock changed less than once per FCLK, yet the clock had a shadow, which caused other shadows in the design. The clock was the output of a MUX that selected between a 1X-type clock and a 2X-type clock. After investigation, it was found that in the mode that would be tested in emulation, the MUX-select would always be low, selecting the 2X-type clock. When the MUX-select was tied low using `tieNet`, the 1X-mode capacity cost was reduced from 24% to 4%.

Similarly, you can consider tying any important control signal if its value is known not to change. A common example is a `TEST_MODE` signal.

Using precompile -noReset1X Command to Reduce Capacity Cost of 1X Mode

Normally, if a flip-flop set or reset has a shadow, the output of the flip-flop automatically has a shadow. As a result, if a reset signal of the design can change at the falling edge of the fastest clock, it usually causes a large capacity cost for 1X mode. In such a design, using the `-noReset1X` option of the `precompile` command can reduce the 1X mode capacity cost. When this option is used, the shadow on the set or reset of a flip-flop does not propagate to the output of the flip-flop.

© 2021 All Rights Reserved.

If the output of a flip-flop has a shadow because of its clock as well as because of its set or reset, the `-noReset1X` option has no effect. If the output Q of a flip-flop has a shadow due only to a shadow on set or reset, using the `-noReset1X` option removes the shadow on Q, and has the same effect on functionality as declaring the Q signal as `none` in `shadowsData`. The value of Q in the active phase is unaffected; the value of Q in the passive phase is copied from the active phase, even if the set or reset becomes active during the passive phase.

The effect of the `-noReset1X` option on the functionality of a flip-flop is not as severe as the effect of declaring the set or reset signal as `none` in `shadowsData`. If the design drives set or reset active in a short pulse of just a single passive phase, declaring the set or reset signal as `none` eliminates the pulse entirely. Using the `-noReset1X` option does not eliminate the set or reset pulse, but only causes flip-flop outputs to not show the effect until the next active phase.

The following diagram shows the behavior of a `Q_FDP1` flip-flop (rising-edge triggered, with active low reset) in 1X mode without and with the use of the `-noReset1X` option, assuming D is constant high. The diagram assumes that the `-syncReset` and `-asyncReset` options are not used. The waveform Q1 shows the Q output of the flip-flop if the `-noReset1X` option is not used, and Q2 shows Q if the `-noReset1X` option is used.

At cycle 3, there is a “race condition”:

The `CK` clock has a rising edge at the same time as `reset` R goes inactive. At cycle 3 without the `-noReset1X` option, signal Q1 remains low. (This behavior is required by some designs: if a flip-flop’s clock has an active edge at the same time as reset goes inactive, the flip-flop remains reset for that cycle.) At cycle 3, with the `-noReset1X` option, signal Q2 is low, the same as without the `-noReset1X` option. The only difference is that the signal Q2 goes low one half-cycle later than Q1 does.

© 2021 All Rights Reserved.

By default, the effect of the `-noReset1X` option applies only to flip-flops, not to latches. To obtain the benefit of the `-noReset1X` option on latches, you should use the `-latchResetAsFF` option in addition to the `-noReset1X` option.

Reducing 1X Capacity Cost

To reduce 1X-mode capacity cost, it is helpful to be able to see a waveform for any signal in the design. A good way to accomplish this is to capture, in a 2X-mode full-vision database, a trace file for a few hundred cycles before and after design reset is de-asserted. Then, you can use offline debug to obtain a waveform for any signal in the design. The waveform will not show all possible waveforms for a given signal, but it shows a possible and likely waveform for each signal. (If emulation waveform waveforms are not available, then simulation waveforms can be used instead.) Sometimes you can confirm that a signal will *always* behave as it happens to behave in the waveform. Or, a signal might behave a certain way in the operating modes that will usually be tested in emulation.

Here are situations where 1X-mode capacity cost can be reduced.

1. Shadows are unnecessary for the actual design behavior, but precompile cannot prove it.

   For example, in one design, tracing back the reason for the shadow on a randomly chosen storage element revealed an important design clock (CKA) which was labeled (both) in the dumpShadows report, even though its waveform was the same as that of the fastest design clock. The clock was the output of a two-input MUX.

   The MUX A0 and A1 inputs were flip-flop outputs and did not have shadows. The S input was the fastest design clock (labeled a=pwr p=gnd). Depending on the values of A0 and A1, it is logically possible that CKA could be either the same, or inverted, compared to the primary clock, so precompile must label it (both). However, the clock module designer said that in actual operation, CKA would never be inverted compared to the primary clock, and would never have a rising edge coinciding with the primary clock falling edge. When clock CKA and four other similar clocks were labeled `pos` in shadowsData, the capacity cost for 1X mode was reduced from near 100%, to less than 1%.

2. Shadows are needed for a certain mode of operation, but that mode will not be used during emulation, or will not usually be used.

© 2021 All Rights Reserved.

In one design, tracing back the reason for the shadow on a randomly chosen storage element revealed an important design clock (BUSCLK) which was labeled (both) in the dumpShadows report, even though its waveform showed a slow clock changing only at the rising edge of the primary clock. The signal was the output of a four-input MUX that selected between four clocks (CK, CK/2, CK/3, CK/4) which were divide-by-one, divide-by-two, divide-by-three and divide-by-four of the primary clock. The (both) attribute on BUSCLK came from the divide-by-three clock. The purpose of these four clocks was to allow different frequency ratios between BUSCLK and another clock (CPUCLK). Since the intention was to use the divide-by-three clock in less than 1% of the testing, tieNet was used to tie the divide-by-three clock `low`. This reduced the 1X-mode capacity cost from 50% to 10%. A separate 2X-mode compilation was used for testing with the divide-by-three clock. Alternatively, if the exact location of edges of the divide-by-three clock relative to the primary clock edges was not important, `shadowsData` could have been used to enforce a `none` property on this clock.

3. Shadows are caused by an emulation model, not by the design itself.

In one design, tracing back the reason for the shadow on a randomly chosen storage element revealed an important design clock which was labeled (both) in the dumpShadows report. Its waveform showed a slow clock whose transitions always coincided with the falling edge of the primary clock. However, this slow clock was generated by a clock generation model that had been written for emulation and was not part of the real design being verified. When the model was modified so that the slow clock changed at the rising edge of the primary clock, most of the 1X-mode capacity cost was eliminated.

Reducing 1x-mode capacity cost is an iterative process, as described earlier. Each iteration consists of the following steps:

1. Choose, at random within the dumpShadows report, a storage element (flip-flop or latch) whose output has a shadow. You can make a random choice by going to an arbitrary line in the file, such as line 1000000, and then searching forward for the first FDRSE or LDRSE.

2. Observe the labels on the storage element's inputs, and determine which input(s) caused the shadow on the output. Once you know which input (usually, clock or reset) caused the shadow on the output, there are two approaches you can take:

- ❑ Immediately eliminate the shadow on the storage element output, by applying a shadowsData keyword to the clock or reset, or by one of the other methods described earlier. Usually this will eliminate shadows on other storage elements as well.
- ❑ Use the dumpShadows report and the waveform to trace back the reason the clock has label (both), or reset has a shadow. Sometimes many clocks labeled (both) have a common root cause. If you can find a single root cause, it is better solve the

© 2021 All Rights Reserved.

problem at its root rather than separately solving many resultant problems. If a clock net labeled (both) is driven by a gate with more than one input labeled (both), it is usually better to trace back an input toggling in the waveform, rather than one that is flat. Tracing a toggling input is likely to lead more directly to the single root cause, if there is one.

You can use the following tactics to remove the shadow on a storage element output.

a. If the shadow on the storage element's output is caused by a shadow on its reset signal, apply keyword `none` to the reset signal in shadowsData, or use `precompile -noReset1X` option.

b. If the shadow on the storage element output is caused by a (both) label on its clock, but in the waveform, the clock is a low frequency clock which changes only on the active edge of the fastest clock, apply keyword `none` (or `pos`) to the clock in shadowsData. The `none` keyword has a stronger effect in removing shadows, but is more likely to affect design functionality.

c. If the shadow on a storage element output is caused by a (both) label on the clock, but in the waveform, in every FCLK cycle the clock is either the same as the fastest clock, or flat low, apply keyword `p=gnd` (or `pos`) to the clock in shadowsData. The `p=gnd` keyword has a stronger effect in removing shadows, but is more likely to affect design functionality.

d. If the shadow on the storage element output is caused by a (both) label on the clock, but in the waveform, the clock never has a rising edge that coincides with the falling edge of the fastest clock, apply keyword `pos` to the clock in shadowsData. If the clock never has a falling edge that coincides with the falling edge of the fastest clock apply keyword `neg` to the clock in shadowsData.

e. If a clock has both rising and falling edges that coincide with the falling edge of the primary clock, then the (both) label in the dumpShadows report accurately describes the clock's behavior, and applying any keyword in shadowsData will be inconsistent with the behavior in the waveform. In this case, try to determine the reason for the observed behavior, and determine whether the design can be modified to change the behavior.

Tactics b, c, and d are for cases where a clock is labeled (both) in dumpShadows, but the waveform shows a more benign behavior. Tactics a,b,c and d all use shadowsData to enforce a desired behavior. If possible, before applying any of tactics a-d for a given signal, ask the designer whether there are modes or conditions where the signal's behavior is expected to be different from the desired behavior.

3. After making the changes, recompile the design and verify that the chosen storage element no longer appears in the dumpShadows report.

© 2021 All Rights Reserved.

Additional Information on 1X Mode

This section provides extra information about 1X mode:

- Using a Dummy Fastest Clock Source
- Implementing Quarter-Cycle Phase-Shifted Clocks
- Active- and Passive-Phase Values in SDL

Using a Dummy Fastest Clock Source

Sometimes it is helpful to define a dummy clock source which is faster than the fastest real clock source in the design. For example, in a design having two fastest clocks with relative frequencies 5 and 3, if the clock with frequency 3 generated too many shadows in 1X mode, define a dummy clock source with frequency 6. Then, the clock with frequency 3 becomes a 2X-type clock, and the clock with frequency 5 still runs almost as fast as it would have without the dummy clock.

Note: In 1X mode, the compiler optimizes the capacity because the fastest CAKE clock source is the same as the FCLK of the emulator. Therefore, in some designs, defining a dummy clock source could increase the 1X mode capacity cost.

To specify a net as a dummy clock source, the net must exist in the design. If you do not want to modify the design netlist, use the `iScopedNets` command to add an unconnected net to a particular module in the design. For example, the following command adds an unconnected net named `my_dummy_clock` to cell TOP in library `LIB`:

```
iScopedNets -VERSION3 LIB TOP - my_dummy_clock . h
```

For more information about the `iScopeNets` command, refer to the Connecting Nets Using iScopedNets Program section in the *Compiling Designs for In-Circuit Emulation with xeCompile* chapter of *VXE User Guide*.

Implementing Quarter-Cycle Phase-Shifted Clocks

Occasionally, a design might require a second clock source which is the same as the fastest clock source, but delayed by one quarter of a cycle. Suppose the names of these first and second clock sources are CLKA and CLKB respectively. To implement such a design in 2X mode, the period of CLKA must be four FCLK cycles, resulting in a lower-than-desired emulation speed. You can implement such a design in 1X mode by defining a dummy fastest CAKE clock source CLKDUMMY, and then using `clockAssign` commands to specify the behavior of CLKA and CLKB relative to that of CLKDUMMY:

```
clockOption -add {technology CAKE 1}
clockFrequency -add {CLKDUMMY 100}
```


© 2021 All Rights Reserved.

```
clockAssign -add {CLKA CLKDUMMY 2}
clockAssign -add {CLKB ~ CLKDUMMY 2}
```

These `clockAssign` commands specify that signals CLKA and CLKB each have frequency half that of CLKDUMMY and that CLKA and CLKB change on the rising and falling edges, respectively, of CLKDUMMY. Outputs of all flip-flops clocked by CLKB will have shadows, so this approach will have a significant capacity penalty unless CLKB clocks only a small portion of the design. If CLKB clocks more of the design than CLKA, swap the names CLKA and CLKB in the above `clockAssign` commands.

You can add the dummy clock CLKDUMMY to the design using `iScopedNets`, as described in the previous section.

Active- and Passive-Phase Values in SDL

SDL samples signals once per FCLK. In 1X mode, if a net can have different values in the active and passive phases of the primary clock, then normally, SDL uses the passive-phase value.

However, if you declare `shadowsData` for a net, precompile provides special visibility; you can use SDL to refer to that net's active- and passive-phase values.

For the net whose hierarchical path is `u1.u2.net3`, the special names `\u1.u2.net3@QTactive` and `\u1.u2.net3@QTpassive` refer to the active- and passive-phase values of the net. Each special name includes a space at the end. The regular name `u1.u2.net3` also refers to the passive-phase value. If one or more components of the hierarchical path are escaped names, the special name is constructed by changing the space delimiting each escaped component to a `!` character, and then constructing the name as usual.

For example, the following net:

```
\u/1 .\u/2 .\net/3
```

has the following active-phase name:

```
\\u/1!.\u/2!.\net/3!@QTactive
```

You can also use the `value` command with a special name:

```
value {\u1.u2.net3@QTactive }
```

Specifying a `shadowsData` keyword such as `pos` or `none` for a net can affect the net's behavior. To enable special visibility without affecting the net's behavior, use the `both` keyword.

For example:

© 2021 All Rights Reserved.

```
shadowsData -add {u1.u2.net3 both}
shadowsData -add {\u/1 .\u/2 .\net/3  both}
```

Using the CAKE 0.5X Mode

CAKE 0.5X mode is a compilation mode that allows the fastest design clock to run at double the frequency of FCLK. Usually, for this mode to be useful, the design's highest-frequency clock must clock only a small fraction of the design, and the design must use only one edge of the highest-frequency clock.

To compile in 0.5X mode, give the following command before precompile:

```
XEC> clockOption -add {technology CAKE 0.5}
```

Precompile only allows 0.5X mode if the following restrictions are met:

- All changes in the values of storage elements (latches, flip-flops, and memories) must occur at the active edge (that is, the rising edge, unless `activeEdge negative` is specified) of the fastest design clock.
- If a storage element needs to be updated depending on the value of one of its inputs in the passive phase of the fastest clock, and this input can have different values in the active and passive phases, then the value in the passive phase must be a constant known at compile time. This rule precludes certain types of gating for the fastest clock in 0.5X mode. Usually, the fastest clock can be gated by a sequence of AND gates, but if it is gated by an AND followed by an OR gate, then precompile will not allow 0.5X mode.
- If a design output can change at the passive phase of the fastest clock, you must specify

  ```
  XEC> clockOption -add {ignoreOutputs on}
  ```

  As a result, the output will reflect only the value the signal would have in the active phase.

When a design satisfies these restrictions, precompile can implement it in 0.5X mode using no more than net and shadow for each user net (that is, it is not necessary to implement four different nets for each user net). In CAKE 0.5X mode, net and shadow represent a user signal's value in the active phase of even- and odd-numbered cycles of the fastest clock source.

In 0.5X mode, a clock with frequency greater than half that of the fastest clock will run faster than FCLK, and is called a 0.5X-type clock. A flip-flop clocked by a 0.5X-type clock must always have a shadow, even though it is clocked by the rising edge of the clock. So, 0.5X mode will always have a significant capacity cost unless 0.5X-type clocks clock only a small fraction of the design.

At run time, the design is implemented in 1X mode. Waveforms show only two time points per FCLK cycle; values of design signals during the passive phase of the fastest clock are not

© 2021 All Rights Reserved.

shown. The values in the passive phase of the fastest clock are not shown. So, the waveform of the fastest clock always appears flat.

The following diagram shows the behavior of clocks in 0.5X mode for a design with five CAKE clocks having relative frequencies 12, 10, 6, 5, and 3. Logic Analyzer waveforms show only two time points per FCLK cycle; the values of signals during the shaded time periods do not appear in the waveform.

Target I/O (except multipulse outputs defined with `terminalTiming -edgeInfo`) can change at most once per cycle of the fastest clock (that is, twice per FCLK). To allow target I/O to change twice per FCLK, you must enable the multisampled I/O feature, and specify as a multiSampledTerminal any input that should be sampled twice per FCLK.

To connect the fastest CAKE clock to the target, use a `terminalTiming -edgeInfo` command similar to the following, with two rising edges and two falling edges per FCLK cycle:

```
terminalTiming -add {sb_if_clk_out -edgeInfo 0 96 0 96 240 336}
```

After compiling, verify in the `dbFiles/<top>.ttime` file that all non-shadow signals are scheduled earlier than the second rising edge of the multipulse clock. For the above `terminalTiming -edgeInfo` example, the second rising edge is at step `240`. The `<top>.ttime` file uses units of TIFs, and one TIF is `8` steps. Thus, for this example, user needs to verify that each non-shadow signal is scheduled earlier than TIF `30`. If any signal is TIF 30 or later, adjust the step numbers in the `-edgeInfo` command if necessary, and recompile.

Except for the 0.5X-type clock implementation, the I/O timing constraints for 0.5X mode are normally the same as for 1X mode.

© 2021 All Rights Reserved.

Changing CAKE Clock Frequencies from DUT

Using the following command you can ensure DUT access to CAKE clock frequencies is activated:

```
precompileOption -add enableDUTCAKE
```

This feature is enabled by default. To disable it, use the following command:

```
precompileOption -add disableDUTCAKE
```

CAKE clock frequency is a 32-bit bus interpreted as signed integer in KHz as opposed to XEL `clockconfig` command which interprets frequencies as floating point numbers in MHz.

Clock frequency names are constructed by appending `$frequency` suffix to CAKE clock name. For example, if hierarchical name of a CAKE clock is `top.inst1.clk1`, then the corresponding frequency should be accessed as OOMR `top.inst1.clk1$frequency[31:0]`. It is important to append `[31:0]` part-select specification because width of OOMRs is unknown to HDL-ICE Compiler, and without this range a 1-bit scalar value will be synthesized.

Naming convention described above must produce a valid Verilog name, therefore CAKE clock must be a scalar, not a bit of a vector, nor a struct member.

Note that HDL-ICE Compiler fails if OOMR target module is visible, but the name (`<clock>$frequency`) is missing.

Clock name can also be escaped. Example:

```
reg \my%favor!t&clock? ;
```

Frequency control bus would be named this way:

```
\my%favor!t&clock?$frequency [31:0];
```

Frequency control bus has no Verilog type (reg/wire), therefore it may be assigned in an arbitrary way in Verilog syntax: in a sequential block, in level sensitive block or through continuous assignment. Value for `$frequency` bus may be in any format: a constant, a variable or any Verilog expression evaluating to 32 bit signed integer. Verilog source files that control `$frequency` buses must be processed by HDL-ICE Compiler into a library, which is separate from the libraries that process files with the corresponding CAKE clock declarations.

Multiple drivers allowed must be tri-stated. Syntactically, frequency value may be any 32-bit integer value, but instrumentation keeps the frequency within limits as described in `clockConfig` section of *VXE Command Reference Manual*. Frequencies are automatically increased or decreased by the instrumentation to stay within the allowed limits. Since this adjustment occurs within the hardware model, there are no warnings for the user.

© 2021 All Rights Reserved.

Inserting Delays between Cross-Domain Synchronizers

Cross-domain synchronizers can be verified in designs having Clock Domain Crossing (CDC). During the precompile process, delays are automatically inserted between the cross-domain synchronizers according to the specifications that you provide. These delays can be dynamically switched on and off at run time without interrupting the emulation. To provide the cross-domain synchronization:

- Describe the cross-domain synchronizers in a separate CDC file (that is, `*.cdc` file) and specify this filename using the following `precompileOption` command option:

  ```
  precompileOption -add {cdc <CDC_file_name>}
  ```

  Refer to Clock Domain Crossing (CDC) Format of Appendix 4, “Files and File Formats” for details about the format of the CDC file.

- Describe the dynamic control for the delays in a special module called `cdc_control`. Create this module and instantiate only once in the design hierarchy. The location in the design hierarchy does not matter. For an example of the `cdc_control` module, refer to Example of Inserting Delays between Cross-Domain Synchronizers.

  For the controls that you specify in the CDC file, provide the driving logic in the `cdc_control` module by using registers or wires. Note that for a sourceless control, the delay will still be instrumented but it will not be activated unless it is forced to `1` at debug time.

  Precompile automatically locates the `cdc_control` module instance in the design hierarchy. If such an instance is missing or there are multiple instances in the hierarchy, an error is generated.

© 2021 All Rights Reserved.

If you set the CDC version to `0` in the CDC file, the instrumentation inserted by precompile is as per the schematics given below. The *Control* in this schematics is connected to the corresponding register or wire in the `cdc_control` module, as specified in the CDC file.

Precompile inserts additional internal flip-flop that drives Delay.

Output mux controlled by Control selects one or two clock delays from DIN to DOUT.

However, if you set the CDC version to `1` in the CDC file, the instrumentation inserted by precompile (shown in the schematics below) is different in the following two ways:

- Delay is controlled by the virtual clock `Virt_clk` instead of the design clock `CLK`. (Virtual clock is driven in the `cdc_conrol` module).
- Delay is placed at *Synchronizer* `D` input instead of `Q` output.

Precompile inserts additional internal flip-flop that drives Delay with virtual clock.

Mux controlled by Control selects a delay for DIN going into D input of the Synchronizer.

© 2021
All Rights Reserved.

Example of Inserting Delays between Cross-Domain Synchronizers

This section discusses the following two examples:

- Example of writing and then instantiating the cdc_control module in a DUT file.
- Example CDC file named Example CDC File: 1.cdc for CDC version `0`.

  Use the following command to invoke the `1.cdc` file:

  `precompileOption -add {cdc 1.cdc}`

cdc_control module

```
module cdc_control(clk, cdc);
input clk, cdc;
wire sync0_sel, sync1_sel; // controls
for delaying synchronizers
reg[2:0] cnt; // just a counter
always@(negedge clk) cnt = cnt+1;
```

Module with predefined name `cdc_control` you must provide and instantiate once in the hierarchy

```
assign sync0_sel = (cnt >= 1 && cnt <=
3) && cdc;
assign sync1_sel = (cnt >= 3 && cnt <=
6) && cdc;
endmodule
```

Delays are ON periodically if cdc is asserted

```
module sync(clk, d, q);
input clk;
output[7:0] d, q;
reg[7:0] d, q;
reg[7:0] sync; // sync is a
synchronizer between d and q

always@(posedge clk) begin
d = d + 1;
  sync <= d;
  q <= sync;

end
endmodule
```

A module with a synchronizer

© 2021 All Rights Reserved.

```
module sync_and_control(clk, cdc, d,
q);
input clk, cdc;
output[7:0] d, q;
reg[7:0] d, q;
reg[7:0] \!sync ; // \!sync is a
synchronizer between d and q

always@(posedge clk) begin
  d = d + 1;
  \!sync <= d;
  q <= \!sync ;
end
```

A module with a synchronizer and `cdc_control` instance

```
cdc_control CTRL(clk, cdc);
endmodule
```

One and only one instance of module `cdc_control`

```
module dut(clk, cdc, d0, d1, q0, q1);
input clk, cdc;
output[7:0] d0, d1, q0, q1;

sync SYNC (clk, d0, q0);
// testing escaped instance name
sync_and_control \SYNC&CTRL (clk, cdc,
d1, q1);

endmodule
```

Top of DUT

Example CDC File: 1.cdc

Controls for synchronizers in module `sync`:

```
crossing_00 dut.SYNC.sync[0] clk
sync0_sel
```

Path to synchronizer from top, with no path to clock

```
crossing_01 dut.SYNC.sync[1] SYNC.clk
sync0_sel
```

Path to synchronizer from top, with path to clock

```
crossing_02 SYNC.sync[2] clk sync0_sel
crossing_03 SYNC.sync[3] dut.SYNC.clk
sync0_sel
```

Path to synchronizer a level below top

© 2021 All Rights Reserved.

```
#
crossing_04 dut.SYNC.sync_reg[4] clk
sync0_sel
crossing_05 dut.SYNC.sync_reg[5]
SYNC.clk sync0_sel
crossing_06 SYNC.sync_reg[6] clk
sync0_sel
crossing_07 SYNC.sync_reg[7]
dut.SYNC.clk sync0_sel
```

Same as above with `_reg` suffix to sychronizer names

Controls for synchronizers in module `sync_and_control`:

```
crossing_10 dut.\SYNC&CTRL .\!sync [0]
clk sync1_sel
```

Path to synchronizer from top, with no path to clock

```
crossing_11 dut.\SYNC&CTRL .\!sync [1]
\SYNC&CTRL .clk sync1_sel
```

Path to synchronizer from top, with path to clock

```
crossing_12 \SYNC&CTRL .\!sync [2] clk
sync1_sel
crossing_13 \SYNC&CTRL .\!sync [3]
dut.\SYNC&CTRL .clk sync1_sel
```

Path to synchronizer a level below top

```
crossing_14 dut.\SYNC&CTRL .\!sync_reg
[4] clk sync1_sel
crossing_15 dut.\SYNC&CTRL .\!sync_reg
[5] \SYNC&CTRL .clk sync1_sel
crossing_16 \SYNC&CTRL .\!sync_reg [6]
clk sync1_sel
crossing_17 \SYNC&CTRL .\!sync_reg [7]
dut.\SYNC&CTRL .clk sync1_sel
```

Same as above with `_reg` suffix to sychronizer names

© 2021 All Rights Reserved.

Connecting Synchronizer Asynchronous Reset with Delay Asynchronous Reset

You can connect asynchronous reset signals of the synchronizer FFs with that of the delay FFs. The asynchronous reset signal is a net connected with the synchronizer similar to the clock.

Precompile inserts additional internal flip-flop that drives Delay.

Output mux controlled by Control selects one or two clock delays from DIN to DOUT.

To enable this connectivity, use the `precompileOption -add useCDCreset` command in your compile script. This option can be used for both CDC versions: 0 and 1.

Provide the synchronizer logic in the DUT file, like the example shown below:

```
always @(posedge clk or negedge reset)          //Clock domain crossing
begin
      if(!reset)
      begin
            cdc_temp <= 0;
      end
      else
      begin
            cdc_temp <= in_1;
      end
end
always @(posedge clk)
      sync_out <= cdc_temp
```

At run time, when you force the reset to the synchronizer (and if the design is compiled with the `precompileOption -add useCDCreset` command in the compile script), then the both synchronizer FFs and delay FFs will be reset together.

© 2021
All Rights Reserved.

© 2021 All Rights Reserved.

12

Loop-Breaking

This appendix describes the compiler's loop-breaking process, the problems that might occur, and how you can solve the problems or avoid them.

Important

This chapter is applicable to the ICE and SA modes. Refer to Loop-breaking in SA Mode for details about the SA mode. The sections prior to the SA-mode section pertain to the ICE mode.

Definition of Loop-Breaking

The VXE back-end compiler (invoked by the `compile` XEL command) does not process combinational loops in its input netlist. Therefore, precompile checks the design for combinational loops, and breaks any loop it finds by inserting a delay at a point within the loop. The delay is usually 1 FCLK cycle, but can be a *half* FCLK cycle as explained below.

A path from flip-flop clock, reset or set to Q, or latch D, enable, set or reset to Q is considered combinational.

Problems Caused by Loop-Breaking

Precompile's loop-breaking is done automatically, and usually you need not be concerned with it. However, there are two situations in which loop-breaking delay could cause a problem.

1. Loop-breaking delay could require extra oversampling.

   Normally, for correct function, signal transitions on a path from a flip-flop Q output to another flip-flop D input must propagate within one clock cycle. Occasionally, transitions must propagate within a half clock cycle. Such paths are called single-cycle and half-cycle paths, respectively. If a clock cycle is 2N FCLKs long, then 2N inserted 1-FCLK delays on a single-cycle path, or N delays on a half-cycle path, will cause the design not to work. In such a case, you can make the design work by increasing the *oversampling*

© 2021 All Rights Reserved.

ratio (the number of FCLKs per cycle of the fastest user clock). For example, when clocks are generated using CAKE technology, to increase the oversampling ratio from 2 to 4, you would specify

```
clockOption -add {technology CAKE 4}
```

to compile in 4X mode, instead of specifying

```
clockOption -add {technology CAKE 2}
```

to compile in 2X mode. Alternatively, without recompiling, use the command

```
XE> clockConfig -sample 4
```

at run-time to increase the oversampling ratio to 4.

2. Loop-breaking could cause clock skew.

   A delay on a path from a clock source to a flip-flop or latch clock pin can cause clock skew, and cause the design not to work. A failure caused by clock skew cannot be solved by increasing the oversampling ratio. Fortunately, precompile uses heuristics that in most designs, ensure that loop-breaking delay will not be added on a clock path unless there is no other way to break a loop.

Proactive Steps to Avoid Loop-Breaking Problems

You can take the following steps to prevent loop-breaking problems:

- Steps to Break Loops and Long Paths
- Miscellaneous Steps for Clock Delays

Steps to Break Loops and Long Paths

Use the following methods to eliminate or break long combinational paths in a design. These actions reduce the need for loop-breaking and improve the compilation step count, resulting in faster emulation speed.

- Tie the scan or test-mode control signals, if possible. Some designs have scan or test logic that does not need to be tested in emulation. In such designs, tying a scan or test-mode control signal to the non-test value can enable simplification of the design logic, eliminating multiple loops and long paths.
- If the design uses flip-flop library models that implement a flip-flop by using master-slave pairs of latches, remodel these flip-flops, if possible, by using flip-flops from the QTREF library.

© 2021 All Rights Reserved.

- If parts of the design are latch-based, and use non-overlapping clocks, use the `noOverlapClock` User Data command to specify information about the non-overlapping clocks.
- By default, when a flip-flop set or reset input goes active, an immediate effect is seen on the flip-flop output. Use the `precompile -syncReset` option to delay the effect of set or reset going active by one FCLK. This effectively eliminates the combinational path from the flip-flop set or reset to the output. Typically, designs work correctly with the `precompile -syncReset` option, but might have fewer loops, and fewer long paths, thereby, causing a poor step count.
- Use the precompile options, especially the `-dontMinimizeDataBreaks` and `-syncData` options to add extra breaks on the latch D pins. These options reduce the need for loop-breaking, but the extra breaks often cause the design to need extra oversampling. So, it is possible that, overall, using these options does not provide any benefit.

Miscellaneous Steps for Clock Delays

At times, the `xe.msg` file contains a warning like the following:

```
WARNING (multclk-1121): Maximum analysis depth 128 exceeded.
```

This means that the precompiler judged that the loop-breaking process was taking too long, and abandoning its usual analysis, it used, instead, a simpler algorithm that tends to insert more loop-breaks than are really needed. The simpler algorithm might require extra oversampling, or it might delay clocks, causing incorrect functionality regardless of oversampling. To get the best loop-breaking results, you should use the `precompile -loopsDepth <n>` option, where `<n>` is a value large enough to eliminate the warning. (Usually, 1024 is large enough.) The drawback of using this option is that precompilation might take longer.

You can use the `precompile -recognizeClockDivider` option to reduce the possibility that loopbreaking will delay a clock in a design, which contains clock dividers. This option is not enabled by default, but its use is recommended. You can also try the `precompile -extraClockSources` option to avoid clock delays.

What to Do if a Loop-Breaking Problem is Suspected

If a loop-breaking problem is suspected, first try increasing the CAKE oversampling. If you compiled in 2X mode, you can increase to CAKE 4 or 8 at run time using the `clockConfig` command, without recompiling. If increasing the oversampling solves the problem, it is likely a loop-breaking problem, although it could be a problem caused by delays other than loop-

© 2021 All Rights Reserved.

breaking. Refer to Avoiding the Need for Oversampling for suggestions about how to solve such problems without increasing oversampling.

Vector Debug does not support CAKE clock generation. However, you can increase oversampling in Vector Debug by doubling the number of vectors for each clock cycle. You can use the script `<install_dir>/share/vxe/gift/misc/double_cbc` to double the vectors in a cbc file. Comments in the script give more information.

If increasing the oversampling does not solve the problem, it still could be a loop-breaking problem in which loop-breaking delayed a clock. (Problems caused by clock delay are generally not solved by increasing the oversampling.) Refer to Solving Clock-Delay Problems Caused by Loop-Breaking for suggestions about how to solve clock delay problems.

Any time a loop-breaking problem is suspected, if `xe.msg` contains the following message, you should try to eliminate it using the `-loopsDepth` option:

```
WARNING (multclk-1121): Maximum analysis depth 128 exceeded.
```

If possible, apply the other proactive steps to avoid loop-breaking problems, mentioned earlier.

Delays in Palladium Z1

If a design works only with extra oversampling (for example, it works in 4X mode but not 2X mode, or it works in 2X mode but not 1X mode), it is because increasing the oversampling increases the periods of the design clocks relative to the following fixed delays:

- Internal delays
- Target I/O delays
- Non-target I/O delays

Internal delays can be added automatically by precompile's loop-breaking, or these can be added by the `breakPin` or `breakNet` userdata commands, or by delay cells (`Q_FDP0B` cells) instantiated in the netlist.

Target I/O delays occur when the transition of a design primary output (for example, a clock output to the target) causes a transition of a design primary input (for example, an input from the target). Unless you use `delayBox` and `terminalTiming` commands to constrain the input to be scheduled at a later step than the output, there will normally be a one-FCLK delay from the output to the input. Also, for a bidirectional primary I/O signal connected to a target, unless you constrain the input component of the bidirectional signal to be scheduled at a later step than the output and enable components, there will normally be a one-FCLK delay from output and enable components to the input component. In the *VXE Command Reference*

© 2021 All Rights Reserved.

Manual, section on the `terminalTiming` command explains how to use `delayBox` and `terminalTiming` commands to avoid target I/O delays.

If a bidirectional top-level I/O signal of the design is not connected to an external target using terminalAssign, precompile adds a one-FCLK delay on the load pins of the top-level signal, unless the compiler option `delayBidirIO OFF` is specified. This delay is called a *non-target I/O delay*.

Avoiding the Need for Oversampling

If internal delays (especially loop-breaking delays) cause extra oversampling to be needed, use one of the following methods to avoid the need for oversampling:

1. Debug the problem, and direct precompile to break the loop differently

   Use Full Vision to debug the design without extra oversampling, and isolate the failure to a particular delay. Then, use the *.brk file (described below) to determine whether the delay is a loop-breaking delay. If it is a loop-breaking delay, you can direct precompile to break the loop differently, using one of the methods described in Preventing a Particular Break.

   If you think precompile could have broken the loop in a more intelligent way, please provide the details to a Palladium Z1 Customer Support engineer. Feedback from you is the best guide to help Cadence improve the loop-breaking algorithm.

2. Use `precompile -breakHalfCycleAll`

   The `precompile -breakHalfCycleAll` option makes all loop-breaking and user-added delays *one-half FCLK*. (Palladium Z1 hardware cannot directly implement a half-FCLK delay. Therefore, half-FCLK delays are implemented using full-FCLK delays and 1X-mode shadow nets.) A half-FCLK delay will break a loop as effectively as a 1-FCLK delay, but the delay is only half as long compared to the design clock cycle, so it is less likely than a full-FCLK delay to make the design fail.

   The drawback of the `precompile -breakHalfCycleAll` option is that in some designs, it causes a large capacity penalty.

   `precompile -breakHalfCycleAll` was originally intended for use in CAKE 1X mode. However, the option is supported in 2X mode as well. If you use `-breakHalfCycleAll` with 2X mode, the waveform shows two samples per FCLK, as in 1X mode. However, the fastest clock source has two FCLKs per design clock cycle, as in normal 2X mode.

© 2021 All Rights Reserved.

If your design works in 4X mode but internal delays are causing the design not to work in 2X mode, you can try `-breakHalfCycleAll`. If the capacity cost is not too much, the compile will pass, and the design will now work correctly.

Preventing a Particular Break

If precompile breaks a loop in an undesired way, and you know the location of the undesired break, there are several ways to direct precompile to break the loop at a different location:

- Specify a zero-frequency clock source.

  If you are using CAKE or SCM technology for clock generation, you can specify nets as clock sources with frequency 0. When you specify a net as a clock source with frequency 0, precompile considers that net as a clock source for the purpose of loop-breaking, but leaves the design logic driving that net intact and does not insert clock generation logic to drive the net. In general, precompile marks clock sources, and all nets that can be reached from clock sources by tracing forward through combinational gates, as `(clk)` nets. When breaking loops, precompile tries never to delay a net marked as `(clk)`.

- Use the `noBreakPath` command.

  The `noBreakpath -add {<startnet> <endnet>}` command directs loobreaking not to insert any break on a path from `<startnet>` to `<endnet>`. For details, see the *VXE Command Reference Manual*.

- Use the `preferBreakInstance` command.

  You can use `preferBreakInstance -add` <*`instance_name`*> to specify that loopbreaks in a certain part of the design are preferred over loopbreaks in other parts of the design.

- Specify a NOBREAKS property.

  You can specify a NOBREAKS property for a module or instance in the gate level netlist. When you specify a NOBREAKS property for a module or instance, precompile will not insert breaks at the input pins of leaf-level cells within that module or instance. Breaks will be inserted instead at the input pins of leaf-level cells elsewhere in the design. To specify a NOBREAKS property for a module, insert the line

  ```
  // pragma CVAINTPROP MODULE <module_name> NOBREAKS 1
  ```

  anywhere between module and endmodule of the given module's definition. For example, in module `mmm` you can insert the line

  ```
  // pragma CVAINTPROP MODULE mmm NOBREAKS 1
  ```

  To specify a NOBREAKS property for an instance, insert the line

© 2021 All Rights Reserved.

```
// pragma CVAINTPROP INSTANCE <instance_name> NOBREAKS 1
```

in the module that instantiates the given instance. Ensure that the NOBREAKS property for an instance is inserted below the line that has the instantiation definition. In the following example,

```
Q_XOR2 U57 ( .A0(n8), .A1(n11), .Z(n12));
//pragma CVAINTPROP INSTANCE U57 NOBREAKS 1
```

precompile will not insert breaks at the input pins of the `Q_XOR2` gate U57.

- Manually break the loop at a different point.

  You can get a listing of the loop broken by the undesired break from the `*.loops` report (described later in this Appendix), and use `breakNet` or `breakPin` commands (or instantiate a `Q_FDP0B` delay cell in the netlist) to break the given loop. A drawback of this approach is that often, a given break can break several different loops. Sometimes when you manually break a loop to avoid a particular break, precompile still inserts the undesired break, because it also breaks a different loop not broken by the manual break. In such a situation, you need to insert multiple breaks, consulting the `*.brk` and `*.loops` files after each precompile run, until precompile no longer inserts the undesired break.

How precompile identifies (clk) nets

A *clock path* is a path taken by a clock transition from a clock source, through logic such as muxes, gates, and clock dividers, to the clock pin of a flip-flop or latch. If loopbreaking inserts delay on a clock path, the delay is likely to introduce skew between different clocks, causing the design to function incorrectly. Precompile attempts to identify likely clock paths by using an algorithm that labels certain nets as `(clk)` nets. Precompile inserts loop-breaking delay at a fanout of a `(clk)` net only if there is no other way to break a given loop.

Precompile uses a combination of *forward-tracing* and *backward-tracing* algorithms to identify nets to be labeled as `(clk)`. In the `PDB/*.loops` report, a (`clk`) net is labeled (`clk ftrace`) or (`clk btrace`), depending on whether it was identified by forward or backward tracing.

The forward-tracing algorithm starts from clock sources, and traces forward through combinational gates, labeling all nets reached as `(clk)`. A limitation of the forward-tracing algorithm is that when a clock path goes through a flip-flop (such as a clock divider) or latch, the tracing algorithm will stop at the flip-flop or latch, and not label the flip-flop or latch output and nets downstream from it as `(clk)`. If you know that the output net of a certain flip-flop or latch is likely to be part of a clock path, use the `clockFrequency` command to specify the net as a *zero-frequency clock source*. Precompile does not insert CAKE clock generation

© 2021 All Rights Reserved.

logic for zero-frequency clock sources, but does treat them as clock sources for the forward-tracing algorithm.

Two options, `-recognizeClockDivider` and `-moreClockSources`, affect forward tracing. The precompile `-recognizeClockDivider` option enables recognition of certain simple clock dividers. The forward-tracing algorithm traces from clock to Q of a recognized clock divider as though it were a combinational gate. The `-moreClockSources` option causes any net connected directly, or through buffers and inverters, to a flip-flop or latch clock pin to be treated as a clock source by the forward-tracing algorithm.

During loopbreaking, if a loop contains no `(clk)` net, then precompile uses an additional backward-tracing algorithm to label `(clk)` nets: it traces backward from flip-flop and latch clock and data pins, marking each net reached as pure clock (not reached by tracing back from any data pin), pure data (not reached by tracing back from any clock pin) or both clock and data (reached by tracing back from both clock and data pins). Pure clock nets in the given loop are labeled `(clk)`.

How Precompile Avoids Delaying Clocks

Precompile's loop-breaking uses the following heuristics that attempt to avoid delaying clocks. The heuristics are listed here in order of decreasing priority.

- When possible, insert delay on an input pin of a combinational gate (not on a flip-flop clock input) which has another input coming from outside the loop.
- Precompile identifies `(clk)` nets as described earlier. If all nets in a loop are labeled `(clk)`, then precompile must insert a break at one of these nets. It prefers to break at an input of a gate which has another input which is a labeled `(clk)`, but which comes from outside the loop.
- When precompile can break a loop by at a non-`(clk)` input pin of a gate whose output is labeled `(clk)`, it prefers this break compared to an input of a gate whose output is not labeled `(clk)`. (This heuristic is not applied to a certain type of loop called a set/reset loop.)
- When a loop contains a flip-flop or latch followed by a chain of gates, insert delay on an input pin of a gate as far back in the chain as possible.

Solving Clock-Delay Problems Caused by Loop-Breaking

If you know, or suspect, that loop-breaking has delayed a clock, try using the precompile options `-recognizeClockDivider` and `-moreClockSources`. Try `-recognizeClockDivider` first, and then if necessary, try the combination of both options.

© 2021 All Rights Reserved.

Clock delay problems cannot be solved by increasing the CAKE oversampling, but they can be solved using the `-safeLoops` precompile option. This option corrects the functional problems caused by loop-breaking, including clock delay problems, at the expense of emulation speed. For more information, refer to the `precompileOption` command description in the *User Data Command* chapter of *VXE Command Reference Manual*.

The `-safeLoops` option has the drawback of slower emulation speed. Alternatively, use the `precompile -verify` option (described in Detecting Clock Delay) to perform the following:

- Quickly determine whether a flip-flop or latch clock transition is ever delayed.
- Find the delayed clock.
- Find the loop-break that caused the delay.

If you know from the waveform that a certain clock is delayed, then you should use Full Vision waveforms to trace the delayed clock transition back through the design and find the loop-break which caused the delay. When tracing back, you will come to a gate whose output transition is delayed, but the corresponding transition of one of its inputs is not delayed. The loop-break is on the input pin of this gate. This can be confirmed by finding the gate in the `PDB/*.brk` file. The `PDB/*.brk` and `PDB/*.loops` files are described in sections The *.brk File and The *.loops File, respectively.

In the `PDB/*.loops` file, find the loop which has the break (already found in `PDB/*.brk`) at the begining of the loop. (If you cannot find such a loop, specify `precompileOption -add {logAllLoops N}`, where `N` is a number larger than the total number of loop-breaks added by loop-breaking, and re-run precompile.) There are now two possible cases, depending on whether or not there is a `(clk)` annotation, that is, `(clk btrace)` or `(clk ftrace)`, at the end of the line in the `PDB/*.loops` file showing the break.

- If there is no `(clk)` annotation, then precompile did not know that the net where it inserted a break is a clock. Use Full Vision waveforms to trace back the clock transition from net `N` gate by gate, until you come to a flip-flop or latch output. Specify the flip-flop or latch output as a zero frequency clock source so that precompile will consider it a clock. Now, start at the clock input of this flip-flop or latch, and continue tracing back the clock transition until you come to the ultimate source of the clock transition. During this tracing, whenever you come to a flip-flop or latch, specify its output as a zero frequency clock source, and continue tracing from the clock input.
- If there is a `(clk)` annotation, then check this entire loop in the `PDB/*.loops` file. All nets in the loop should have a `(clk)` annotation, that is, `(clk-ftrace)` or `(clk-btrace)`. If not, contact Palladium Z1 Customer Support, since this is probably a bug. If all nets have a `(clk)` annotation, you must guide the loop-breaking process by telling it where not to put a loop-break. Use FullVision waveforms to trace the signal transition back from the output (say, `net <B>`) of the gate whose output transition was delayed, to a flipflop or latch output net (say, `net <A>`). Add the `noBreakPath -add`

© 2021 All Rights Reserved.

`{<A> <B>}` command to your compile script and recompile. The `noBreakPath` command ensures that no breaks are inserted on any combinational path from `<A>` to `<B>`. After recompiling, a break will no longer be inserted at the input of the gate that drives `<B>` — the break will be moved to another place in the loop. This new break position might still be harmful. In such cases, Cadence recommends you to use the existing `noBreakPath` command, however, you should repeat the same procedure again, to find another `noBreakPath` pair `<C>` `<D>` which guides loop-breaking not to put a break at the new harmful position.

Designs Containing Latches

For a latch, unlike a flip-flop, loop-breaking considers there to be a path from D to Q. As a result, precompile tends to find many loops in designs that have many latches.

To make loop-breaking manageable in designs containing latches, precompile uses a technique called exclusivity point analysis. If precompile can prove that a particular latch D never changes when G is active, it can safely add a delay on the latch D input pin. (The delay is one FCLK in 2X or higher mode, and one-half FCLK in 1X mode.) Such a latch D input is called an *exclusivity point*. Exclusivity point delays reduce the need for loop-breaking and can shorten the design critical path as well (thus reducing step count). An exclusivity point delay never affects the function of the design, nor the waveform of any signal.

Precompile cannot recognize all possible exclusivity points, but it recognizes simple cases where, for some signals C, E1, and E2,

- A latch is enabled only when E1 and C is true, and
- The D input of this latch changes only when E2 and ~C is true

Some latch-based designs obey a design rule called *2-phase non-overlapping clocking*. A set of latches obeys this rule if it can be divided into two groups, where latches in each group receive data only from the other group, and latches from different groups are never enabled at the same time. In such set of latches, every latch ideally should be recognized as an exclusivity point. However, if precompile cannot prove the clocks of the two groups are non-overlapping, it will not recognize the exclusivity points.

In a latch-based design using 2-phase non-overlapping clocking, if there are many latches not recognized as exclusivity points, you might be able to increase the number of exclusivity points in the following ways:

- You can identify pairs (or even n-element sets) of clocks as non-overlapping. You can specify as many pairs (or sets) as you want. Specify them before precompile. Examples:

```
XE> noOverlapClocks -add {A B}
```


© 2021 All Rights Reserved.

means A, B are never simultaneously high.

```
XE> noOverlapClocks -add {C D ~ E ~ U1__F}
```

means no two of C, D, ~E, ~U1__F are ever simultaneously high.

- You might be able to simplify the clocking, or else define extra CAKE clock sources, in a way that allows precompile to prove that clocks are non-overlapping.

Using these techniques can increase the number of exclusivity points identified by precompile. This in turn can avoid the need for extra oversampling, and also reduce the design step count by breaking long combinational paths.

Clock Gating Cells

For clock gating, designs often use a cell with a negative-enabled latch and an AND gate, as in the following diagram:

In this circuit, the ena2 input of the AND gate can change only when the ck input is low. Precompile recognizes the ena2 input of the AND as a *gate-input exclusivity point*, and inserts a delay (one FCLK in 2X mode and one-half FCLK in 1X mode). Similarly, if an OR gate input can change only when the other input of the OR is high, it is also recognized. Like latch D input exclusivity point delay, gate-input exclusivity point delay cannot affect design functionality nor the waveform of any signal. By default, precompile only analyzes direct fanouts of latches (not counting buffers and inverters) and flip-flops as candidates for gate-input exclusivity points. However, if you specify `precompile -moreExclusive`, precompile does not limit the search for gate-input exclusivity points to direct fanouts of latches and flip-flops. Specifying `-moreExclusive` can increase the number of gate-input exclusivity points found. The drawback of this option is a possible increase in precompile execution time.

© 2021 All Rights Reserved.

Classification of Loops

Precompile distinguishes between four types of loops.

- purely combinational loops
- clock loops
- data loops
- set/reset loops

These four types of loop are discovered by precompile in the following way:

First, *purely combinational loops* containing only combinational gates are found and broken. Then, each additional type of loop is found by considering one additional type of path, and loops of the new type are broken. *Clock loops* contain clock to Q paths of latches and flip-flops. *Data loops* contain paths from D to Q of latches. *Set/reset loops* contain paths from set or reset to Q of flip-flops or latches.

Do not assume that a clock loop is broken by delaying a clock path (path from clock source to flip-flop clock pin). For most designs, loop-breaking delay is never added on a clock path.

Using the Precompile -breakHalfCycleAll Command

In 1X mode, by default, each internal delay (added by automatic loop-breaking, breakNet/breakPin, or by a `Q_FDP0B` instance) is either a full FCLK cycle, or one-half FCLK. A `Q_FDP0B` instance gives a half FCLK delay if the net being delayed has a shadow; otherwise it gives a full FCLK. A delay added to an input pin of a gate by loop-breaking, breakNet, or breakPin gives a half FCLK delay if the gate's output has a shadow; otherwise it gives a full FCLK. If there are loops in a part of the design clocked by the fastest design clock, loop-breaking delays are likely to make the design not work, since most nets do not have shadows and most delays will be a full cycle of the fastest design clock.

You can use the `precompile -breakHalfCycleAll` option to specify that internal delays on almost all nets should be only one-half FCLK rather than one FCLK. The option applies to all three types of internal delays: loop-breaking, `breakNet`/`breakPin` or `Q_FDP0B`. For information on which delays remain full FCLK even when `breakHalfCycleAll` option is used, see the description of `precompileOption -add breakHalfCycleAll` command in *VXE Command Reference Manual*.

In some designs, `precompile -breakHalfCycleAll` has a large capacity cost (even doubling the capacity requirement) but in other designs, the option makes the design work without any significant capacity cost.

© 2021 All Rights Reserved.

The following XEL commands give a half-cycle delay on a specific net or pin even when `precompile -breakHalfCycleAll` is not used:

```
XEC> breakNetHalfCycle -add <netname>
XEC> breakPinHalfCycle -add <pinname>
```

In a design where long combinational paths cause a high step count, you might want to use delays to break the long paths and improve the step count. Because of the way a half-FCLK delay is implemented, it will break a loop, but it cannot break a long combinational path to improve step count. However, when `-breakHalfCycleAll` is specified, you can insert a full FCLK delay by specifying a breakNet of 2 units, as follows:

```
XEC> breakNet -add {<netname> 2}
```

Using the Precompile -dontMinimizeDataBreaks (or -dataLoops101) Command

In Palladium release 1.1, precompile's method of breaking of loops that go through latch D inputs (called *data loops*) was changed in the following way. Previously, precompile used a simple method of breaking data loops which often delayed every latch D input through which any loop went. Now, precompile uses a more complex algorithm which tends to minimize the number of delays added, and to delay combinational gate inputs in preference to latch D inputs.

The new algorithm allows some designs to run successfully in 2X mode that previously could only run in 4X mode or higher. However, contrary to expectation, there are a few designs that could run previously in 2X mode, that with the new algorithm can only run in 4X mode or higher. For such designs, the `precompile -dontMinimizeDataBreaks` option gives the old algorithm for data loops. Another name for the same option is -dataLoops101 (this name reflects the fact that this option gives the Palladium 1.0.1 algorithm for data loops).

Similar to `-dataLoops101`, there are also options `-combinLoops101`, `-clockLoops101` and `-resetLoops101`. The `-loops101` option (also known as `-dontMinimizeLoops`) gives the complete old algorithm. These options are rarely used.

A problem with the new algorithm is that for some designs, it takes a long time. By default, precompile will break loops up to a certain *analysis depth* using the new algorithm, and then break the remaining loops using the old algorithm. A message like the following will be printed in `xe.msg`:

```
WARNING (MultiClock#44756): Maximum analysis depth 128 is reached for data loops.
Faster technique will be applied.
```

If this message is printed, it is usually best to eliminate the message by using the `precompile -loopsDepth N` option, where `N` is a number larger than the default (128). However, increasing `N` can cause the precompile step of compilation to take longer.

© 2021 All Rights Reserved.

If you want to reduce the precompile time (at the risk of introducing functional problems due to loop-breaking) you can use `-dontMinimizeDataBreaks` or `-loopsDepth 0`. The `-dontMinimizeDataBreaks` option affects only the analysis of data loops, while `-loopsDepth` affects the analysis of all loops. Another possible problem with the new algorithm is that it usually yields a design with longer combinational paths. In some designs this could increase the step count, which could decrease the emulation speed.

Note: Setting `-dontMinimizeDataBreaks` or `loopsDepth 0` disables, for data loops, some of the heuristics that attempt to avoid delaying clocks.

The *.brk File

Precompile breaks loops by adding delays on input pins of Palladium Z1 *generic library* primitives (combinational gates, flip-flops (FDRSE) or latches (LDRSE).

The file `PDB/<lib>.<cell>.brk` normally reports all the delays added by the compiler. For each primitive where a pin was delayed, the file contains an entry something like this:

```
MX2 SDRAM0.U636.mux_1.mux_1.U00
  <-- O SDRAM0.U636.n1y
  -->*I1 SDRAM0.U519.nor_5.U00.NET00
  =F>*I0 SDRAM0.data_out_9
  -->*S SDRAM0.U518.nor_5.U00.NET00
  ---
```

The symbols `<--` and `-->` indicate output and input pins. Loop-breaking delays are always added on input pins. When delay is inserted on an input pin, a symbol `=F>` or `=H>` is used instead of `-->`. The letters `F` and `H` indicate Full and Half FCLK delays respectively. The letter `X` (as in `XF>` or `XH>`) is used instead of `=` for an exclusivity point delay. This makes it easier for a user to distinguish between safe delays `X` which cannot affect functionality, and unsafe delays `=` which can potentially affect functionality. The letter `U` (as in `UF>` or `UH>`) is used for a user-inserted delay. The symbol `Y->` does not indicate a delay. It represents a property used in `1X` mode shadow analysis: a "both" attribute at a `Y->` input of a gate does not propagate to the gate's output.

In the above example, a full FCLK delay has been inserted at the `I0` pin of the `MX2` primitive whose instance name is `SDRAM0.U636.mux_1.mux_1.U0` (since this pin is marked with `=F>`). This pin is connected to net `SDRAM0.data_out_9`. (The `*` characters at some pins indicate logical inversion.)

If you run precompile with the -keepBreaks option, precompile will use an already-existing PDB/*.brk file if there is one, in the following way. Precompile will first add all the non-exclusivity-point delays from this file, and re-calculate and add all exclusivity point delays. Then, if any loops still exist, precompile will add additional delays to break the remaining

© 2021 All Rights Reserved.

loops, and the *.brk will be updated to include the new delays as well as the original ones. Without -keepBreaks, precompile ignores and overwrites any existing PDB/*.brk file.

If you compile the same design with different software versions, sometimes the number of loop-breaking delays will be different in the different versions. This happens because the loop-breaking algorithms are often improved in a new release. A given delay can break more than one loop. Thus, the number of delays needed to break all the loops in the design can change.

In extreme cases (usually latch-based designs) the number of breaks could be so high that writing *.brk file itself might cause a slowdown. In this case, `xe.msg` will contain a warning:

```
  Warning 44849 Number of breaking points exceeds 100000. File
  PDB/*.brk will not be created. Option -logAllBreaks will force precompile to
create this file.
```

If you want to examine `*.brk` and it does not exist, you might need to use `-logAllBreaks`.

The *.loops File

The `PDB/*.brk` file reports where delays were added, but does not show the actual loops that are broken. To see the actual loops which precompile identified and broke, use the file `PDB/<lib>.<cell>.loops`.

Often, a single loop-break breaks multiple overlapping loops which exist in a design. Precompile does not attempt to report all loops broken by a given break. It reports only one loop for each break.

In the `*.loops` file, by default, only certain types of loops (*combinational loops* and *clock loops*) are reported. To make the report show a loop for every delay in *.brk, use

```
precompile -logAllLoops N
```

where N is a number greater or equal to the number of delays reported in the `*.brk` file. The option `-logAllLoops` N tells precompile to log in the *.loops file, up to N *data loops* and *set/reset loops* in addition to the loops it would normally log.

Precompile limits the size of the `*.loops` file to 128MB and by default stops writing loops when this limit is reached. However, if you use the option `-logAllLoops` (with any N>0), this limit is ignored and precompile will keep writing the file. So, `-logAllLoops 1` might be needed just to observe all combinational and clock loops regardless of the size of `.loops` file.

The special option `-logAllLoops 0` will stop writing .loops file at all (neither combinational nor clock loops will be written). This is not quite logical because for example

© 2021 All Rights Reserved.

`-logAllLoops 1` means "write all combinational and clock loops plus 1 more data or set/reset loop".

In some designs, when `-logAllLoops N` is used, the `*.loops` file might become huge, taking many hours to write and eventually filling the disk. To avoid this problem, run precompile first without `-logAllLoops`, to obtain the `*.brk` file. Then create in the design directory a file named `loop.request`, containing the entries from the `*.brk` file for one or more breaks whose loops you want to investigate. For example, for a break on a mux input, the `loop.request` file can contain the following:

```
MX2 SDRAM0.U636.mux_1.mux_1.U00
     <-- O SDRAM0.U636.n1y
     -->*I1 SDRAM0.U519.nor_5.U00.NET00
     =F>*I0 SDRAM0.data_out_9
     -->*S SDRAM0.U518.nor_5.U00.NET00
     ---
```

Now run precompile again. When a loop.request file is present, precompile reports in the `*.loops` file only loops which are broken at locations specified in the loop.request file. If you use a `loop.request` file, you need not specify `-logAllLoops`.

During the loop-breaking process, sometimes a break is added, but later removed because a subsequent break made the first break unnecessary. When a break is removed in this way, the `*.loops` file still shows the corresponding loop with its break, even though the break has been removed. To know which breaks are retained, you must use the `*.brk` file.

The `-logAllLoops` option can have a side effect of reducing the number of data and set/reset breaks added by loop breaking. The reason for the reduction is that before printing the loop for a given break, precompile performs an extra check to verify that the break has not been made unnecessary by previous breaks. If the break is unnecessary, it is not set. Normally, setting fewer breaks is desirable, but if you want to avoid this effect, use a `loop.request` file to report the loops for specific breaks, and do not use `-logAllLoops`.

In the `*.loops` file, each loop is reported as a sequence of gates between { } characters, as in the example below. Moving downward through the sequence of gates takes you forward along the combinational path. (This is counter-intuitive, as moving downward in the file does not take you backward along the path.)

For each gate, an `@` character shows the input pin on the gate which the loop goes through. An = character shows where the loop was broken. Some nets have the label `(clk ftrace)` or (`clk btrace`), as described in How precompile identifies (clk) nets.

```
Feedback loop (clock) {
     MX2 3gio.hsync_pad.UI101.U4.g2.U00.U00
     <-- O 3gio.hsync_pad.UI101.U4.g2.S1HI
     --> I1 3gio.hsync_pad.rst_or_susp_en_ pwr
     @=> I0 3gio.hsync_pad.vip_hbus_pad_en_
     --> S 3gio.hsync_pad.QT_func_sel[1] (clk ftrace)
     ---
```


© 2021 All Rights Reserved.

```
A2 3gio.hsync_pad.U3.u0.U01@QT1
<-- O hsync
@-->*I1 3gio.hsync_pad.UI101.U4.g2.S1HI (clk ftrace)
--> I0 3gio.hsync_OB
---
FDRSE 3gio.hsync_pad.pad_en__reg.map.u0.U01.U00.U00
<-- Q 3gio.hsync_pad.vip_hbus_pad_en_
--> D 3gio.hsync_pad.pad_en_79 gnd
@--> CK hsync
--> CE
--> R
-->*S reset_tri_
---
} // Feedback loop (clock)
```

Displaying a Loop in the Schematic Viewer

If you have the Novas tools (Verdi or Siloti), you can use the `<install_dir>/share/vxe/gift/misc/loops.qel` script to display a particular loop in the Schematic Viewer. Perform the following steps:

- Perform the steps described in Setting the Environment Variables for Running Novas Tools.
- Invoke xeDebug with the `-fsdb` option.
- Give the following commands:

  ```
  source <install_dir>/share/vxe/gift/misc/loops.qel
  loops <net>
  ```

  where `<net>` is the name of a net within the loop you want to display.

This command sequence brings up the Schematic Viewer and displays in color, any loops reported in `PDB/*.loops` and which contain the specified net. Press the *100%* zoom button to see the loop. Currently, the tool has the limitation that it shows entire nets in color, even though a particular loop actually involves only a single fanout of each net. This can make the schematic confusing.

The command sequence generates two files: `loop_nets.v` and `loop_nets.v.ses`. Once these files are generated, you can view them without running xeDebug, by doing the following:

- Bring up Verdi and Import Design from the `loop_nets.v` file.
- Click *File – Restore Session* and select `loop_nets.v.ses`.

© 2021 All Rights Reserved.

Loops through Memory

Memories with byte-write-enable or bit-write-enable are sometimes modeled using a *read-modify-write* memory model. Such a model uses an extra read port to pre-read data from the address about to be written. The output from the extra read port is given through a mux (each bit controlled by the bit-write-enable) to the memory data input. If there is no flip-flop in this feedback path, it is a combinational loop.

When a memory model contains a combinational loop, the `tmp/<lib>.<cell>.MultiClock` file will contain a warning such as:

```
 WARNING (38412):  Loops through memory detected. See tmp/<lib>.<cell>.MultiClock
for details.
```

In addition, in the loop-breaking section of this file will be an entry as shown below and the `*.loops` file will show loops involving this memory:

```
  memory: MPW2X32 resetseq.resetmemcell.RAMBLK0_0.ET3_MEM_0
```

If the overall memory being modeled is synchronous (that is, it has a clock and registers its inputs) then it is usually possible to model the memory in a better way, without a combinational loop. For examples, refer to Synthesizing Designs with Memories of Chapter 9, "Compiling Designs for In-Circuit Emulation with xeCompile."

Loops Discovered after Precompile

In rare cases, compilation might fail during `et3compile` with a message with a message reporting a logic loop. To avoid this problem, run `precompileOption -breakTerminalTiming` command as described in the Command Reference Manual.

Detecting Clock Delay

In most cases, if loop-breaking delays cause a design to function incorrectly, you can obtain correct function by increasing the CAKE oversampling ratio. However, if loop-breaking delay causes an internal clock net to be delayed, then usually the design will function incorrectly regardless of the CAKE oversampling ratio.

Compiling with the `precompile -verify` option creates a special database that you can use to quickly identify any internal clock net delayed by loop-breaking, and find the particular loop-breaking delay (or other delay) that caused the problem.

The `-verify` option is allowed with CAKE 2 only. If your design requires a higher CAKE oversampling ratio such as 4, a workaround is to add a dummy clock to your design with

© 2021 All Rights Reserved.

frequency double your fastest real design clock, and compile with CAKE 2. The fastest design clock (other than the dummy) effectively runs with CAKE 4, but `-verify` will work properly and not complain.

To add a dummy clock source, the dummy clock source net must exist in the design. You can use `iScopedNets` to add the dummy clock source to the design without modifying the netlist. For more information, see Using a Dummy Fastest Clock Source in Chapter 11, "Modeling Clocks in ICE Mode."

To use the `-verify` option, specify this option in addition to any other precompile options you normally use.

For example:

```
precompile -ignoreEmptyCells -verify
```

or

```
precompileOption -add ignoreEmptyCells
precompileOption -add verify
precompile
```

With `-verify`, CAKE clock generation leaves an idle FCLK cycle after every FCLK cycle when some clock source toggled. This is close to technology CAKE 4 but somewhat different. With CAKE 4 there is an idle FCLK cycle after every FCLK cycle when the fastest clock source toggles, but slower clock sources might toggle at idle cycles of the fastest clock. The `-verify` option ensures that if *any* clock source toggles, then the next FCLK cycle is idle for *all* clock sources.

The term *clock net* is defined as follows: in general, precompile removes most buffers and inverters from the design, placing an inversion property on input pins of gates and storage elements when necessary. A clock net is any net connected directly (after buffer/inverter removal) to a clock pin of a flip-flop or enable pin of a D latch.

A clock net will never change value at an idle cycle unless a loop-breaking or other delay has been introduced on a path between a CAKE clock source and that clock net.

With `-verify`, precompile modifies the emulator's normal triggering mechanism in the following way. When running in STB mode with `sdl -enable`, if any clock net changes value at one of the idle FCLK cycles mentioned earlier, the emulation will stop immediately (without the normal two-FCLK trigger-to-stop delay of Enhanced SDL).

After compiling with `-verify`, run your emulation test almost as you normally would, but with the following changes:

- You must run in STB mode.
- You cannot use the `clockConfig` command.

© 2021 All Rights Reserved.

- Normally, you must modify any `run` *n* commands, doubling the value of *n* for verify mode compared to normal CAKE 2 mode.
- You must make sure any run is done with `sdl -enable` if you want to detect delayed clock transitions that occur during that run. If you use the following state machine,

  ```
  State s0 { if (1==0) { TRIGGER ; } };
  ```

  the emulator will stop only if a delayed clock is detected or the cycle count limit *n* in a run *n* command is reached.
- When using `-verify`, any `run` command in an XEL script must have the `-nowait` option specified. Without -nowait, a delayed clock will not cause the emulation to stop. Therefore, instead of the command `run 500`, you must give the command `run -nowait 500; waitWhileBusy`.

Whenever the emulator stops, you can determine whether the stop was due to the `-verify` feature by using the `value` command on the special signal `qt_verify_CAKE_edges_trigger`. If the value is `1`, the stop was caused by the `-verify` feature. If the value is `0`, the stop was not caused by the `-verify` feature.

The `-verify` feature stops the emulation if a clock has a transition at an idle cycle. You can easily find out which clock net(s) had a transition, as follows: Use the `value` command on all the signals listed in the file `PDB/clock.edges`. For each clock net `N` connected to the clock pin of a rising edge flip-flop or negatively enabled latch, the file has a signal with a name similar to `N@QTposedgeVerify`. This signal has value 1 only if net `N` had a rising edge at this idle cycle. Similarly, for each clock net `M` connected to the clock pin of a falling edge flip-flop or positively enabled latch, the file has a signal with a name similar to `M@QTnegedgeVerify`, which has value 1 only if net M had a falling edge at this idle cycle.

Once the `value` command has returned the clock net that was delayed, set `sdl -disable`, and give the `run 2` command to run two more FCLKs, then upload waveforms in the usual way. You can now use waveforms from incremental upload or offline debug to trace the clock transition back through the design, to find the reason the clock was delayed. No knowledge of the design is required for this tracing; you can trace back the transition at the idle cycle, gate by gate. If two inputs of a gate both have a transition, pick one arbitrarily and continue tracing. You will eventually trace back to a net which is delayed even though the inputs of its driving gate are not delayed. If the delay is caused by loop-breaking, the `PDB/*.brk` file will show the delay on an input pin of the gate.

A current limitation of the `-verify` feature is that if somehow a clock is delayed by two FCLK cycles, the feature will not detect it.

Sometimes, a particular clock net `N` can be delayed, but you know that delay on this net does not cause a problem for the design. In such a case, at run time you can selectively disable the

© 2021 All Rights Reserved.

`-verify` triggering for net `N`. To disable the checking, force low the appropriate net listed in `PDB/clock.edges`.

Rarely-Used Options

With the `-add noLoopBreaks` option of the `precompileOption` command, precompile breaks loops as it normally does (writing `.brk` and `.loops`), but then, if any loops were detected, it fails with a message:

```
Info 39539 Breaking feedback loops was disabled per user request.
```

Loop-breaking in SA Mode

This section describes loop-breaking support for SA mode.

- Making Loop-breaking Delays Visible in Waveforms
- Loops in BFM Modules

Making Loop-breaking Delays Visible in Waveforms

When you upload a waveform in a design compiled using IXCOM, by default, the waveform, might not show every FCLK cycle. Rather, it shows only the last FCLK cycle in each simulation time bin. If the emulation includes behavioral FCLK cycles that consume zero simulation time, some FCLK cycles will not be shown by default. A loop-breaking delay is one FCLK cycle in duration. So, loopbreaking delays are sometimes not visible in the waveform obtained with default options. To make all loop-breaking delays visible, use one of the following methods:

- Use the `-event` option in the `database -open` command. Then, if using Simvision, use one of the options under *View -> Expand Sequence Time*, such as *View -> Expand Sequence Time -> All Time*. If using Verdi, use *Tools -> Event Sequence*. When these options are used, the waveform shows every FCLK cycle where any signal changed value. If there are three FCLK cycles with simulation time 100ns, the times for these cycles are shown as *100ns(1)*, *100ns(2)*, and *100ns(3)*.
- If the design includes uncontrolled clocks, then, instead of using the `-event` option of the `database -open` command, you can use the `database -captureMode uncontrolled` command.

© 2021 All Rights Reserved.

Usually the `-event` option is preferred over `database -captureMode uncontrolled`, because with the `-event` option, the horizontal axis is simulation time rather than FCLK cycle number.

Loops in BFM Modules

For the purpose of loop-breaking, a module completely generated by IXCOM, or a user-defined module which interacts with the testbench, is called a BFM module. Starting from the VXE 16.1.0 release, IXCOM marks the BFM modules by instantiating a `Q_NOT_TOUCH` module inside each BFM module.

Note: Marking a module as BFM does not imply that the sub-modules are BFM.

If every gate (flip-flop, latch, or memory) of a loop is in a BFM module (not in a sub-module), the loop is assumed to be entirely inside the BFM module. If a loop is entirely inside a BFM module and the `precompileOption breakBFMloops` option is not specified, precompile fails with the following ERROR message:

```
ERROR (multclk-1714): Loop in BFM module is detected. See ./PDB/
hdldb.xcva_top.loops.bad for details. To pass compilation you may try
precompileOption -add breakBFMloops, which cannot guarantee BFM correctness.
```

If a loop is partly but not completely inside a BFM module, during loopbreaking, the loop is broken at an input of a gate which is not in the BFM module, and no error is generated.

If you specify the `precompileOption breakBFMloops` option, a loop, which is entirely inside a BFM module does not cause an ERROR, and during precompile the loop is broken according to the algorithms that are normally used for loops in non-BFM modules.

Caution

Loop-breaks in BFM modules have a higher likelihood of causing functional problems than most loop-breaks. When loops in BFM modules are reported, it is recommended that, when possible, you modify the BFM modules to eliminate the loops.

© 2021 All Rights Reserved.

13

Using FIFO

This appendix describes different scenarios where introducing FIFO might be helpful when using Pipelined I/O.

Why Do We Need FIFO?

When using the Pipelined I/O capability on a target interface, the input signals incur a delay of one emulation FCLK cycle prior to being received by the DUT. This delay might change the alignment of input data relative to the the target interface clock, and result in lost or duplicate data values. The impact of one-FCLK latency depends on the clock CAKE mode and the frequency of the target interface clock relative to the fastest clock in the design. Adding FIFO in the input data path helps to overcome this one-FCLK delay by providing a buffering so that input data can be re-aligned with the target interface clock.

Signal Delays in CAKE Clock Mode

Let us look at a a few examples to see how one-FCLK latency affects input signals in CAKE clock mode and how adding FIFO helps to synchronize output signals with target interface clocks.

In the waveform above, `clk_100_out` represents the fastest clock in the emulated design. While `clk_67_out` represents a clock that has a frequency 2/3 as fast as `clk_100_out`. This design is compiled using CAKE 1X mode. In CAKE 1X mode, each emulation FCLK cycle represents one cycle of the fastest design clock. Therefore, each clock period for `clk_100_out` in the waveform above represents one-FCLK cycle. `clk_67_out` is slower than `clk_100_out`, and has a rising edge in 2/3 of the FCLK cycles. This results in the correct ratio of rising clock edges for `clk_100_out` and `clk_67_out` being produced. This

© 2021 All Rights Reserved.

illustrates how CAKE mode generates clocks of arbitrary frequency. The waveform also shows that the `clk_67_out` signal has an active rising edge in some FCLK cycles, and does not have an active rising edge in other FCLK cycles.

Now, let us consider a target interface where input data is delayed by one-FCLK cycle. In the waveform above, `counter_67` represents a data bus with a data pattern that increments on every rising edge of `clk_67_out`. While `counter_67_dly` represents this data bus delayed by one-FCLK cycle. The waveform illustrates that one-FCLK delay is not necessarily equal to one target clock cycle of delay. When using CAKE clocks, a one-FCLK delay can result in data changing in emulation cycles without an active clock edge. The data might also fail to change when there is an active clock edge. Thus, the data is no longer aligned with the corresponding clock.

The waveform above illustrates the problem of introducing a one-FCLK cycle delay on an interface with CAKE clocks. The `counter_67_dly` signal is registered on the rising edge of `clk_67_out` to produce `counter_67_dly_reg`. The `counter_67_dly` signal (`counter_67` delayed by one-FCLK) is not registered properly relative to `clk_67_out`. As can be seen, some data pattern values are missed and do not show up in `counter_67_dly_reg`. In this case, the data pattern for `counter_67_dly_reg` misses every other data value. For interfaces running on CAKE clocks, introducing a one-FCLK delay on input data might result in the DUT not sampling the input data correctly. Some data may be repeated, while some may be dropped.

© 2021
All Rights Reserved.

Adding FIFO in Input Data to Correct FCLK Delay Issue

The waveform above shows how such one-FCLK latency issue can be addressed by modifying the top-level netlist to introduce a FIFO in the input data path when using Pipelined I/O on a target interface. While the block diagram below it illustrates where in the emulated design, FIFO is added. The FIFO provides buffering so that input data can be re-aligned with the target interface clock – in this case, `clk_67_out`. Input data is written to FIFO using a delayed version of `clk_67_out`, and is read from the FIFO using the original `clk_67_out`. This results in a data bus that is delayed by one target clock cycle, instead of one-FCLK cycle. The FIFO output is aligned with the original target clock with no data lost or repeated. The waveform also illustrates how the addition of FIFO can compensate for CAKE clocks and produce a consistent input data bus. Here, `VXE_fifo_data` represents the output data of the FIFO. `VXE_fifo_data` can be compared to original `counter_67` data bus to see that it is effectively a registered version of the data bus.

FIFO Example Implementation

Use the following example implementation of FIFO to address the Pipelined I/O FCLK latency issue:

```
module VXE_fifo_HDIO (
  clk_in,
  data_in,
  data_out
);

Q_FDP0 U_FDP0_DV0 (.D(clk_div2_inv), .CK(clk_in), .Q(clk_div2));
Q_INV  U_INV_DV0  (.A(clk_div2), .Z(clk_div2_inv));

Q_FDP0A U_FDP0_41 (.D(clk_div2),       .Q(clk_div2_dly1));
Q_FDP0A U_FDP0_42 (.D(clk_div2_dly1), .Q(clk_div2_dly2));
```


© 2021
All Rights Reserved.

```
Q_XNR2  U_XNR2_40 (.A0(clk_div2), .A1(clk_div2_dly2),
                  .Z(clk_inactive));

for (j=0; j<p_data_len; j=j+1) begin : FIFO_gen_block
  Q_FDP0A U_FDP0_S0 (.D(data_out[j]),       .Q(data_out_dly1[j]));
  Q_FDP0A U_FDP0_S1 (.D(data_out_dly1[j]), .Q(data_out_dly2[j]));

  Q_MX02  U_MX02_44 (.A0(data_in[j]), .A1(data_out_dly2[j]),
                  .S(clk_inactive), .Z(data_out[j]));
end

endmodule
```

Adding FIFO When Using HDDC Muxing

The waveform below represents interface signals for a normal Palladium Z1 compile with no Pipelined I/O added. Here, `port_1_txd` and `port_1_rxd` busses use HDDC muxing when connected to the target. Inside the emulator, they appear as 384 bit wide busses. As shown below, the `port_1_txd` bus is aligned with `TX_CLK`.

In the waveform below, we see that the `port_1_rxd` is aligned with `RX_CLK`. When using HDDC Muxing, the input data is sampled by the emulator in the same FCLK cycle as active clock edge.

When using Pipelined I/O, the emulator samples the input data in the next FCLK cycle. This results in a one-FCLK delay on the `port_1_rxd` bus. In the waveform below, we see that this

© 2021
All Rights Reserved.

delay causes `port_1_rxd` to be mis-aligned to `RX_CLK`. A FIFO is thus added to `port_1_rxd` to correct this mis-alignment. The signal `fifo_1_rxd` represents the output of this FIFO. This newly added FIFO signal is aligned to `RX_CLK`, though the data incurs a one target clock delay.

© 2021
All Rights Reserved.

© 2021 All Rights Reserved.

14

Modeling Guide for Verilog Designs for ICE Mode

This appendix describes how to synthesize Verilog designs for emulation in ICE mode using the Palladium HDL-ICE Compiler. It also describes Verilog features that HDL-ICE Compiler does not synthesize, and provides caveats for special cases such as multi-language designs.

Important

This appendix is applicable only to ICE mode. Refer to Chapter 16, “Compiling Designs for Simulation Acceleration Using IXCOM” for details about modeling guidelines for SA mode.

The following topics are discussed in detail in this chapter:

- Creating Synthesizable Verilog Designs
- Considering Synthesis Assumptions
- Modeling Sequential Logic with always Blocks
- Synthesizing Tristate Logic
- Verilog -2001 Configuration
- SystemVerilog Support
- Deviations from IEEE 1800-2012 Standard

Creating Synthesizable Verilog Designs

This section describes how to create a Verilog design for RTL synthesis. HDL-ICE Compiler synthesis procedure is performed using the `hdlSynthesize` XEL command. Refer to the *HDL-ICE Compiler Commands* chapter in the *VXE Command Reference Manual* and Synthesizing the RTL Design for Emulation.

© 2021 All Rights Reserved.

This section assumes that you are already familiar with Verilog as a simulation language. So, this section describes only the differences between Verilog as a simulation language and as a synthesis language. For a complete description of Verilog, see the Verilog documentation included with the Verilog simulation tool.

Using Verilog HDL

Verilog contains a rich set of syntax to describe a digital system design. A synthesizable Verilog circuit can be described in either of the following two ways:

- A structural description models a design as gate-level components, and the connections between them.
- A behavioral description (also called a functional description) models what a design does at the more abstract behavioral or functional level.

The structural and behavioral descriptions can be mixed when modeling a design. Design hierarchy is represented through module hierarchy, and each design module has one corresponding Verilog module. The following subsections explain these description styles.

Using Structural Descriptions

To model a design, a typical netlist description uses signal nets to connect instances of library-specific gates. A Verilog structural description also models the structural view of a design. In addition to library-specific gates, the structural description can also contain Verilog primitive logic and user-defined modules.

Moreover, a signal net can be driven by an arbitrary expression that models the logical value assigned to the net. The continuous assignment capability in Verilog uses an arbitrary expression to assign a value to a net.

The following is a structural description of a 4-bit adder.

```
module adder4 (in1, in2, sum, zero);
  input [3:0] in1;
  input [3:0] in2;
  output [4:0] sum
  output zero;
  fulladd u1 (in1[0], in2[0], 1'b0, sum[0], c0);
  fulladd u2 (in1[1], in2[1], c0, sum[1], c1);
  fulladd u3 (in1[2], in2[2], c1, sum[2], c2);
  fulladd u4 (in1[3], in2[3], c2, sum[3], sum[4]);
  nor u5 (zero, sum[0], sum[1], sum[2], sum[3], sum[4]);
endmodule

module fulladd (in1, in2, carryin, sum, carryout);
  input in1, in2, carryin;
  output sum, carryout;
```


© 2021 All Rights Reserved.

```
  assign {carryout, sum} = in1 + in2 + carryin;
endmodule
```

A behavioral design model describes how the design works in abstract terms. Outputs of a module are described in terms of the inputs of that module, but the model does not describe how the module is implemented.

The key elements of a behavioral description are functions and always blocks. A function implies combinational logic, although it might appear to be sequential. An always block, however, can be sequential, and implies sequential logic.

Using behavioral modeling, the 4-bit adder from the previous example can be described as shown in the code sample below.

```
module adder4 (in1, in2, sum, zero);
  input [3:0] in1;
  input [3:0] in2;
  output [4:0] sum;
  output zero;
  reg [4:0] sum;
  reg zero;

  always @(in1 or in2) begin
    sum = in1 + in2;
    if (sum == 0)
      zero = 1;
    else
      zero = 0;
  end
endmodule
```

Mixing Descriptions

You can mix structural and behavioral descriptions within a Verilog module. In a mixed-level design, the combinational portions are typically described in Verilog functions, always blocks, and continuous assignments. You can use multiple always blocks, which infer flip-flops and latches, to maintain state values between different clock cycles.

The following code sample shows the 4-bit adder from the preceding examples, described in a mixed mode.

```
module adder4 (in1, in2, sum, zero);
  input [3:0] in1;
  input [3:0] in2;
  output [4:0] sum;
  output zero;
  reg zero;

  fulladd u1 (in1[0], in2[0], 1'b0, sum[0], c0);
  fulladd u2 (in1[1], in2[1], c0, sum[1], c1);
  fulladd u3 (in1[2], in2[2], c1, sum[2], c2);
  fulladd u4 (in1[3], in2[3], c2, sum[3], sum[4]);
```


© 2021 All Rights Reserved.

```
  always @ (sum)
    if (sum == 0)
      zero = 1;
    else
      zero = 0;
  end
endmodule

module fulladd (in1, in2, carryin, sum, carryout);
  input in1, in2, carryin;
  output sum, carryout;
  assign {carryout, sum} = in1 + in2 + carryin;
endmodule
```

Using Module Hierarchy

Verilog uses module blocks to describe a unit of logic in your design. Module hierarchy represents the design hierarchy—that is, an instance within a module represents another module in the design hierarchy. HDL-ICE Compiler can synthesize hierarchical Verilog designs.

The mixed-level version of the 4-bit adder example uses a two-level hierarchy, where the top-level `adder4` module contains the `u1` instance, which is an instance of another design hierarchy, the 1-bit full adder.

HDL-ICE Compiler supports a majority of Verilog constructs, including all structural elements and most behavioral (functional) statements. However, within an expression, HDL-ICE Compiler synthesis does not support operators used exclusively for simulation.

Identifying Simulation Directives

HDL-ICE Compiler synthesis process accepts, but ignores, Verilog simulation directives. Following are the examples of simulation directives:

- `` `accelerate ``
- `` `celldefine ``
- `` `default_nettype ``
- `` `endcelldefine ``
- `` `endprotect ``
- `` `expand_vectornets ``
- `` `noaccelerate ``
- `` `noexpand_vectornets ``

© 2021 All Rights Reserved.

- `` `nounconnected_drive ``
- `` `noremove_netnames ``
- `` `protect ``
- `` `remove_netnames ``
- `` `reset ``
- `` `timescale ``
- `` `unconnected_drive ``

Identifying System Tasks

All Verilog system tasks begin with the dollar sign ($), and are usually used for waveform monitoring and input/output generation during simulation. HDL-ICE Compiler synthesis procedure accepts system tasks, but issues warnings when they occur. Following are examples of system tasks:

$finish	$log	$showvars	$time
$input	$monitor	$stop	$write
$list	$output	$strobe	

Identifying Keywords

Keywords are predefined, non-escaped identifiers that define Verilog constructs. These keywords cannot be used as variable names, unless the backslash (\) escape character is used.

Following is a list of Verilog keywords:

always	and	assign
begin	buf	bufif0
bufif1	case	casex
casez	cmos	deassign
default	defparam	disable

© 2021 All Rights Reserved.

else	end	endcase
endfunction	endmodule	endprimitive
endtable	endtask	event
for	force	forever
fork	function	highzo
highz1	if	initial
inout	input	integer
join	large	medium
module	nand	negedge
nmos	nor	not
notif0	notif1	or
output	parameter	pmos
posedge	primitive	pulldown
pullup	pull0	pull1
rcmos	reg	release
repeat	rnmos	rpmos
rtran	rtranif0	rtranif1
scalared	small	strong0
strong1	supply0	supply1
table	task	time
tran	tranif0	tranif1
tri	triand	trior
trireg	tri0	tri1
vectored	wait	wand
weak0	weak1	while
wire	wor	xnor
xor		

© 2021 All Rights Reserved.

Identifying Unsupported Statements

HDL-ICE Compiler synthesis process does not support the following Verilog statements, and reports error messages if it encounters them:

- `deassign`
- `defparam` (with upward reference in design hierarchy)
- `event (control)`
- `force`
- `fork`
- `join`
- `release`
- `repeat`
- `wait`

Note: During synthesis, most names in the design are preserved, but the names of blocks, tasks, or functions are not preserved.

Considering Synthesis Assumptions

HDL-ICE Compiler assumes the following when synthesizing Verilog RTL designs.

Constant Expressions

The synthesis process evaluates constant expressions, and propagates them as far as practical, to reduce the amount of circuitry required. No circuitry is synthesized to compute the constant value of the expression.

Size of Constant Numbers and Parameters

The default size of an unsized constant or parameter is 32 bits.

Size of Integer Declaration

The default size for integers is 32 bits. the synthesis process calculates the maximum required bit-width, and assigns the appropriate size.

© 2021 All Rights Reserved.

Part Select

A part-select expression selects a group of bits from a wire, register, or parameter vector. Unlike the bit-select operator, this expression must be constant. It must also follow the same ascending or descending index order as originally declared for the variable to which you apply the part-select.

Comparison to X or Z

The synthesis process ignores a comparison to *x* (unknown) or *z* (high impedance)—that is, the comparison always evaluates to false. A warning message is displayed, because such treatment might cause simulation results to mismatch synthesis results.

Delays and Drive Strengths for Module Statements

HDL-ICE Compiler synthesizes weak, pull, and high drive strengths; but it does not distinguish between the weak and pull strengths. HDL-ICE Compiler synthesis accepts, but ignores, delay values or other drive strength values associated with module statements, or with constructs for simulation.

Undriven or Multiply Assigned Wire

HDL-ICE Compiler synthesis process leaves undriven wires unconnected. For multiple connections or assignments to a wire, HDL-ICE Compiler either creates logic to resolve them correctly, or reports an error message.

Assignment to Tristate

All but one variable that drives the tristate should have a value of z (high-impedance). This variable determines the value of the tristate. You must ensure that this rule is followed, because HDL-ICE Compiler synthesis does not check or enforce it.

Continuous Assignment

If the number of bits on the left side of an assignment is greater than the number on the right side, then HDL-ICE Compiler synthesis adds zeroes to extend the right-side operands.

© 2021 All Rights Reserved.

Parameters in Module Instantiations

Verilog lets you override parameter values in a module during instantiation, using either the `defparam` statement or the following syntax:

```
moduleName #(parameterValue, parameterValue, ... instanceName
```

Because the parameter overrides in the design hierarchy can change the module name in synthesized netlist, HDL-ICE Compiler does not support use of a parameterized module name as first segment of the hierarchical reference. Instead, you can use either of the following:

- Use an instance name as first segment of the hierarchical reference.
- Use a non-parameterized module that is higher up in the design hierarchy as first segment of the hierarchical reference.

Modeling Sequential Logic with always Blocks

Verilog models sequential logic through the `always` block. This section describes modeling guidelines, all of which are compatible with established industry practice.

Two types of `always` blocks are available:

- Combinational logic, such as:

  ```
  always @ (a or b) begin
    f = a & b;
  end
  ```

- Sequential logic, such as:

  ```
  always @ (posedge ck) begin
    f = a & b;
  end
  ```

The main difference between the above types is how they trigger events.

An always block contains logic that responds as follows:

```
always @ (event-expression) begin
  ... statements ...
end
```

The event-expression declares triggers, grouped by the word or. If any of the triggers in the event-expression occur, then the always block is executed.

Two types of triggers are used:

© 2021 All Rights Reserved.

- A combinational `always` block triggers based on changes in the logic levels. Interpreting a combinational `always` block is straightforward, except while making an inference to latches (see Latch Logic Inference on page 488).
- A sequential `always` block triggers based on changes of signal edges (rising or falling edges).

Register Inference

Any variable assigned a value from within a sequential always block, is implemented as a D-type edge-triggered flip-flop (DFF). The clock of the DFF is the trigger, even if only one such trigger exists, or one of the triggers is based on modeling guidelines.

If only one trigger exists, then the DFF is clocked using the signal and edge specified in the event-expression. In the following example of clock edges in a sequential always block, the DFFs that implement `f` and `g` are clocked at the rising edge of `ck1`, and `h` and `i` are clocked at the falling edge of `ck2`.

```
always @ (posedge ck1) begin
  f = a & b;
  g = x & y;
end

always @ (negedge ck2) begin
  h = a & b;
  i = x & y;
end
```

Asynchronously Reset Registers

To model a DFF that has an asynchronous reset, you must follow a modeling template. The following is a basic example showing register inference with a basic asynchronous reset. The DFF is clocked by CK on the rising edge, and has an asynchronous reset following the falling edge of RST.

```
always @ (posedge CK or negedge RST) begin
  if (!RST ) begin
    ... statements ...
  end
  else begin
    ... statements ...
  end
end
```

The following example infers DFF registers with two asynchronous resets: RSTa on the rising edge and RSTb on the falling edge.

```
always @ ( posedge CK or posedge RSTa or negedge RSTb) begin
  if (RSTa) begin
    ... statements ...
```


© 2021 All Rights Reserved.

```
  end
  else if ( !RSTb) begin
    ... statements ...
  end
  else begin
    ... statements ...
  end
end
```

To model DFF registers that have asynchronous resets, use the following template as a modeling guideline:

```
always @ (posedge clock or posedge reset1 or negedge reset2) begin
  if (reset1) begin
  /* statements for asynchronous inputs to flip flops
  * triggered by reset1
  */
    ...statement1...
  end
  else if (!reset2) begin
  /* statements for asynchronous inputs to flip flops
  * triggered by reset2
  */
    ...statement2...
  end
  else begin
  /* no explicit event reference here */
  /* assignments to D-flip flop inputs */
    ...statement3...
  end
end
```

If the final `else` clause assigns a value to a variable, then the only triggering event not referenced in any `if-else` clause, clocks the DFF for that variable. Any assignment to a variable in an `if` or `else if` clause implies an asynchronous reset condition for the corresponding DFF.

The event-expression must list all reset conditions, and must list the clock, with all desired senses of edges. All conditions that the if-else clause tests, such as `reset1` and `reset2`, must match the declared sense of edges for the corresponding triggers in an event-expression. The following are examples:

```
else if (condition)
  // reset on rising edge; posedge appears in
  // event-expression
```

or

```
else if (! condition)
  // reset on falling edge; negedge appears in
  // event-expression
```

Each reset condition must have a single-bit value. Explicitly reference all but one trigger in the if-else-if clauses. A lone trigger, with its declared sense of edge in the event-expression, serves as the clock to the DFFs.

© 2021 All Rights Reserved.

Asynchronously Set/Reset Registers

In the preceding template example, `statement1` and `statement2` are basic statements such as:

```
f = 1'b1;
```

This statement example describes a preset condition, rather than a reset condition. For the corresponding DFF, an asynchronous preset is created, rather than a reset.

A sequential `always` block can assign values to more than one variable. Each variable is subject to a different configuration of set and reset conditions. That is, each variable is synthesized as if each has its own dedicated if-else-if, and else specification.

The following is an example of registers with asynchronous set and reset:

```
always @ (posedge CK or posedge RST or posedge PR) begin
  if (RST) begin
    f = 1'b0;
    g = 1'b0;
  end

  else if (PR) begin
    f = 1'b1;
  end

  else begin
    f = a & b;
    g = x & y;

  end
end
```

The above example produces two DFFs, one for f and one for g. The DFF for f has both asynchronous set (PR) and reset (RST), but the DFF for g has only asynchronous reset (RST).

Modeling a Flipflop Using Delay Cells

Some flipflops cannot be described by using the standard template given earlier, but these can be described by a structural model using the `Q_FDP0B` delay cells.

For example, a flipflop with two clocks, two data inputs, and an active-low asynchronous reset can be modeled in the following way:

```
    module ff_2ck(input rst_l, l1clk, d, siclk, si, output q);
    Q_FDP0B u_rst_l_prev(.Q(rst_l_prev), .D(rst_l));
    Q_FDP0B u_l1clk_prev(.Q(l1clk_prev), .D(l1clk));
    Q_FDP0B u_siclk_prev(.Q(siclk_prev), .D(siclk));
    Q_FDP0B u_d_prev(.Q(d_prev), .D(d));
    Q_FDP0B u_si_prev(.Q(si_prev), .D(si));
    Q_FDP0B u_q_prev(.Q(q_prev), .D(q));
    wire m1 = (siclk && ! siclk_prev) ? ! si_prev : q_prev;
```


© 2021 All Rights Reserved.

```
wire m2 = (l1clk && ! l1clk_prev) ? d_prev : m1;
wire q = rst_l && rst_l_prev && m2;
endmodule
```

The structural model is based on the following idea. Let q, d, and so on be the values of signals q, d, and so on in the current FCLK cycle, and let q_prev, d_prev, and so on be the values in the previous FCLK cycle. The desired behavior of signal q is given by the following equation:

```
q = rst_l && rst_l_prev &&
   (l1clk && ! l1clk_prev) ? d_prev :
   (siclk && ! siclk_prev) ? ! si_prev :
   q_prev;
```

The structural model implements this equation directly.

Modeling a Flipflop Using Multiple always Statements

For some flipflops that cannot be modeled using the standard template, instead of using a model with delay cells, you can use multiple edge-triggered `always` statements that set a single register.

If you use the multiple-`always`-statement approach, you must obey the following rules:

- The sensitivity list for each always statement must consist of a single edge (posedge or negedge).
- The register set by multiple `always` statements must be read somewhere outside the multiple `always` statements. If the register is never read, messages like the following are issued:

  ```
  WARNING: Module 'm': net 'q' has no load.
  WARNING: reg 'q' has multiple drivers and its name is not preserved.
  ```

  No logic is synthesized to drive the register.
- The register must not be tri-state (with `'1bz` value assigned to it).
- The register output must not connect directly (or through only hierarchical connections) to a primary output of the design. If it does, then after precompile, the `tmp/drc.rpt` will contain a warning like the following:

  ```
  Warning #100017: Bus repeater is ignored for top terminal q. Pullup is applied.
  ```

  The module will not behave as a flipflop.

For example, the two-clock flipflop `ff_2ck` in the previous section can be implemented in the following way:

```
module ff_2ck(input rst_l, dclk, d, siclk, si, output q);
reg q_internal;
always @ (posedge dclk )
```


© 2021 All Rights Reserved.

```
if(rst_l) q_internal <= d ;
always @ (posedge siclk )
if(rst_l) q_internal <= si;
always @ (negedge rst_l)
q_internal <= 1'b0;
buf u1(q, q_internal);
endmodule
```

In the synthesized module `ff_2ck`, `q_internal` is a tristate net with a retain-state cell (`Q_BRP`). For each `always` statement, the `q_internal` net in the synthesized module has a tristate buffer whose `enable` is active when the edge condition for the `always` statement is true.

Compared to the model using delay cells, the model using multiple `always` statements has the following advantages:

- The register can be set during Palladium run time. For the model using delay cells, to make sure the register can be set, you must specify its output as a keepNet.
- The model does not use Palladium-specific delay cells in its definition. So, it can be fully simulated by any standard simulator, such as Cadence IUS.

The multiple-`always`-statement approach has the following limitations:

- If both clocks have rising edges at the same time, and the data inputs have different values, the resulting flipflop value is unspecified. If a clock has a rising edge at the same time reset goes inactive, the resulting flipflop value is unspecified.

  For the model using delay cells, the behavior in Palladium is fully specified. If both clocks have rising edges at the same time, the flipflop gets the value of `d`. If a clock has a rising edge at the same time reset goes inactive, the flipflop remains reset.

- The emulation resources for the synthesized flipflop can be much higher than for an ordinary flipflop. A flipflop with a single clock and asynchronous reset has an area of 8 gates. The `ff_2ck` flipflop implemented using multiple `always` statements has an area of 67 gates.

  The `ff_2ck` model using delay cells requires more emulation resources than an ordinary flipflop, but not as much as for the multiple-`always` implementation. The area for the structural implementation of `ff_2ck` is 20 gates.

Latch Logic Inference

To maintain consistency between simulation and synthesis, latches are sometimes inferred and created in a combinational always block. However, if you do not intend such behavior, and you need to eliminate extra latches, then use the guidelines in this section to rewrite portions of the Verilog file.

© 2021 All Rights Reserved.

Verilog register variables can hold their states over time in simulation. To maintain consistency between simulation and synthesis, synthesis must simulate such behavior in synthesized logic. HDL-ICE Compiler synthesis inserts and implements qualifying variables, which require such behavior and use a D-type flow-through latch. This latch allows a variable to hold its value (state) until that value is reassigned.

A qualifying variable needs to hold its state if its previous value changes, but at the same time the current calculation requires that previous value. The following is an example of latch inference with an `if` statement:

```
always @ (cond or a or b) begin
  if (cond) begin
    x = a & b;
  end
end
```

In the above example, if `cond` is false, then the `x` variable should retain its previous value. The same combinational logic that implements the new value of `x` when `cond` is true (the normal condition), should be blocked when `cond` is false.

To block this logic when `cond` is false, the following occurs:

- To retain the previous value of `x`, a latch is created for `x`.
- The data input of the latch is connected to (`a & b`).
- The latch output is connected to `x`.
- `cond` controls the latch gate.

The following example shows the behavior of the new implementation in the rewritten description. In the preceding example, the if statement controls the condition under which x must retain its previous value. A case statement can also control these conditions, as in the following example:

```
input [1:0] d;

always @ (d or a or b) begin
case (d)
  2'b00: x = a & b;
  2'b01: x = a | b;
endcase
end
```

To avoid unwanted latches, make sure that the variable is assigned a value under all circumstances. You can rewrite the two preceding examples to avoid creating latches, as shown in the following two examples:

```
always @ (cond or a or b) begin
  if (cond) begin
    x = a & b;
  end
```


© 2021 All Rights Reserved.

```
  else begin
    x = 1'b0;
  end
end
input [1:0] d;

always @ (d or a or b) begin
case (d)
  2'b00: x = a & b;
  2'b01: x = a | b;
  2'b10: x = 1'b0;
  2'b11: x = 1'b0;
endcase
end
```

Synthesizing Tristate Logic

You can synthesize the tristate logic embedded in a Verilog description using either of two methods: the `hdlSynthesize` XEL command.

- Select the toolbar menu Command: *Design>HDLCompiler>Synthesize*

Examples of Verilog Tristate Logic

Tristate logic typically involves testing a condition under which a z (high-impedance) value should be assigned to a variable.

The following Verilog code segments require testing conditions:

- IF Statements
- Conditional Assignments
- Case Statements

The following is an example of synthesizing tristate logic with an *if statement*:

```
always @ (ENA or din) begin
  if (ENA)
    dout = 1'bz;
  else
    dout = din;
  end
end
```

The following is an example of synthesizing tristate logic with a *conditional assignment*:

```
always @ (ENA or din) begin
  dout = ENA ? 1'bz : din;
end
```


© 2021 All Rights Reserved.

The following is an example of synthesizing tristate logic with a `case` *statement*:

```
always @ (d or a or b) begin
case (d)
  2'b00: x = a & b;
  2'b01: x = a | b;
  2'b10: x = 1'bz;
  2'b11: x = 1'bz;
endcase
end
```

Enabling or Disabling Memory Synthesis

HDL-ICE Compiler will automatically choose to use the `hardmacro` memories for constructs for which they are suitable. In addition, HDL-ICE Compiler provides fine granular control of memory synthesis with two synthesis directives.

- `no_hardmacro`
  This command disables memory synthesis for the signal specified. For example, if there is a memory declared in the RTL:

  ```
  reg [1024:9] mem [23:13];
  ```

  You can add the synthesis directive after the declaration,

  ```
  //quickturn no_hardmacro mem
  ```

- `use_hardmacro`

  This command forces the signal to be synthesized into hardmacro memories. The usage is similar to `no_hardmacro`.

Both synthesis directives are effective when memory synthesis option is turned on. They can be used with either Verilog or VHDL designs.

Out of Module Reference

HDL-ICE Compiler currently supports out of module references only if the objects are directly inside the module, and not in any nested code. OOMRs are supported only if the objects are nets or variables of a scaler or vector type. They are not supported if the objects are a parameter, function, or task. Multiple-dim arrays are also not supported. An OOMR that refers to a different object in a different instantiation is not supported. Out of module references to memory nets or variables is not supported, except when used in `$readmem` system tasks.

Import all objects referenced by the OOMR in the same `hdlImport` session: Cadence recommends that OOMRs should not be used with an incremental import.

© 2021 All Rights Reserved.

If the object referenced by the OOMR is not imported in the same `hdlImport` session, it is assumed to be scalar or vector, with a range that minimally covers all indexes used in bit/part-select of all OOMR's referring to it. For example: if `a.b.c[2:4]`, `a.b.c[5]`, `a.b.c[1:3]` are used, `c` is assumed to be a vector of `[1:5]`. Only locally static indexes are taken into account. If no bit/part-select is used, the OOMR is assumed to be scalar. If the OOMRs occur in different `hdlImport` sessions, a warning message is generated stating the assumptions made by HDL-ICE Compiler.

Note: Hierarchical reference into a procedural block is not supported.

Verilog -2001 Configuration

HDL-ICE Compiler provides limited support of Verilog-2001 configuration with the following restrictions:

- The configuration must be imported with the corresponding module in the same import session, although they can be in different files.
- The `design` statement can only contain a module name.

  For example:

  ```
  design <cell>;
  ```

  The following yields an error from HDL-ICE Compiler

  ```
  design [<library>.]<cell>;
  ```

- The `default` clause is fully supported. The libraries specified are linked in the same manner as in VHDL.
- In an `instance` clause, only instances under the current module (one level down) are supported, and only one library is allowed in the `liblist` clause.

  For example:

  ```
  instance top.a1 liblist mylib1;
  // this is ok
  ```

  The following yields an error from HDL-ICE Compiler:

  ```
  instance top.a1.a2 liblist mylib1;
  // 2-level hierarchy
  instance top.a1 liblist mylib1 mylib2;
  // 2 libraries in liblist
  ```

- In a `cell` clause, only one library is allowed in the `liblist` clause, similar to an `instance` clause.

  For example:

© 2021 All Rights Reserved.

```
cell adder liblist mylib1;
// this is ok
cell adder liblist mylib1 mylib2;
// this yields an error
```

- `use` clauses are supported except those with the `:config` suffix. In other words, hierarchical configuration is not supported, but `use <lib.>cell` is supported.

  For example:

  ```
  instance top.mux2 use mux4;
  // use mux4 in the same lib as top
  cell add1 use add4;
  // use mux4 in the same lib as top
  instance top.mux2 use mylib3.mux4;
  cell add1 use mylib3.add4;
  ```

  The following yields an error from HDL-ICE Compiler

  ```
  instance top.mux2 use mylib3.mux4:config;
  ```

- The library map file (IEEE1364-2001 [13.2]) is not supported. Continue to use the existing method to specify libraries; `hdlDefinePath` in XEL or `va.lib` on the UNIX command-line interface.

SystemVerilog Support

By default, the IEEE Verilog-1995 standard is assumed for a Verilog input file. To enable recognition of the SystemVerilog keywords, use the `-sv` option with either the `hdlImport` XEL command or the `vavlog` UNIX command as follows:

```
hdlImport -sv
vavlog -sv
```

The following sections describe the support for SystemVerilog by HDL-ICE Compiler:

- Supported SystemVerilog Constructs for IEEE 1800-2005
- Supported SystemVerilog Design Constructs for IEEE 1800-2012
- Supported SystemVerilog Testbench Constructs for IEEE 1800-2012
- Deviations from IEEE 1800-2012 Standard

Supported SystemVerilog Constructs for IEEE 1800-2005

The following section lists the supported SystemVerilog features with their corresponding LRM section numbers in brackets [ ]. Refer to IEEE Standard 1800-2005 for these sections.

- Literal Values [3]

© 2021 All Rights Reserved.

- Data Types [4]
- Arrays [5]
- Data Declarations [6]
- Operators and Expressions [8]
- Procedural Statements and Control Flow [10]
- Processes [11]
- Tasks and Functions [12]
- Assertions [17]
- Hierarchy [19]
- Interfaces [20]
- System Tasks and System Functions [22]
- Compiler Directives [23]
- Packages [26]
- Formal Syntax [Annex A]
- Encryption/Decryption Flow [Annex O]

Literal Values [3]

- unsized literals [3.3]
- `shortreal` literals (parsed then ignored) [3.4]
- time units in literals (parsed then ignored) [3.5]
- string literals [3.6]
- array literals [3.7]

Data Types [4]

- Integer data types including `logic`, `integer`, `bit`, `byte`, `shortint`, and `longint` [4.3]
- unsigned integer type [4.3.3]

```
integer unsigned n ;
```


© 2021 All Rights Reserved.

- `shortreal` data type (recognized but not synthesized) [4.4]
- user-defined type [4.9]

  Limitation: Forward `typedef` is not supported
- enumeration type [4.10]
  - `typedef enum` [4.10.1]

    Limitation: Forward `typedef` is not supported
  - enum type ranges [4.10.2]

    ```
    enumerated data type shorthand (name [N]), [N]=C) (name [N:M]); (name
    [N:M]=C)
    ```
  - enum type checking [4.10.3]
  - enum type methods [4.10.4]
    - enum type method first() [4.10.4.1]
    - enum type method last() [4.10.4.2]
    - enum type method next() [4.10.4.3]

      ```
      typedef enum {e0,e1,e2,e3} t ;
      t x,y ;
      y = x.next() ;
      ```
    - enum type method prev() [4.10.4.4]

      ```
      typedef enum {e0,e1,e2,e3} t ;
      t x,y ;
      y = x.prev() ;
      ```
    - enum type method num() [4.10.4.5]
    - enum type method name() [4.10.4.6]

      ```
      typedef enum logic [1:0] {e0,e1,e2,e3} t ;
      wire [15:0] w ;
      assign w = e2.name() ;
      ```

  Limitation: enum type method name() [4.10.4.6] is not supported.
- Unpacked structure [4.11]

  Limitation: The `initial` value is ignored
- Packed structure [4.11]

  Limitations:
  - The `initial` value is ignored.

© 2021 All Rights Reserved.

- Treated only as vector at run time.

```
struct packed unsigned {integer a ; logic [7:0] b ;} pack1 ;
```

- Packed union [4.11]

```
typedef union packed
{
logic [15:0] bit_slice ;
logic [1:0][7:0] byte_slice ;
} word_type ;
```

Limitations:

- The initial value is ignored.
- Treated only as vector at run time.

Arrays [5]

- Packed and unpacked arrays [5.2]
- Positive number for range of unpacked dimension [5.2]

Note: `[size]` is equivalent to `[0:size-1]`

```
logic [3:0] A[8] ; // that is, logic [3:0] A[0:7]
```

- Multiple dimensions array port [5.3]

```
module top #(width=2,depth=4)
(
input [depth:1][width:1] i,
output [depth:1][width:1] o,
) ;
...
endmodule
```

- Multiple dimensions array parameter [5.3]

```
parameter [3:0][7:0] p = `{8'h00,8'h01,8'h10,8'h33} ;
```

- Indexing and slicing of arrays [5.4]
- Array querying functions [5.5]

`$left`, `$right`, `$low`, `$high`, `$increment`, `$size`, `$dimensions`

- Array assignment [5.7]

```
int A[10:1];
```

fixed-size array of 10 elements

```
int B [0:9];
```

fixed-size array of 10 elements

© 2021 All Rights Reserved.

```
int C [24:1];
```

fixed-size array of 24 elements

```
wire [31:0] W [9:0];
A = B;
```

Correct usage: compatible type and same size

```
A =C;
```

type check error: different sizes

```
assign W = A;
```

Correct usage

- Array as arguments [5.8]

```
task fun (int a[3:1][3:1])
```

Given the above declaration of `fun`, consider the following actual:

```
int b[3:1][3:1];
```

Correct usage: same type, dimension, and size

```
int b[1:3][0:2];
```

Correct usage: same type, dimension and size (different range)

```
int b[3:1][4:1];
```

Error: incompatible size (3 vs. 4)

Data Declarations [6]

- `type` parameter [6.3.3]

```
module top (input [3:0] x,output [3:0] y) ;
sub #(logic [3:0]) s (x,y) ; // instantiate with type override
endmodule
module sub #(parameter type t = logic) (input t a,output t b) ;
assign b = a ;
endmodule
```

Limitations:

  - The `type` parameter override is not supported across different import sessions. Instantiating module and sub-module with the type parameter must be imported together.
  - Command-line override of the `type` parameter is not supported.

- `const` constants [6.3.5]

```
const logic a = 1'b0;
```


© 2021 All Rights Reserved.

- `net` datatype [6.5]

  ```
  wire integer n ;
  ```

- Declarations in unnamed blocks [6.6]
- Continuous assignments to variables [6.7]

  In Verilog, continuous assignments can only drive nets. In SystemVerilog, continuous assignments are allowed to drive nets or variables.

Operators and Expressions [8]

- Assignment operators as statements [8.3]
- Assignment operators as expressions [8.3]
- Postincrement/decrement statements [8.3]
- Preincrement/decrement statements [8.3]
- `++` and `--` as expressions [8.3]
- Concatenation [8.12]
- Assignment pattern for unpacked array/struct [8.13]

  ```
  var int B[4] = '{0,1,2,3} ;
  ```

  Limitations:

  - The obsolete SV 3.1.a assignment pattern syntax is not supported.
  - Type key for assignment pattern is not supported.

- Aggregate expressions [8.15]
- Streaming operator : as expression [8.17]

  ```
  logic [3:0] A[3:0] ;
  {<< {A}} ;    // pack array bits in right-to-left order
  // that is, {A[0][0],...,A[0][3],...,A[3][0],...,A[3][3]}
  ```

  Limitations:

  - Only streaming operator used as expression will be supported.
  - Streaming operator used as assignment target is not supported.

- set membership (inside operator) [8.19]

  ```
  assign y = x inside {4'h4,4'h1} ;
  assign z = x inside {[4'h4:4'h9],[4'1:4'h3]} ;
  ```


© 2021 All Rights Reserved.

Limitations:

- '$' in open_range is not supported.
- Evaluation context not considered, all operands are treated as self-determined. (This aspect is not specified in the IEEE standard.)
- Recursive application of inside operator into unpacked array in right operand is not supported.

Procedural Statements and Control Flow [10]

- Blocking and non-blocking assignments (time units) [10.3]

  SystemVerilog allows a time unit to be specified.

  ```
  #1ns  r = a;
  r = #1 ns a;
  r <= #1 a;
  ```

  The time unit is parsed but ignored.

- Selection statements -- `if` [10.4]
- Selection statements -- `case` [10.4]
- `do while` loop [10.5.1]
- Enhanced `for` loop [10.5.2]

  ```
  for (int i=0,j=0 ; i<n ; i++,j++)
  ```

  Note: Multiple initialization and step statements are supported.

- Jump statements (`return`, `break`, `continue`) [10.6]
- Named block (matching end block name) [10.8]
- `iff` event control [10.10]

Processes [11]

- `always_comb` [11.2]
- `always_latch` [11.3]
- `always_ff` [11.4]

© 2021 All Rights Reserved.

Tasks and Functions [12]

- Default direction and type for task/function formal arguments [12.2]

```
function int f (a,b,output c,d) ;
// input logic a,b ;
// output logic c,d ;
```

- Function output/inout port [12.3]

```
function int func (input x,output y) ;
...
endfunction
```

  - `void` functions [12.3.1]

```
function void myfunc (input x) ;
...
endfunction
```

  - Discarding function return values [12.3.2]

```
void'(f(i,o)) ; // used as statement
```

- Task/function default argument values [12.4.3]

```
function int f (input int n=3, input [3:0] a,b,c=4'hf) ;
...
endfunction
// function call with
f(,4'h3,4'h5)          //equivalent to f(3,4'h3,4'h5,4'hf)
```

- Task/function argument binding by name [12.4.4]

```
function int f (input int n, input [3:0] a,b,c) ;
...
endfunction
//function call with
f(3, .c(4'h3), .b(4'h5), .a(4'hc))
```

Clocking block [15]

This is not synthesizable. It is parsed and ignored.

Deferred immediate assertion [16.4]

This is not synthesizable. It is parsed and ignored.

Note: This is not part of the IEEE 1800-2005 SystemVerilog standard. The section number is referring to the IEEE Std 1800-2012 standard.

Assertions [17]

- Sampled value functions [17.7.3]

© 2021 All Rights Reserved.

The following functions are supported:

- `$sampled(<expression>[,<clocking_event>])`

  Return type: type of first argument

- `$rose(<expression>[,<clocking_event>])`

  Return type: bit

- `$fell(<expression>[,<clocking_event>])`

  Return type: bit

- `$stable(<expression>[,<clocking_event>])`

  Return type: bit

- `$past(<expression>[,[<number_of_ticks>][,[<expression>][,`
  `[<clocking_event>]]]])`

  Return type: type of first argument

- System functions [17.10]

  - `$onehot(<expression>)`

    Return type: bit

  - `$onehot0(<expression>)`

    Return type: bit

  - `$countones(<expression>)`

    Return type: not specified in IEEE standard, implemented as `int`

  - `$isunknown(<expression>)`

    Return type: bit

Hierarchy [19]

- package [19.2]

```
package pkg ;
function id (input x) ;
return x ;
endfunction
endpackage
module test (input a,output b) ;
import pkg::* ;
```


© 2021 All Rights Reserved.

```
assign b = id(a) ;
endmodule
```

- Compilation unit scope [19.3]

  Limitations:

  - Only parameter, typedef, task, function, net, variable declarations are supported in compilation unit scope.
  - Only global compilation unit scope is supported, not file compilation unit scope.
  - Compilation unit scope should not be used with incremental import. Only the one with latest HDL import will be supported.
  - When using `vavlog` for importing the source files, all files containing the compilation unit scope should be compiled in the same `vavlog` session.

```
parameter int p = 2 ;
typedef logic myBit ;
module mod  ( … );
...
endmodule
```

- interface array port [19.8]

```
interface itfc () ;
endinterface
module sub (itfc i[0:1]) ; // i[0] and i[1] are interface port
endmodule
```

  Limitation:

  - Sub-module with interface port and its instantiating module need to be imported in the same import session.

- Variable ports [19.8]

  With SystemVerilog, a port can be declared as a variable.

```
module mod ( input byte a, b);
...
endmodule
```

  Only variable ports with basic data type are supported.

- Array ports [19.8]
- instance array [19.11]

  Use of only single expression for range is supported.

```
sub s[4] (a[3:0],b[3:0]) ; // This is same as s[0:3]
module sub (input x,output [3:0] y) ;
// The instance array creates the following instances of the sub module:
//   sub \s[0] (a[3],b[3:0]) ;
//   sub \s[1] (a[2],b[3:0]) ;
```


© 2021 All Rights Reserved.

```
//   sub \s[2] (a[1],b[3:0]) ;
//   sub \s[3] (a[0],b[3:0]) ;
```

- recursive module instantiation [19.11]

```
module test #(W=16)
(input [W-1:0] x,y,input cin,output cout,output [W-1:0] z) ;
if (W<3)
assign {cout,z} = x+y+cin ;
else
begin
test #(W/2) s0 (x[0+:W/2],y[0+:W/2],cin,carry,z[0+:W/2]) ;
test #(W-W/2) s1 (x[W-1:W/2],y[W-1:W/2],carry,cout,z[W-1:W/2]) ;
end
endmodule
```

- Implicit .name port connection [19.11.3]

```
.name (name)
```

- Implicit .* port connection [19.11.4]

Interfaces [20]

- Interfaces [20]

  Limitations:

  - Interface instance is flattened and seen as a collection of objects at run time, objects are named as `<instance_name>_<obj_name>`.
  - Nested interface is not supported, neither is nested module.
  - Hierarchical reference (OOMR) into interface is not supported, only direct reference through interface or modport port is allowed.
  - Instance array of interface is not supported.
  - The interface declaration and all modules and interfaces that use that interface need to be in the same import session.

- always block and continuous assignment in interface declaration [20]

```
interface itfc #(w=4) (input clk,en) ;
logic [w-1:0] din,dout ;
logic clk_out ;
assign clk_out = en ? clk : 1'b0 ;
always @(negedge clk) dout <= din ;
endinterface
```

- named bundle [20.2.2]

```
interface itfc ;
logic req,gnt ;
logic [7:0] addr,data ;
logic start,rdy ;
```


© 2021 All Rights Reserved.

```
endinterface : itfc
module sub (itfc a,input bit clk,avail) ;
always @(posedge clk) a.gnt <= a.req & avail ;
endmodule
module cpu (itfc b,input bit clk) ;
...
endmodule
module test (input logic clk,output logic gnt) ;
logic avail = 0 ;
itfc sb () ;
sub s(sb,clk,avail) ;
cpu c(.b(sb),.clk(clk)) ;
endmodule
```

- Ports in interfaces [20.3]

```
interface itfc (input bit clk) ;
logic req,gnt ;
logic [7:0] addr,data ;
logic start,rdy ;
endinterface : itfc
module sub (itfc a) ;
...
endmodule
module cpu (itfc b) ;
...
endmodule
module test (input logic clk) ;
itfc sb1 (clk) ;
itfc sb2 (clk) ;
sub s1(sb1) ;
cpu c1(.b(sb1)) ;
sub s2(sb2) ;
cpu c2(.b(sb2)) ;
endmodule
```

Limitation:

- Port direction `ref` is not supported, as in module, task, or function.

- Modports [20.4] (only 20.4.1 - 20.4.3)

```
interface itfc ;
     wire a,b,c,d ;
     modport master (input a,b,output c,d) ;
     modport slave (output a,b,input c,d) ;
endinterface
module m (itfc.master i) ;
     ...
endmodule
module s (itfc.slave i) ;
     ...
endmodule
module test ;
     itfc i () ;
     m u1 (i) ;
     s u2 (i) ;
endmodule
```

- Parameterized interface [20.7]

© 2021 All Rights Reserved.

```
interface itfc #(WIDTH=1) (input clk) ;
     logic [WIDTH-1:0] data ;
     ...
     endinterface
```

- System function : `$clog2(<expression>)` [20.8.1]

  ```
  return type : int
  ```

  Non-static argument is supported.

  Note: This feature is not a part of IEEE 1800-2005 SystemVerilog standard; the section number is from the IEEE 1800-2009 SystemVerilog standard.

System Tasks and System Functions [22]

- `$bits()` [22.3]

  Limitation: Argument of `$bits()` can be either `expression` or `type`. Only `expression` argument is fully supported. For `type` argument, only simple type name (from `typedef`) is supported.

  ```
  parameter w = 3 ;
  wire [w:0] d1 ;
  wire [w-1:0] d2 ;
  assign d2 = d1[$bits(d2)-1:0] ;
  ```

Compiler Directives [23]

- `` `define `` macros [23.2]
- `` `begin_keywords `` and `` `end_keywords `` directives [23.4]

  ```
  `begin_keywords "1364-2001"
  ...
  `end_keywords
  ```

  The supported version specifiers include `1364-1995`, `1364-2001`, `1364-2001-noconfig`, and `1800-2005`.

Packages [26]

- Using package in module header [26.4]

  ```
  package pkg ; typedef struct {logic a,b ;} t ; endpackage
  module m import pkg::* ; (input clk,input t d,output t q) ;
  // module m (input clk,input pkg::t d,output pkg::t q) ;
  ```

  Note: This feature is not a part of IEEE 1800-2005 SystemVerilog standard; the section number is from the IEEE 1800-2009 SystemVerilog standard.

© 2021 All Rights Reserved.

Formal Syntax [Annex A]

- local `genvar` declaration [A.4.2]

  ```
  for (genvar i=0 ; i<4 ; i++)
  ```

  Note: This feature appears in Annex A (formal syntax). Semantically, it is not defined in either IEEE 1800-2005 or 1800-2009 SystemVerilog standard.

  Implementation treats the `genvar` as if it were declared in the `generate` block for loop body. There is no implicit scope created similar to procedural for-loop.

Encryption/Decryption Flow [Annex O]

- IP protection [Annex O.4]

  ```
  `pragma protect
  `pragma protect begin
  … text to be protected …
  `pragma protect end
  ```

  Note: This feature is not a part of IEEE 1800-2005 SystemVerilog standard; the section number is from the IEEE 1800-2009 SystemVerilog standard.

Supported SystemVerilog Design Constructs for IEEE 1800-2012

This section lists the supported SystemVerilog design constructs with the corresponding LRM section numbers in brackets [ ]. Refer to IEEE Standard 1800-2012 for these sections.

Note: At present, HDL-ICE Compiler does not support default port values. Refer to LRM section 23.2.2.4 to know about default port values.

All the constructs listed in this section can be synthesized using HDL-ICE Compiler, according to industry-RTL-synthesis standards, and you can verify the functional correctness by using Conformal 14.10.100.

- Type key in struct literals [5.10]
- Type parameter for aggregate types [6.20.3]
- Unpacked union [7.3]
- Packed Structures [7.2.1]
- Net Alias [10.11]
- unique0-if [12.4.2]
- unique0-case [12.5.3]

© 2021 All Rights Reserved.

- Parameterized tasks and functions [13.8]
- Macro default text [22.5.1]
- Macro `__LINE__ and `__FILE__ [22.13]
- modport expression [25.5.4]
- Task or function in interface [25.7.1]
- Task or function in modport [25.7.2]
- Package scope identifier [26.3]

Type key in struct literals [5.10]

```
module test(o1);
typedef logic [1:0] t1;
typedef struct packed {
    t1 a;
    int b;
  } pst1;
typedef struct packed {
    pst1 a;
    int b;
  } pst2;
output pst2 o1[1:0];
assign o1 = '{t1:2, int:3};
endmodule
```

Type parameter for aggregate types [6.20.3]

Support type parameter override with unpacked array/struct type.

```
typedef struct {logic v1,v2 ;} t[4] ;
module test (input clk,rst,cond,input t init,x,output t y) ;
       sub #(t) s (clk,rst,cond,init,x,y) ;
endmodule
```

```
module sub #(parameter type DATA_T=logic[9:0])
       (
       input logic clk,rst,e,
       input DATA_T rv,d,
       output DATA_T q
       ) ;
always_ff @(posedge clk,posedge rst)
       if (rst)
       q <= rv ;
       else if (e)
       q <= d ;
endmodule
```


© 2021 All Rights Reserved.

Arrays of typedef

Support unpacked array of user defined type in typedef.

```
typedef logic [1:0] node ;
typedef enum node {e0,e1,e2,e3} et ;
module test #(n=4) (input node x[n],output et y[n]) ;
       typedef node at[n] ;
       typedef et at2[n] ;
       at r ;
       sub #(at) s (x,r) ;
       assign y = at2'(r) ;
endmodule
module sub #(type t=logic) (input t a,output t b) ;
       assign b = a ;
endmodule
```

Separate packages import

Support using packages that were imported in previous import sessions.

Unpacked union [7.3]

Unpacked Union with Same Number Of Bits

```
module test(i1, o1);
       input logic [3:0] i1;
       output logic [3:0] o1;
       typedef union {
               logic [3:0] a;
               logic [3:0] b;
       } ut;
         ut u1;
       assign u1.a = i1;
       assign o1 = u1.b;
endmodule
```

Unpacked Union with Different Number Of Bits

```
module test(i1, o1);
       input logic [3:0] i1;
       output logic [4:0] o1;
       typedef union {
               logic [3:0] a;
               logic [4:0] b;
       } ut;
         ut u1;
       assign u1.a = i1;
       assign o1 = u1.b;
endmodule
```


© 2021 All Rights Reserved.

Unpacked Union with Different Dimension

```
module test(i1, o1);
       input logic [1:0][1:0] i1;
       output logic o1 [1:0][2:0];
       typedef union {
               logic [3:0] a;
               logic b [1:0][2:0];
       } ut;
         ut u1;
       assign u1.a = i1;
       assign o1 = u1.b;
endmodule
```

Unpacked Union Embedded with Unpacked Union

```
module test( input logic[7:0] in0, output logic [7:0]out0,
       input logic in1, output logic out1) ;
       union {
              logic isfloat;
              union {logic[7:0] opcode; logic [23:0] addr;} n;
       } ir;
       assign ir.n.opcode = in0;
       assign out0 = ir.n.opcode;
       assign ir.n.addr[0] = in1;
       assign out1 = ir.isfloat;
endmodule
```

Unpacked Union Array

```
module test( input logic[7:0] in0, output logic [7:0]out0,
       input logic in1, output logic out1) ;
       union {
              logic isfloat;
              union {logic[7:0] opcode; logic [23:0] addr;} n;
       } ir[1:0];
       assign ir[1].n.opcode = in0;
       assign out0 = ir[1].n.opcode;
       assign ir[1].n.addr[0] = in1;
       assign out1 = ir[1].isfloat;
endmodule
```

Embedded Unpacked Union Array

```
module test( input logic[7:0] in0, output logic [7:0]out0,
       input logic in1, output logic out1) ;
       union {
              logic isfloat;
              union {
                     logic[7:0] opcode;
                     union {
                            logic [10:0] aa1;
                            logic [23:0] addr;
                     } n1;
              } n;
       } ir[1:0];
       assign ir[1].n.opcode = in0;
```


© 2021 All Rights Reserved.

```
        assign out0 = ir[1].n.opcode;
        assign ir[1].n.n1.addr[0] = in1;
        assign out1 = ir[1].isfloat;
endmodule
```

Nested unpacked union arrays

```
module test( input logic[7:0] in0, output logic [7:0]out0,
             input logic in1, output logic out1) ;
      union {
             logic isfloat;
             union {
                    logic[7:0] opcode;
                    union {
                           logic [10:0] aa1;
                           logic [23:0] addr;
                    } n1;
             } n[1:0];
      } ir[1:0];
      assign ir[1].n[1].opcode = in0;
      assign out0 = ir[1].n[1].opcode;
      assign ir[1].n[0].n1.addr[0] = in1;
      assign out1 = ir[1].n[0].n1.addr[0];
endmodule
```

Packed Structures [7.2.1]

This release supports SystemVerilog packed struct mapped to VHDL record.

SystemVerilog packed struct to record

```
typedef struct packed {
    logic           a;
    logic [2:0]     b;
} p1;

module top(
    input  p1 i1,
    output p1 o1,
);

  test i_test (
    .i1(i1),
    .o(o1)
  );
endmodule
```

SystemVerilog array of packed struct to array of record

```
typedef struct packed{
reg [15:0] data1_s;
bit [15:0] data2_s;
} mystruct_in;
```


© 2021 All Rights Reserved.

```
typedef struct packed{
logic [15:0] result_s;
} mystruct_out;

module packed_struct (input logic clk, input logic sel, input mystruct_in s1 ,
output mystruct_out s2);
record_test d0 (.clk(clk), .sel(sel), .input_port(s1) , .output_port(s2));
endmodule

module top(clk,sel,s_in,s_out);
input logic clk;
input logic sel;
input mystruct_in s_in ;
output mystruct_out s_out;

packed_struct dut (.clk(clk), .sel(sel), .s1(s_in), .s2(s_out));
endmodule
```

SystemVerilog packed struct member to record

```
typedef struct packed {
logic        id1;
logic        id2;
logic        id3;
logic [3:0]  id4;
logic        id5;
logic [3:0]  id6;
logic        id7;
} id_t1; // 13

typedef struct packed {
logic       id8;
logic       id9;
logic       id10;
logic [3:0] id11;
logic       id12;
} id_t2; // 8

typedef struct packed {
id_t2 id_02; // 8
id_t1 id_01; // 13
} id_t3; // 21

module top (
 input id_t3 sv_in1,  // 21
 input logic sv_in2   // 1
);

test i_test (
 .vhd_in1    (sv_in1.id_01),
 .vhd_in2    (sv_in1.id_02),
 .vhd_in3    (sv_in2)
);
endmodule
```

Net Alias [10.11]

```
module test (inout wire [31:0] A, inout wire [31:0] B);
       alias alias {A[7:0],A[15:8],A[23:16],A[31:24]} = B;
endmodule
```


© 2021 All Rights Reserved.

Note: Alias could use multiple equal assignments.

```
module test(i1, o1);
       input i1;
       output o1;
       wire i2, i3, i4;
       alias i1 = i2 = i3 = i4;
       assign o1 = i3;
endmodule
```

unique0-if [12.4.2]

```
module test(a);
input integer a;
initial
       unique0 if ((a==0) || (a==1)) $display("0 or 1");
       else if (a == 2) $display("2");
       else if (a == 4) $display("4");
endmodule
```

unique0-case [12.5.3]

```
module test(a);
input integer a;
initial
       unique0 case(a) // values 3,5,6,7 cause a violation report
       0,1: $display("0 or 1");
         2: $display("2");
         4: $display("4");
       endcase
endmodule
```

Parameterized tasks and functions [13.8]

Parameterized function out of module

```
virtual class C#(parameter DECODE_W, parameter ENCODE_W = $clog2(DECODE_W));
  static function logic [ENCODE_W-1:0] ENCODER_f
         (input logic [DECODE_W-1:0] DecodeIn);
    ENCODER_f = '0;
    for (int i=0; i<DECODE_W; i++) begin
      if(DecodeIn[i]) begin
        ENCODER_f = i[ENCODE_W-1:0];
        break;
      end
    end
  endfunction
  static function logic [DECODE_W-1:0] DECODER_f
        (input logic [ENCODE_W-1:0] EncodeIn);
```


© 2021 All Rights Reserved.

```
    DECODER_f = '0;
    DECODER_f[EncodeIn] = 1'b1;
  endfunction
endclass
module test ();
  logic [7:0] encoder_in;
  logic [2:0] encoder_out;
  logic [1:0] decoder_in;
  logic [3:0] decoder_out;
  // Encoder and Decoder Input Assignments
  assign encoder_in = 8'b0100_0000;
  assign decoder_in = 2'b11;
  // Encoder and Decoder Function calls
  assign encoder_out = C#(8)::ENCODER_f(encoder_in);
  assign decoder_out = C#(4)::DECODER_f(decoder_in);
endmodule
```

Parameterized function with default value

```
virtual class graycode#(int W = 32);
  static function logic[W-1:0] bin_to_gray(logic[W-1:0] x);
    return x ^ (x >> 1);
  endfunction
  static function logic[W-1:0] gray_to_bin(logic[W-1:0] x);
    logic[W-1:0] m = x;
    for(int i = 1; i < W; i++) begin
      x = x ^ (m >> i);
    end
    return x;
  endfunction
endclass

module test(
  output logic[31:0] o1,
  output logic[31:0] o2,
  input  logic[31:0] i,
  input  clk
   );
  logic [31:0] o1p;
  logic [31:0] o2p;
  assign o1p = graycode#()::bin_to_gray(i);
```


© 2021 All Rights Reserved.

```
  assign o2p = graycode#()::gray_to_bin(i);
  always @(posedge clk) begin
    o1 <= o1p;
    o2 <= o2p;
  end
endmodule
```

Parameterized function in module

```
module test (input logic[7:0] i);
  virtual class graycode#(int W = 2);
    static function logic[W:0] bin_to_gray(logic[W:0] x);
      bin_to_gray = x;
    endfunction
  endclass

  logic [7:0] p2 ;
  assign p2 = graycode#(.W(7))::bin_to_gray(i);
endmodule
```

Parameterized function in package

```
package pkg ;
  virtual class C #(int W=4) ;
    static function logic [W-1:0] f (logic [W-1:0] v1,v2) ;
      return v1 ^ v2 ;
    endfunction
  endclass
endpackage

module test (input [3:0] x,y,output [3:0] z) ;
  assign z = pkg::C#(4)::f(x,y) ;
endmodule
```

Parameterized function in block

```
module test #(n=4)(input [0:n-1] x[n],y[n],output [0:n-1] z[n]) ;
  for (genvar i=0 ; i<n ; i++)
  begin
    virtual class C #(int W=1) ;
      static function logic [W-1:0] f (logic [W-1:0] v1,v2) ;
        return v1 ^ v2 ;
```


© 2021 All Rights Reserved.

```
      endfunction
    endclass
    assign z[i] = C#(i)::f(x[i][0:i],y[i][0:i]) ;
  end
endmodule
```

Parameterized task enable in always

```
virtual class myClass#(int W = 7);
  static task myTask(input logic[W:0] x, output logic[W:0] y);
    y = x ;
  endtask
endclass

module test(output logic[6:0] o1, input  logic[6:0] i, input  clk);
  always@(*)
  begin
    myClass#(6)::myTask(i,o1);
  end
endmodule
```

Parameterized task enable in initial

```
virtual class myClass#(int W = 7);
  static task myTask(input logic[W:0] x, output logic[W:0] y);
    y = x ;
  endtask
endclass

module test(output logic[6:0] o1, input logic[6:0] i, input clk);
  initial
  begin
    myClass#(6)::myTask(i,o1);
  end
endmodule
```

Macro default text [22.5.1]

```
`define MACRO1(a=5,b=6,c) $display(a,b,c);
module test ();
       initial `MACRO1 (,,3)
endmodule
```


© 2021 All Rights Reserved.

Macro `__LINE__ and `__FILE__[22.13]

Note that for this construct, the `-2009` option of the `vavlog` command should be used.

```
module test;
initial $display("Internal error: null handle at %s, line %d.", \
                    `__FILE__, `__LINE__);
endmodule
```

modport expression [25.5.4]

```
interface itfc () ;
   logic u,v ;
   modport m1 (input .p(u)) ;
   modport m2 (input .p(v)) ;
endinterface

module test (input d,x,y,output z[1:2]) ;
   itfc inst () ;
   sub s1 (inst.m1,x,y,z[1]) ;
   sub s2 (inst.m2,x,y,z[2]) ;
   assign inst.u = d ;
   assign inst.v = ~d ;
endmodule

module sub (itfc i,input logic a,b,output c) ;
   assign c = i.p ? a : b ;
endmodule
```

Note that only simple identifiers are supported for expressions in modport_simple_ports declaration. Complex expressions, such as bit-select or part-select, concatenation, or assignment patterns are not supported.

Task or function in interface [25.7.1]

```
interface itfc #(w=1) (input clk,input [w-1:0] a,b) ;
   logic [w-1:0] r ;
   task t (input logic [w-1:0] u,v,output logic [w-1:0] w) ;
           w = u ^ v ;
   endtask
   always @(posedge clk)
            t(a,b,r) ;
endinterface

module test (input clk,input [1:0] x,y,output [1:0] z) ;
   itfc #(2) inst (clk,x,y) ;
   assign z = inst.r ;
endmodule
```

Note that hierarchical task or function calls through interface instance are not supported.

Task or function in modport [25.7.2]

```
interface itfc () ;
   logic [1:0] r ;
   function void write (input logic [1:0] v) ;
```


© 2021 All Rights Reserved.

```
          r = v ;
   endfunction
   function logic chk (input logic [1:0] v) ;
          return r!=v ;
   endfunction
   modport mp1 (import chk) ;
   modport mp2 (import write) ;
endinterface

module test (input [1:0] x,y,output [1:0] z) ;
   itfc inst () ;
   sub1 s1 (inst,y,z) ;
   sub2 s2 (inst,x) ;
endmodule

module sub1 (itfc.mp1 i,input logic [1:0] a,output [1:0] b) ;
       assign b = i.chk(.v(2'b00)) ? ~a : a ; endmodule

module sub2 (itfc.mp2 i,input logic [1:0] v2) ;
       always_comb i.write(.v(v2)) ;
endmodule
```

Note that hierarchical task or function calls through interface instance are not supported.

Package scope identifier [26.3]

Package Scope Unpacked Struct

```
package pkg;
        typedef struct {
        logic l;
        } s_type;
        s_type s;
endpackage : pkg

module test (input i, output o);
       assign pkg::s.l = i;
       assign o = pkg::s.l;
endmodule
```

Package Scope Packed Struct

```
package pkg;
        typedef struct packed {
                logic [7:0] l;
                byte b;
        } s_type;
        s_type s;
endpackage : pkg

module test (input i, output o);
       assign pkg::s.l[3] = i;
       assign o = pkg::s.l[3];
endmodule
```


© 2021 All Rights Reserved.

Package Scope Unpacked Union

```
package pkg;
        typedef union {
                logic [7:0] l;
                byte b;
        } s_type;
        s_type s;
endpackage : pkg

module test (input byte i, output byte o);
       assign pkg::s.b = i;
       assign o = pkg::s.b;
endmodule
```

Package Scope Packed Union

```
package pkg;
        typedef union packed {
                logic [7:0] l;
                byte b;
        } s_type;
        s_type s;
endpackage : pkg

module test (input byte i, output byte o);
       assign pkg::s.b = i;
       assign o = pkg::s.b;
endmodule
```

Package Scope enum

```
package pkg;
        typedef enum { FALSE, TRUE } bool_t;
                       bool_t e;
endpackage : pkg

module test (output int o);
       assign pkg::e = pkg::e.first() ;
       assign o = pkg::e.next() ;
endmodule
```

Supported SystemVerilog Testbench Constructs for IEEE 1800-2012

This section lists the supported SystemVerilog testbench constructs with the corresponding LRM section numbers in brackets [ ]. Refer to IEEE Standard 1800-2012 for these sections.

The constructs listed in this section are NOT synthesizable, according to industry-RTL-synthesis standards, and you cannot verify the functional correctness by using Conformal. You should, instead, use Xcelium software simulator to verify the functional correctness.

- Hierarchical reference to interface [23.6]

© 2021 All Rights Reserved.

- Hierarchical Task Calls [23.6]
- $countbits, $changed, $sample
- SystemVerilog bind [23.11]

Hierarchical reference to interface [23.6]

If a hierarchical reference to interface is an OOMR (Out of module reference), this hierarchical reference is not synthesizable according to industry RTL synthesis standard and functional correctness cannot be verified by Conformal, should be verified by Xcelium software simulation.

Hierarchical references to nested interfaces are not supported.

Hierarchical Reference to Scalar in Interface

```
interface intf (); logic l; endinterface
module test_mod (); intf inst();
       logic tmp ;
       assign tmp = inst.l;
endmodule

module test_bench (input i, output o);
       test_mod m() ; assign m.inst.l = i; assign o = m.inst.l;
endmodule
```

Hierarchical Reference to Array in Interface

```
interface intf (); logic [7:0] a; endinterface
          module test_mod (); intf inst () ; logic tmp ;
          assign tmp = inst.a[3] ; endmodule
          module test_bench (input i, output o); test_mod m ();
          assign m.inst.a[3] = i; assign o = m.inst.a[3];
endmodule
```

Both packed array and unpacked array are supported.

Hierarchical Reference to Struct/Union in Interface

```
typedef struct { logic l; } USER_DEF_TYPE;
interface intf (); USER_DEF_TYPE s; endinterface
module test_mod (); intf inst ();
       logic tmp ;
       assign tmp = inst.s.l;
endmodule

module test_bench (input i, output o); test_mod m() ;
       assign m.inst.s.l = i; assign o = m.inst.s.l;
endmodule
```


© 2021 All Rights Reserved.

Both packed and unpacked struct/union type are supported.

Hierarchical Reference to enum in Interface

```
typedef enum int { FALSE, TRUE } bool_t;
interface intf (); bool_t e; endinterface
module test_mod (); intf inst (); bool_t tmp;
       assign tmp = inst.e;
endmodule

module test_bench (output int o); test_mod m ();
       assign m.inst.e = m.inst.e.first() ; assign o = m.inst.e.next();
endmodule
```

Hierarchical Reference to Interface Array

```
interface intf (); logic l; endinterface
module test_mod (); intf inst [1:0] (); logic tmp;
       assign tmp = inst[1].l;
endmodule

module test_bench (input i, output o); test_mod m ();
       assign m.inst[1].l = i; assign o = m.inst[1].l;
endmodule
```

Hierarchical Reference to Interface Port Instance

```
interface intf (); logic l; endinterface
module test_mod (intf port); logic tmp;
       assign tmp = port.l;
endmodule

module test_bench (input i, output o); intf inst ();
       test_mod m(inst); assign m.port.l = i; assign o = m.port.l;
endmodule
```

Simple in Module Hierarchical Reference to Interface

```
interface intf (); logic l; endinterface
module test_bench (input i, output o); intf inst ();
       assign inst.l = i; assign o = inst.l;
endmodule
```

Hierarchical Reference to Interface Through if-generate Block

```
interface intf (); logic l; endinterface
module test_bench (input i, output o);
       generate
          if (1) begin: bk intf inst (); logic tmp;
              assign tmp = inst.l;
          end
       endgenerate
       assign bk.inst.l = i; assign o = bk.inst.l;
endmodule
```


© 2021 All Rights Reserved.

Hierarchical reference to interface through for generate block

```
interface intf (); logic l; endinterface
module test_bench (input i, output o); genvar i1;
       genvar i2;
       for (i1=0; i1<5; i1=i1+1) begin :bk intf inst ();
       logic tmp;
       assign tmp = inst.l;
       end
       assign bk[2].inst.l = i; assign o = bk[2].inst.l;
endmodule
```

Hierarchical Task Calls [23.6]

```
module test_mod(); logic g;
       task t(input a, output b); begin
              b = a; g = a; $displ:w ay(b);
         end
       endtask
endmodule
```

```
module test_bench(input clk); test_mod m();
       logic tmp1; logic tmp2;
       always @(posedge clk) begin m.t(tmp1,tmp2);
       end
endmodule
```

Hierarchical task calls are supported with following limitations:

- The task should contain only assignment statements or system task calls.
- The task does not contain other hierarchical reference.
- The task does not contain reference to parameter.

$countbits, $changed, $sample

$countbits [20.9]

The syntax of `$countbits` is:

```
$countbits ( expression , control_bit { , control_bit } )
```

HDL-ICE Compiler supports `$countbits` with no more than four `control_bit`, instead of an infinite number of `control_bit`.

Conformal does not support cases that only expression or `control_bit` contains `x` or `z`. For example:

```
$countbits(a, 'x);
$countbits(4'b110x, a);
```

So, verify the above with Xcelium software simulation.

© 2021 All Rights Reserved.

Exceptions:

Unlike in simulation, `x` and `z` are not propagated in synthesis model. So, the behavior of HDL-ICE Compiler synthesized model is not identical to that of the simulation model if expression or `control_bit` contains signals of `x` and `z`, or signals propagated from `x` and `z`, unless the signals are explicitly initialized in RTL.

For example:

```
case 1: a = 4'b110x;
        $countbits(a, 'x);
case 2: a = 4'b110x; b = in; c = a & b;
        $countbits(c, 'x);
```

In `case 1`, `x` in `a` is explicitly initialized, and the synthesized model and simulation model have identical behavior. On the other hand in *case 2*, the behavior is different.

$changed [20.13]

The syntax of `$changed` is:

```
$changed ( expression [ , [ clocking_event ] ] )
```

The clocking event could be explicitly defined or inferred from the procedural context. For example:

```
case 1: a = $changed(req,@(posedge clk1));
case 2: always@(posedge clk1)
        a = $changed(req);
```

Both of the above cases are legal, and their behavior is identical.

$sampled [20.13]

The syntax of `$sampled` in IEEE standard 1800-2012 is:

```
$sampled ( expression )
```

In the above syntax, clocking event is not used.

On the other hand, in IEEE standard 1800-2005, the syntax of `$sampled` is:

```
$sampled ( expression [, clocking_event] )
```

For downward compatibility, HDL-ICE Compiler supports both syntaxes. If there is an explicit or inferred clocking event, the behavior of `$sampled` follows the rules defines in IEEE standard 1800-2005 section 22.9. Else, its behavior follows the rules defined in IEEE standard1800-2012 section 20.13, that is:

$$sampled(a) \equiv a$$

© 2021 All Rights Reserved.

SystemVerilog bind [23.11]

SystemVerilog `bind` statements are defined in SystemVerilog LRM, section 23.11, binding auxiliary code to scopes or instances.

```
bind_directive ::=
     bind bind_target_scope [: bind_target_instance_list] bind_instantiation ;
     | bind bind_target_instance bind_instantiation ;
```

There are two forms of bind syntaxes:

- Binding instance to instance:

  In the first form, `bind_target_scope` specifies a target scope into which the `bind_instantiation` should be inserted. A bind target scope shall be a module or an interface. A bind target instance shall be an instance of a module or an interface. In the absence of a `bind_target_instance_list`, the `bind_instantiation` is inserted into all instances of the specified target scope, design-wide. If a `bind_target_instance_list` is present, the `bind_instantiation` is only inserted into the specified instances of the target scope. The `bind_instantiation` is effectively a complete module, interface, program, or checker instantiation statement.

- Binding instance to module:

  The second form of bind syntax can be used to specify a single instance into which the `bind_instantiation` should be inserted. If the second form of bind syntax is used and the `bind_target_instance` identifier resolves to both an instance name and a module name, binding shall only occur to the specified instance.

HDL-ICE Compiler supports only the second form of bind syntax, that is binding instance to module.

Example of binding an instance to a module:

```
module A; endmodule
module binder;
       bind A Mbind_0 Ubind_0(); endmodule
module Mbind_0(); endmodule
```

The bind directive can be written in separate files, as follows:

```
test.v module A; endmodule

bind.v

module binder;
       bind A Mbind_0 Ubind_0();
endmodule

module Mbind_0();
endmodule
```

There are some limitations for using `bind` directive in separate files:

© 2021 All Rights Reserved.

- All the files contains `bind_target_scope` and `bind` directive must be compiled with `vavlog` in one line, as follows:

  ```
  vavlog -sv test.v bind.v
  ```

- The SystemVerilog `bind` is compile-order-dependent. The `bind_target_scope` fille(`test.v`) should be compiled before the bind directive file(`bind.v`).

Important

SystemVerilog `bind` statements are not supported by Conformal, and cannot be verified using Conformal 14.10.100. Either you can write the testbench, and verify the resultant netlist by using Xcelium software simulator, or write functionally-equivalent testcases if you want to verify with Conformal 14.10.100.

Deviations from IEEE 1800-2012 Standard

The following section lists certain deviations for Palladium Z1 HDL-ICE Compiler from IEEE standard 1800-2012.

Non-ANSI Style Port Declarations

According to IEEE Standard 1800-2012, section *23.2.2.1 Non-ANSI style port declarations*, the following example is considered as illegal and is not allowed. However, other compiler tools, such as RTL Compiler and Conformal allow such declaration. To maintain the compatibility with such tools, Palladium Z1 HDL-ICE Compiler also allows such declaration.

```
module abc(a[1]);
     input int a[8];
endmodule
```

Bit-Stream Casting

The IEEE Standard 1800-2012, does not permit bit-stream casting on unpacked unions. However, compiler tools such as Conformal and RTL compiler allow the bit stream casting for unpacked unions also. To maintain compatiblity with such tools, Palladium HDL-ICE Compiler also allows bit-stream casting on unpacked unions.

Multidimensional enum Base Types Support

According to SystemVerilog LRM 2009, only one packed dimension is allowed for vectors:

```
enum_base_type ::=
integer_atom_type [ signing ]
```


© 2021 All Rights Reserved.

```
        | integer_vector_type [ signing ] [ packed_dimension ]
        | type_identifier [ packed_dimension ]
```

However, HDL-ICE Compiler supports multidimensional enum base types, and does not issue any error messages in such scenarios.

© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

15

Modeling Guide for VHDL Designs for ICE Mode

This chapter describes the features of VHDL that Palladium Z1 HDL Compiler can synthesize for emulation in ICE mode. It also describes VHDL features that HDL Compiler does not synthesize.

Important

This appendix is applicable only to ICE mode. Refer to Chapter 16, “Compiling Designs for Simulation Acceleration Using IXCOM” for details about modeling guidelines for SA mode.

The following topics are discussed in detail in this chapter:

- Entities and Architectures
- Objects
- Data Types
- Operators
- Operands
- Sequential Statements
- Concurrent Statements
- Subprograms
- Packages
- VHDL Synthesis Directives
- Syntax and Semantic Restrictions
- Modeling Styles
- Support for VHDL 2008

© 2021 All Rights Reserved.

- Deviations from VHDL 2008 LRM

Entities and Architectures

A digital system is usually designed as a hierarchical collection of blocks of logic. Each logic block corresponds to a design unit in VHDL.

Design units represent portions of a hardware design that have well-defined inputs and outputs and perform well-defined functions.

Each design unit has two parts: an entity declaration, and architecture bodies. An entity declaration describes the external interfaces of a component, and architecture bodies describe its internal implementations.

The following example describes an implementation of a 1-bit full adder:

```
entity FULL_ADDER is
  port( A, B, CIN  : in  BIT ;
        SUM, COUT  : out BIT ) ;
end;
architecture DATAFLOW of FULL_ADDER is
  signal S : BIT ;
begin
  S    <= A xor B ;
  SUM  <= S xor CIN;
  COUT <= (A and B) or (S and CIN);
end;
```

The entity describes the boundary of the design unit. Each port is assigned a direction (in this example, `in` or `out`) and a data type (such as `BIT`).

The architecture describes the contents of the `FULL_ADDER` entity. The architecture starts with a declarative region; in this example, an internal signal `S` is defined.

The architecture content starts with `begin` and ends with `end`. The region between the `begin` and `end` is called the *dataflow* region. All statements in the dataflow region are executed concurrently, so the order of the statements is irrelevant.

An entity can have several architectures. A design can be modeled using different implementation methods, at different views, or at different levels of abstraction. Only one architecture is synthesized each time. By default, VHDL links the last defined architecture to the entity.

Use generic constants to control the structure of an entity. For example, in the following VHDL description, the <*N*> generic constant specifies the number of bits for the adder. The default

© 2021 All Rights Reserved.

value of the <*N*> generic constant is 4, until otherwise specified when you add an instance of the component.

```
entity ADDER is
  generic(N: INTEGER := 4 ) ;
  port( A, B : in  BIT_VECTOR (N-1 downto 0) ;
        CIN  : in  BIT ;
        SUM  : out BIT_VECTOR (N-1 downto 0) ;
        COUT : out BIT ) ;

end;
architecture GEN_FOUR of ADDER is
  signal TMP : BIT_VECTOR(N downto 0) ;
  component FULL_ADDER
    port (A, B, CIN : in BIT;
          SUM, COUT : out BIT) ;
  end component ;

begin
  TMP(0) <= CIN ;
  G :  for I in 0 to 3 generate
    FA : FULL_ADDER port map (
      A(I), B(I), TMP(I), SUM(I), TMP(I+1) ) ;
    end generate ;
  Cout <= TMP(N) ;

end
```

Objects

Data objects hold values of a specific type.

VHDL provides three standard classes of objects: constants, variables, and signals. It also supports three special object types: ports, generics, and loop variables. Ports and generics are associated with an entity declaration, and loop variables are associated with an iteration scheme.

Declarations list the data objects to use, state the object type for each data object, and optionally set the initial value of each object.

This section describes each of the standard and special object types.

Constants

A constant is an object initialized to a specific value when you create it, and that you cannot subsequently modify. You can use constant declarations in packages, entities, architectures, subprograms, blocks, and processes.

```
constant  YES       : BOOLEAN := TRUE ;
constant  CHAR7     : BIT_VECTOR(4 downto 0) := "00111" ;
constant  MSB       : INTEGER := 5 ;
```


© 2021 All Rights Reserved.

Variables

Variables hold temporary data. Declare variables only in a process or a subprogram.

A variable declaration must include a type. Its value can be either a range constraint or an initial value. By default, the initial value is the lowest (left-most) value of the range for that variable type.

```
variable  STAND,BROKE  : BIT ;
variable  TEMP         : BIT_VECTOR(8 downto 0) ;
variable  DELAY        : INTEGER range 0 to 15 := 0;
```

Signals

Signal declarations create new signals (wires) of a specified type. Signals can communicate between processes, or they can synchronize processes. Declare signals in package declarations (global signals), entities, architectures, and blocks. You can also use, but not declare, signals in processes and subprograms.

```
signal  BEEP       : BIT := '0' ;
signal  RESULT     : INTEGER range 0 to 1023 ;
signal  COUNT      : STD_LOGIC_VECTOR(7 downto 0);
```

Ports

Ports specify the interface channels of an entity. Ports are signals that communicate with the outside world.

As in standard object types, port declarations must specify a type, and can assign an initial value.

Each port is associated with a mode or direction, either `in`, `out`, `buffer`, or `inout`.

- `In` ports can be read but not assigned a value.
- `Out` ports can be assigned a value but not read.
- `Buffe`r and `inout` ports can be read and assigned a value.

```
entity ADDER is
   port( A, B : in  BIT_VECTOR (3 downto 0) ;
   CIN        : in  BIT ;
   SUM        : out BIT_VECTOR (3 downto 0) ;
   COUT       : out BIT ) ;
end ADDER ;
```

Port mapping rules have changed in IEEE 1076-2002. The following table shows the new port mapping rules specified in IEEE 1076-2002 LRM (section 1.1.12):

© 2021 All Rights Reserved.

Formal Port	Actual Port
IN	IN, INOUT, BUFFER
OUT	OUT, INOUT, BUFFER
INOUT	INOUT, BUFFER
BUFFER	OUT, INOUT, BUFFER

IEEE 1076-1993 LRM also specified that a BUFFER port can have at most one source, and any actual associated with a formal BUFFER port can have at most one source. These restrictions have been removed in IEEE 1076-2002.

The following example shows the kinds of port mapping allowed in IEEE 1076-2002:

```
-- File: test.vhd
library ieee;
use ieee.std_logic_1164.all;
entity FULLADD is
  port (X: in std_logic;
        SUM : out std_logic);
end FULLADD;

architecture STRUCTURAL of FULLADD is
  component HALFADD
    port (a : in std_logic;
          sum : buffer std_logic);
  end component;
  begin
  U2: HALFADD port map (a => X, sum => SUM);
end STRUCTURAL;
```

Compiling this example without `-v2008` or `-v200x` option results in a BUFFUP error.

Generics

Generics are defined in an entity that controls the structure or behavior of a design.

© 2021 All Rights Reserved.

Generic constants can be used to control the structure of an entity. For example, in the following VHDL description, the <*N*> generic constant specifies the number of bits for the adder.

```
entity ADDER is
    generic(N: INTEGER := 4 ) ;
    port( A, B       : in  BIT_VECTOR (N-1 downto 0) ;
          CIN        : in  BIT ;
          SUM        : out BIT_VECTOR (N-1 downto 0) ;
          COUT       : out BIT ) ;
   end ADDER ;
```

<*N*> is defined as a generic constant with an initial value of 4. You can use a generic map statement to overwrite the value of <*N*> when you add an instance of the component.

For example, if you defined ADDER as a cell in your design, you can use the following statement to add an instance of an 8-bit adder:

```
U1: ADDER generic map(N=>8)
          port map( IN1, IN2, RESULT, CARRY);
```

Loop Variables

Loop variables are special objects that you do not have to declare. You can use them in `loop` statements or `generate` statements within a `for` loop.

```
G :  for I in 0 to 3 generate
FA : FULL_ADDER port map (
                A(I), B(I), TMP(I), SUM(I), TMP(I+1) ) ;
end generate ;
```

In this example, the `I` variable is an integer with a range from 0 to 3. You can use a loop variable only inside the loop, and you cannot assign a value to the loop variable.

Data Types

All data object definitions in VHDL must include a data type. You can either use predefined data types, or define your own data types.

IEEE 1076 Predefined Types

IEEE 1076-1987 predefined a set of data types in the standard VHDL packages. You can synthesize the following standard data types:

```
type boolean is (false,true);
type bit is ('0', '1');
type character is (
     nul, soh, stx, etx, eot, enq, ack, bel,
     bs,  ht,  lf,  vt,  ff,  cr,  so,  si,
```


© 2021 All Rights Reserved.

```
     dle, dc1, dc2, dc3, dc4, nak, syn, etb,
     can, em,  sub, esc, fsp, gsp, rsp, usp,
     ' ', '!', '"', '#', '$', '%', '&', '''',
     '(', ')', '*', '+', ',', '-', '.', '/',
     '0', '1', '2', '3', '4', '5', '6', '7',
     '8', '9', ':', ';', '<', '=', '>', '?',
     '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
     'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
     'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
     'X', 'Y', 'Z', '[', '\', ']', '^ ' '_',
     '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
     'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
     'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
     'x', 'y', 'z', '{', '|', '}', '~', del);

type string is array (positive range <>) of character;
type integer is range -2147483648 to 2147483647;
subtype natural is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
type bit_vector is array (natural range <>) of bit;
```

HDL Compiler treats `character` and `string` as an enumeration type, and `string` as an array. See Modeling Styles on page 571 for restrictions on how HDL Compiler synthesis uses these types of objects.

IEEE 1164 Predefined Types

The IEEE 1164 standard defines a nine-valued logic type for simulation. The meaning of the nine values are:

- ’U’ —Uninitialized
- ’X’—Forcing Unknown
- ’0’ —Forcing Low
- ’1’ —Forcing High
- ’Z’—High Impedance
- ’W’—Weak Unknown
- ’L’—Weak Low
- ’H’—Weak High
- ’-’ —Don’t Care

HDL Compiler supports these types, but with restrictions. Among the nine values, only '0', '1', 'X', '-', and 'Z' have well-defined meaning for synthesis.

Useful data types in the package include:

© 2021 All Rights Reserved.

```
type Std_ULogic is ('U','X','0','1','Z','W','L','H','-');
type Std_ULogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic is resolved std_ulogic;
type std_logic_vector is (natural range <>) of std_logic;
SUBTYPE X01     IS resolved std_ulogic RANGE 'X' TO '1';
SUBTYPE X01Z    IS resolved std_ulogic RANGE 'X' TO 'Z';
SUBTYPE UX01    IS resolved std_ulogic RANGE 'U' TO '1';
SUBTYPE UX01Z   IS resolved std_ulogic RANGE 'U' TO 'Z';
```

To use the types defined in the STD_LOGIC_1164 package, your VHDL design must include the following statements:

```
library IEEE ;
use IEEE.STD_LOGIC_1164.all ;
```

IEEE 1076.3 Predefined Types

The IEEE 1076.3 standard defines SIGNED and UNSIGNED data types.

A data object of SIGNED or UNSIGNED data type is treated as a bit_vector (or std_logic_vector), but it can be used in arithmetic computation.

```
type UNSIGNED is array (NATURAL range <>) of BIT/STD_LOGIC;
type SIGNED is array (NATURAL range <>) of BIT/STD_LOGIC;
```

The standard synthesis packages (1076.3) include two VHDL packages: NUMERIC_BIT and NUMERIC_STD. The NUMERIC_BIT package is based on the BIT type, and the NUMERIC_STD package is based on the STD_LOGIC type.

Both packages interpret the UNSIGNED type as an unsigned binary integer, and the SIGNED type as a two's complement binary integer.

Both packages are compiled into the IEEE library. Because the numerical array types in both packages are defined using the same name, use only one package at a time.

To use the definitions and functions of the NUMERIC_BIT package, your VHDL design must include the following statements:

```
library IEEE ;
use IEEE.NUMERIC_BIT.all ;
```

To use the definitions and functions of the NUMERIC_STD package, your VHDL design must include the following statements:

```
library IEEE ;
use IEEE.STD_LOGIC.all ;
use IEEE.NUMERIC_STD.all ;
```

Enumeration Types

An enumeration type is defined by listing all possible values of that type.

© 2021 All Rights Reserved.

You define all values of an enumeration type. These values can be either identifiers or single-character literals.

- An identifier is a name, such as `white`, `monday`, or `lily`.
- Character literals are single characters enclosed in quotes, such as 'X', '0', or '1'.

```
type COLOR is (RED, ORANGE, YELLOW, GREEN, BLUE, PURPLE);
type STD_ULOGIC is ('U','X','0','1','Z','W','L','H','-');
```

Integer Types

Integer types (or subtypes) are for mathematical integers.

All normal predefined mathematical functions, such as add, subtract, multiply, and divide, apply to integer types.

```
type MY_INTEGER is range -64 to 64;
type TWOS_COMPLEMENT_INTEGER is range -32768 to 32767;
```

Array Types

Array types group one or more elements of the same type together as a single object.

Array elements can be of any VHDL data type.

An array has one and only one index, whose value selects each element. The number of indexes (the dimension) of an array can be any positive number. The range of the index (or the *called index range*) determines the number of elements in the array, and their direction (low-to-high, or high-to-low). In VHDL, the index type can be either integer or enumeration.

For example, the following defines a 64-element array with type as `AR1`, in which the type of each element is `INTEGER`:

```
type AR1 is array (0 to 63) of INTEGER ;
```

An unconstrained array type is a type whose dimension is specified, but the exact range and direction of each dimension is left unspecified. This enables multiple subtypes to share a common base type.

For example, VHDL uses two predefined unconstrained array types: `BIT_VECTOR` and `STRING` types.

```
type BIT_VECTOR is array (NATURAL range <>) of BIT ;
type STRING is array (POSITIVE range <>) of CHARACTER ;
```

To use an unconstrained array type, you must specify the index constraint. For example:

© 2021 All Rights Reserved.

```
subtype B4 is BIT_VECTOR (3 downto 0) ;
variable V5 : BIT_VECTOR(4 downto 0) ;
```

Note that specifying the index or slice of an external name in the following manner is not supported, as per the LRM:

```
<<signal sel : BIT_VECTOR>>
```

Following ways are supported:

```
    <<signal sel : BIT_VECTOR(3 downto 0)>>(1 downto 0));
Or
    sel : in BIT_VECTOR(3 downto 0)
```

and

```
    <<signal sel : BIT_VECTOR(1 downto 0)>>
```

The subtype indication of an external name must be constrained.

HDL Compiler lets you use multiple dimensional arrays with arrays of arrays.

```
type TWO_D_AR is array(0 to 7, 0 to 3) of BIT ;
    constant TWO_D_ROM : TWO_D_AR :=
    ( ( '0', '0', '0', '1'),
      ( '0', '0', '1', '1'),
      ( '0', '1', '0', '1'),
      ( '0', '1', '0', '1'),
      ( '0', '0', '1', '1'),
      ( '1', '0', '0', '1'),
      ( '1', '1', '0', '1'),
      ( '1', '1', '1', '1') ) ;
X := TWO_D_ROM(2, 3) ;
```

Record Types

Record types group one or more elements of different types together as a single object. HDL Compiler can access each element of the record by its field name.

Record elements can include elements of any type, including *array* and *records*. The following is an example of a record type declaration:

```
type DATE_TYPE is (SUN, MON, TUE, WED, THR, FRI, SAT) ;
type HOLIDAY is
     record
       YEAR: INTEGER range 1900 to 1999 ;
       MONTH: INTEGER range 1 to 12 ;
       DAY: INTEGER range 1 to 31 ;
       DATE: DATE_TYPE ;
     end record ;
```

A selected name is used to reference an element of a record. A selected name consists of the name of the object, followed by a dot (.), followed by the name of the element.

© 2021 All Rights Reserved.

The type of a selected name is the type of its element.

For example:

```
signal S: HOLIDAY ;
variable T1: INTEGER range 1900 to 1999 ;
variable T2: DATE_TYPE ;
   T1 := S.YEAR ;
   T2 := S.DATE ;
```

Subtypes

VHDL provides subtypes, which are subsets of other types. Anywhere that you can add a type definition, you can also add a subtype definition.

A subtype is a subset of a previously defined type or subtype. You can compare overlapping subtypes of a given parent type, and assign them to each other.

Subtypes are a powerful way to ensure valid assignments and meaningful data handling in VHDL. Subtypes inherit all operators and subprograms defined for their parent types.

For example, `NATURAL` and `POSITIVE` are subtypes of `INTEGER`, and you can use them with any `INTEGER` function. HDL Compiler can add, multiply, compare and assign them to each other, if the values are within the appropriate range for the subtype.

The following are examples of subtype declarations:

```
subtype INT4 is INTEGER range 0 to 15 ;
subtype BIT_VECTOR6 is BIT_VECTOR(5 downto 0) ;
```

Operators

Predefined VHDL operators that you can synthesize using HDL Compiler meet one of three IEEE standards: IEEE 1076, IEEE 1164, or IEEE 1076.3.

IEEE 1076 Predefined Operators

VHDL provides six classes of operators. Each VHDL operator has a precedence level.

Table 15-1 lists the classes of the operators, from lowest to highest precedence.

© 2021 All Rights Reserved.

Table 15-1 Class Precedence for IEEE 1076 Predefined Operators

Class of Operator	Members of Operator Class
logic_operator	and, or, nand, nor, xor
relational_operator	=, /=, <, <=, >, >=
adding_operator	+, -, &
sign	+, -
multiplying_operator	*, /, mod, rem
miscellaneous_operator	**, abs, not

Logical Operators

The logical operators (`and`, `or`, `nand`, `nor`, `xor`, and `not`) accept operands of predefined type `BIT`, type `BOOLEAN`, or one-dimensional array type of `BIT` or `BOOLEAN`.

For the binary operators (`and`, `or`, `nand`, `nor`, and `xor`), the parent type of both operands must be the same. If the type of the operands is a one-dimensional array, then the operands must be arrays of the same length.

The binary operators designate the `bitwise` function on matching elements of the arrays, and the result is an array with the same index range as the left operand.

The unary operator (`not`) computes the bitwise negation of its operand. If the type of the operand is an array, then the result is an array with the same index range as the operand.

Relational Operators

The relational operators work on operands of the same type, and return a `BOOLEAN` value.

- The equality (`=`) and inequality (`/=`) operators are defined for all types.
- The ordering (`<`, `=`, `>`, `>=`) operators are defined for all enumerated types, integer types, and one-dimensional arrays of enumeration or integer types. The internal order of a type's values determines the result of the ordering operators. Integer values are ordered from negative infinity to positive infinity.

© 2021 All Rights Reserved.

Adding Operators

Adding operators include adding `+`, `-`, and `&`.

The arithmetic operators (`+` and`-`) work on `INTEGER` operands.

The concatenation operator (`&`) groups element of the same type into an array of that type. Each operand of `&` can be an array, or an element of an array.

Unary Operators

All three unary operators take an object of numeric type, and return the same type.

- The unary plus operator (`+`) returns the value of its operand.
- The unary minus operator (`-`) computes the arithmetic negation of its operand.
- The unary absolute operator (`abs`} returns the absolute value of its operand.

Multiplying Operators

The multiplying operators (`**`, `*`, `/`, `mod`, and `rem`) work on `INTEGER` types. HDL Compiler restricts the supported values for the operands of the multiplying operators:

- For exponentiation (`**`), the left operand must be a power of 2.
- For division (`/`), modulus (`mod`), and remainder (`rem`) operations, the right operand must be a power of 2.
- Multiplication (`*`) has no limitation.

IEEE 1164 Predefined Operators

The logical operators (`and`, `or`, `nand`, `nor`, `xor`, and `not`) are overloaded in the `STD_LOGIC_1164` package for `std_logic` and `std_ulogic` types, and for `std_logic_vector` and `std_ulogic_vector` array types.

IEEE 1076.3 Predefined Operators

The base type of `SIGNED` and `UNSIGNED` is either a `BIT` type or `STD_LOGIC` type. You can apply any operators that work on `BIT` type or `STD_LOGIC` type to these two data types.

Except for the exponentiation (`**`) operation, all adding operators and multiplying operators also work on `SIGNED` and `UNSIGNED` data types.

© 2021 All Rights Reserved.

Operator Overloading

The predefined VHDL operators work only on data types that VHDL defines for the specific operator. To use an operator on data types that are not defined for that operator, use operator overloading. For example, in HDL Compiler, the + operator is overloaded so it can add two operands with `bit_vector` types and return a result of integer type.

```
architecture op_ov5 of op_ov5 is
  signal in1, in2 : bit_vector ( 0 to 2);
  signal out1 : integer;
function v2i (in1 : bit_vector(0 to 2)) return positive is
begin
  case in1 is
    when "001" => return 1;
    when "010" => return 2;
    when "011" => return 3;
    when "100" => return 4;
    when "101" => return 5;
    when "110" => return 6;
    when "111" => return 7;
    when "000" => return 8;
  end case;
end;
function "+" (in1,in2: bit_vector(0 to 2)) return integer is
begin
  return v2i(in1) + v2i(in2);
end;

begin
  out1 <= in1 + in2;
end op_ov5;
```

Operands

In an expression, the operator uses the operands to compute the value of the expression. Operands can themselves be expressions.

The types of operands you can use in an expression include literals, identifiers, indexed names, slice names, attribute names, aggregates, qualified expressions, type conversions, and function calls.

Literals

Literals are fixed (constant) values that you cannot change. HDL Compiler can synthesize numeric, character, enumeration, and string literals in your VHDL design.

© 2021 All Rights Reserved.

Numeric Literals

Numeric literals are constant integer values, which can be either *decimal* or *based*. The numeric literal can be separated by underscores.

- A decimal literal is written in base 10.
- A based literal can be written in any base from 2 to 16, as a base number, a pound sign (#), the value in the given base, and another # sign. The following example is equivalent to the decimal number 27:

  ```
  4#0_123#
  ```

Character Literals

Character literals are single characters enclosed in single quotation marks. For example:

```
'Z'
```

Enumeration Literals

Enumeration literals list (enumerate) all possible values of a specified data type. The two types of enumeration literals are characters and identifiers. The following example uses enumeration identifiers:

```
type SOME_ENUM is (ENUM_ID_1, ENUM_ID_2, ENUM_ID_3);
```

String Literals

String literals represent one-dimensional arrays of characters. The two types of strings are *character strings* and *bit strings*.

- Character strings are a sequence of characters in double quotes, such as:

```
"DEMO"
"10101"
```

- Bit strings are similar to character strings, but represent binary, octal, or hexadecimal values, such as:

```
B"11001"
O"277"
X"4C"
```

Identifiers

Identifiers are the names for constants, variables, signals, entities, ports, subprograms, or parameter declarations.

© 2021 All Rights Reserved.

Names must begin with a letter (`a` to `z`), followed by letters, underscores, or digits. However, underscore (_) cannot be the last character of an identifier.

VHDL identifiers are not case-sensitive—that is, `name`, `Name`, and `NAME` are all the same identifier.

Some identifiers, such as `entity`, `port`, `is`, and `end`, are reserved words. Each reserved word has a fixed meaning, so you cannot use a reserved word for other purposes in VHDL.

Indexed Names

Indexed names identify elements of array objects or signals. The syntax of an indexed name is:

```
<identifier> (<expression>)
```

where `<identifier>` is an array-type signal or variable, and `<expression>` returns a value within the array's index range. For example:

```
Data_Array(ADDR)
```

Slice Names

Slice names identify a sequence of elements of array objects. The slice direction can be either `to` or `downto`. However, the slice direction must be consistent with the direction of the identifier's declaration—for example:

```
bitvec(8 downto 2)
```

Attribute Names

Attributes are data attached to VHDL objects. VHDL attributes use variables or signals of a selected type, and these return values.

```
subtype IDX_RANGE is INTEGER range 10 downto 0 ;
variable VEC1: BIT_VECTOR (IDX_RANGE) ;
VEC1'left -- equal to IDX_RANGE'left == 10
VEC1'right -- equal to IDX_RANGE'right == 0
VEC1'high -- equal to IDX_RANGE'high == 10
VEC1'low -- equal to IDX_RANGE'low == 0
VEC1'range -- equal to IDX_RANGE == 10 downto 0
VEC1'reverse_range -- equal to == 0 to 10
VEC1'length -- 11
```

The `event` and `stable` attributes are associated with signals. These attributes indicate whether the signal value changes at the current simulation time. However, in HDL Compiler synthesis, you can use a VHDL `event` or `stable` attribute only to specify a leading or falling clock edge. For example:

© 2021 All Rights Reserved.

```
signal'event and signal='1' -- leading edge of signal
signal'event and signal='0' -- falling edge of signal
NOT signal'stable and signal='1' -- leading edge of signal
```

You can use the signal edge expression only as a condition within the `wait` and `if` statements.

Aggregates

Aggregates combine values into composite values of either *array* or *record* type. You can use aggregates to assign values to array type or record type objects during the initial declaration, or in an assignment statement.

You can use either *positional* or *named* association to specify the index of an element.

- If you use positional association, then HDL Compiler assigns a value to each element in order, by calculating the expression for that element.
- If you use name association, then HDL Compiler explicitly makes the correspondence by naming each element with a value.

For example:

```
type Color_List  is (Red, Orange, Blue, White) ;
type Color_Array is array (Color_List) of BIT_VECTOR
  (1 downto 0) ;
variable X : Color_array ;
X := ("00", "01", "10", "11") ; -- positional association
X := (Red => "00",  Blue => "10", Orange => "01", White => "11") ;
```

Qualified Expressions

Qualified expressions are expressions or aggregates that resolve ambiguities in operand types. This type of expression explicitly states the type or the subtype of an operand.

The syntax of a qualified expression is:

```
<type_name>'(<expression>)
```

where `<type_name>` is the name of a defined operand type, and `<expression>` evaluates to an appropriate type.

For example:

```
signed'("0000")
```


© 2021 All Rights Reserved.

Type Conversions

Type conversion explicitly converts data types to related data types. Type conversions change the type of expressions; qualified expressions only resolve the type of expressions.

Function Calls

Function calls transfer execution to a specified function body (which can have parameters), and retrieve values.

The syntax of a function call is:

```
<function_name> ([<parameter_name> =>] <expression> {, [<parameter_name> =>]
<expression>})
```

where `<function_name>` is the name of a defined function, the optional `<parameter_names>` are the names of formal parameters of the function, and `<expression>` returns a value that evaluates to an appropriate type for the parameter.

For example:

```
function FUNC(A, B, C : INTEGER) return BIT;
```

Sequential Statements

Sequential statements are statements within a process or a subprogram. VHDL uses the following sequential statements:

- Wait Statements
- Signal Assignment Statements
- Variable Assignment Statements
- If Statements
- Case Statements
- Loop Statements
- Next Statements
- Exit Statements
- Null Statements
- Assertion Statements
- Function and Procedure Calls

© 2021 All Rights Reserved.

Wait Statements

HDL Compiler can synthesize `wait` statements only if they include edge-detecting attributes (`event` and `stable`). This restriction assures correct compilation of a VHDL program into a logic circuit.

HDL Compiler synthesizes the following three statements the same way:

```
wait until CLK = '1';
wait until CLK'event and CLK='1';
wait until not CLK'stable and CLK='1';
```

You can also include a conditional expression in the `wait` statement. For example, the following means “wait for the falling edge of the BCLK signal, and until the value of START is 1.”

```
wait until BCLK = '0' and start = '1';
wait until BCLK'event and BCLK='0' and start = '1';
wait until not BCLK'stable and BCLK='0' and start = '1';
```

HDL Compiler can synthesize multiple `wait` statements in a process. However, multiple wait statements within a single process must not specify different clock sources, as this will produce an error similar to the following:

```
ERROR: design.vhd(line 49): Model error: More than one clock source (Done, CLK) in
a module.
```

Signal Assignment Statements

VHDL associates signals or ports with time. A signal assignment changes the value of a signal or a port.

In VHDL, when you assign a value to a signal, the value of a signal is not updated immediately. Instead, VHDL schedules it to update at a delta time (or a specified delay time).

The <= operator assigns values to signals or ports.

```
signal S1,S2: BIT;
signal S3,S4,S5: STD_LOGIC_VECTOR(7 downto 0);
process(S2,S3,S4)
begin
  S1 <= not S2;
  S5 <= S3 or S4;
end process ;
```

In a process, the value of the signal changes after a `wait` statement, or after the complete process is executed.

© 2021 All Rights Reserved.

Variable Assignment Statements

A variable assignment changes the value of a variable. The value of a variable updates immediately after the assignment.

You can declare variables only in a process or a subprogram. VHDL uses variables for temporary storage; other concurrent statements cannot detect these variables.

The `:=` operator assigns values to variables.

The following is an example of using variable assignment statements:

```
subtype INT16 is INTEGER range 0 to 65535 ;
signal S1, S2: INT16 ;
signal GT: BOOLEAN ;
  process(S1, S2)
    variable  A, B : INT16 ;
    constant  C : INT16 := 100 ;
  begin
        A  := S1 + 1 ;
        B  := S2 * 2 - C ;
        GT <= A > B ;
end process ;
```

If Statements

An `if` statement creates a branch in the execution flow. Depending on how it evaluates the conditions, HDL Compiler executes one or none of the enclosed sequences of statements.

The following example, which represents a multiplexer function, shows how to use `if` statements:

```
signal IN1, IN2, OU   : STD_LOGIC_VECTOR(0 to 3);
signal SEL : STD_LOGIC_VECTOR(1 downto 0);

MUX : process (SEL, IN1, IN2)
begin
    if SEL = "00" then
        OU <= IN1;
    elsif SEL = "01" then
        OU <= IN2;
    elsif SEL = "10" then
        OU <= IN1 or IN2;
    else
        OU <= "0000";
    end if;
end process ;
```

Case Statements

A `case` statement is another form of conditional control that VHDL provides. It is a multi-way branch based on the value of a control expression.

© 2021 All Rights Reserved.

A `case` statement is similar to an `if` statement, in that a condition is used to choose among a number of collections of statements. It is different, however, in that HDL Compiler chooses `case` statements based on the value of an expression.

The following is an example of using `case` statements:

```
signal S1         : INTEGER range 0 to 7 ;
signal I1, I2, I3 : BIT ;
select_process :
process(S1, I1, I2, I3)
begin
  case S1 is
    when 0 | 2 =>
      OU <= '0' ;
    when 1 =>
      OU <= I1 ;
    when 3 to 5 =>
      OU <= I2 ;
    when others =>
      OU <= I3 ;
  end case ;
end process select_process ;
```

Loop Statements

A `loop` statement contains a loop body of statements to execute repeatedly, zero or more times.

The following example repeatedly executes the same statements enclosed in the loop, starting when the value of `Sec` is 0, until the value of `Sec` becomes 60.

```
process
    variable Sec : INTEGER range 0 to 60 ;
begin
        L1 : loop
          wait until CLK'event and CLK = '1' ;
          Sec := Sec + 1 ;
          exit L1 when (Sec = 60) ;
          end loop L1 ;
end process;
```

You can associate a loop statement with an iteration scheme, either `for` or `while`.

Using `For` Loops

A `for` loop is a sequential statement in a process that iterates over a number of values. For example:

```
for I in 1 to 10 loop
     I_SQUARED(I) := I * I ;
end loop ;
```


© 2021 All Rights Reserved.

In this example, you do not need to declare the *I* loop index, and you cannot reassign a value within the loop. By default, the *I* index is an integer, and takes on the values 1 to 10. This expression returns the squares of all numbers from 1 to 10 (1, 4, 9, 16, 25, 36, 49, 64, 81, 100).

You can use a `downto` clause in a `for` loop to create a descending range. For example:

```
for I in 10 downto 1 loop
     I_SQUARED(I) := I * I ;
end loop ;
```

This example returns the same values as the preceding example, in reverse order (100, 81, 64, 49, 36, 25, 16, 9, 4, 1).

Using `While` Loops

Another iteration scheme is the `while` loop, where the statement is executed by first evaluating the `condition`.

If the result is TRUE, then HDL Compiler repeatedly executes the statements in the loop body, until either the `condition` evaluates to FALSE, or an `exit` statement transfers control out of the loop body.

The following is an example of using a WHILE loop.

```
signal S      : INTEGER range 0 to 10 ;

process
   variable I : INTEGER range 0 to 10 ;
begin
   wait until (CLK'event and CLK = '0');
   I := 0 ;
while (I < 10) loop
   wait until (CLK'event and CLK = '0');
   S <= I ;
   I := I + 1 ;
   end loop ;
end process ;
```

Each `while` loop statement must find at least one `wait` statement in each circular execution path. Otherwise, HDL Compiler gives an error.

Next Statements

The `next` statement skips execution to the next iteration of an enclosing `loop` statement. If you nest loops in VHDL, then you can assign a label to each loop.

© 2021 All Rights Reserved.

The `next` statement can include one or more `loop_label` parameters to specify particular loop labels to skip. If you do not use the `loop_label` parameter, then the `next` statement applies to the innermost enclosing loop.

The following example shows how to use the `next` statement.

```
L1 : while I < 10 loop
  L2 : while J < 20 loop
    Sum(I,J) := Sum(I,J) + DataIn;
    next L1 when Sum(I,J)<0;
  end loop L2 ;
end loop L1 ;
```

Exit Statements

The `exit` statement skips the remainder of an enclosing loop statement, and continues with the statement immediately after the exited loop.

As in the `next` statement, you can specify a particular enclosing loop to exit.

```
loop
B := B - A ;
exit when B > A ;
end;
```

Null Statements

The `null` statement in VHDL performs no action. HDL Compiler ignores `null` statements, and proceeds to the next statement.

You can use this statement to explicitly state not to perform any action when a condition is true.

The following is an example of using a `null` statement:

```
variable SEL : INTEGER range 0 to 31;
variable V   : INTEGER range 0 to 31;

case SEL is
  when 0 to 15  => V := SEL;
  when others   => null;
end case;
```

Assertion Statements

The `assertion` statement indicates that you expect a specified condition to be true. If the condition is false, then this statement reports a diagnostic message.

HDL Compiler ignores `assertion` statements.

© 2021 All Rights Reserved.

Function and Procedure Calls

In behavioral design descriptions, subprograms are a convenient way to implement frequently used functions. A subprogram consists of sequential statements, just as a process does.

VHDL uses two types of subprograms:

- A procedure returns multiple values.
- A function returns a single value.

Procedure calls invoke procedures to execute in a process. You must declare the procedure in a declaration section of your VHDL design, before you can call the procedure. These procedures act as extensions to the behavior of the process statement.

Concurrent Statements

An architecture that describes the functionality of an entity consists of one or more concurrent statements. Each concurrent statement defines a unit of computation that reads signals, performs computations based on their value, and assigns computed values to signals.

Concurrent statements are executed in parallel. These statements can appear in any order in the architecture.

The types of concurrent statements in VHDL are:

- Process Statements
- Concurrent Signal Assignments
- Conditional Signal Assignments
- Selective Signal Assignments
- Block Statements
- Component Instantiations
- Generate Statements
- Functions and Procedures

© 2021 All Rights Reserved.

Process Statements

A process consists of a set of sequential statements, but processes are themselves concurrent statements. To communicate with the rest of a design, a process reads values from, or writes values to, signals or ports declared outside the process.

You can use `process` statements to describe a combinational circuit. The following example of using a `process` statement describes a 2-to-1 multiplexer.

```
signal IN1, IN2, OUT   : STD_LOGIC_VECTOR(0 to 3);
signal SEL : STD_LOGIC;

process (SEL, IN1, IN2)
begin
  if SEL = '0' then
    OUT <= IN1;
  else
    OUT <= IN2;
  end if;
end process ;
```

You can also use `process` statements to describe a sequential circuit. In the following example, on each 0-to-1 transition of the CLK signal, the process gates the output of the multiplexer into the OUT register.

```
signal CLK: BIT ;
signal IN1, IN2, OUT   : STD_LOGIC_VECTOR(0 to 3);
signal SEL : STD_LOGIC;

process
begin
  wait until (CLK'event and CLK = '1') ;
  if SEL = '0' then
    OUT <= IN1;
  else
    OUT <= IN2;
  end if;
end process ;
```

When a `process` statement assigns a value to a signal, the value defines a driver of the signal. If more than one `process` statement, or other concurrent statements, assign a value to a signal, then that signal has multiple drivers.

In VHDL, if the signal is not a resolved data type, then you must define a resolution function for the signal:

```
  signal OUT, IN1, IN2 : STD_LOGIC;
  signal SEL1,SEL2: BIT;
P1: process (SEL1,IN1)
  begin
    if SEL1 = '0' then
      OUT <= IN1;
    end if;
  end process;
P2: process (SEL2,IN2)
  begin
```


© 2021 All Rights Reserved.

```
    if SEL2 = '0' then
      OUT <= IN2;
    end if;
  end process;
```

In this example, the OUT signal has multiple drivers. Because `STD_LOGIC` is a resolved data type, it allows multiple drivers. See Resolution Functions on page 558 for more information.

Concurrent Signal Assignments

A concurrent signal assignment statement represents an equivalent `process` statement that assigns values to signals.

The following example shows a concurrent signal assignment within a VHDL architecture.

```
architecture Description1 of EXAMPLE is
signal I1, I2, AND_OUT: BIT ;
begin
  AND_OUT <= I1 and I2;
end Description1 ;
```

This example is equivalent to the following:

```
architecture Description2 of EXAMPLE is
signal I1, I2, AND_OUT: BIT ;
begin
  process (I1, I2)
  begin
    AND_OUT <= I1 and I2;
  end process ;
end Description2;
```

Conditional Signal Assignments

A conditional signal assignment is a concurrent statement that has one target, but it can use more than one expression. If one condition evaluates to TRUE, then HDL Compiler uses the corresponding expression; otherwise, HDL Compiler uses the remaining expression. HDL Compiler uses one and only one of the expressions at a time.

You can transform any conditional signal assignment into an equivalent `process` statement.

The following is an example of using conditional statements:

```
architecture Description1 of EXAMPLE is
  signal A, B, Z: BIT ;
  signal X: INTEGER ;
  constant C: BIT :='0' ;
begin
  Z <= A when ( X > 10 ) else
       B when ( X > 5  ) else
       C ;
end Description1 ;
```


© 2021 All Rights Reserved.

This example is equivalent to the following process statement:

```
architecture Description2 of EXAMPLE is
  signal A, B, Z: BIT ;
  signal X: INTEGER ;
  constant C: BIT :='0' ;
begin
  process (A, B, X)
  begin
    if ( X > 10 ) then
      Z <= A ;
    elsif ( X > 5 ) then
      Z <= B ;
    else
      Z <= C ;
    end if ;
  end process ;
end Description1 ;
```

Selective Signal Assignments

A selective signal assignment can have only one target, and can have only one `with` expression. HDL Compiler tests this value for a match, in a manner similar to the `case` statement. A selective signal assignment runs whenever the selected signal changes in any way.

You can describe any selected signal assignment using a `process` statement that contains a `case` statement.

The following is an example of a selective signal assignment:

```
architecture Description1 of EXAMPLE is
  signal A, B, Z: BIT ;
  constant C: BIT :='0' ;
begin
  with SEL select
    Z <= A when 0 | 1 | 2 ,
         B when 3 to 10,
         C when others ;
end Description1;
```

This example is functionally equivalent to the following:

```
signal A, B, Z: BIT ;
  constant C: BIT :='0' ;
process (SEL, A, B)
  begin
    case SEL is
      when 0 | 1 | 2 =>
        Z <= A ;
      when 3 to 10 =>
        Z <= B ;
      when others =>
        Z <= C ;
    end case ;
  end process ;
```


© 2021 All Rights Reserved.

Block Statements

A `block` statement consists of a set of concurrent statements. Blocks organize a set of concurrent statements (procedure calls) hierarchically.

```
architecture BHV of EXAMPLE is
  signal IN1         : INTEGER ;
begin
  B1                 : block
    signal S         : BIT ;
  begin
    B1_1 : block
      signal S       : INTEGER ;
    begin
      S <= IN1 ;
    end block B1_1 ;
  end block B1 ;
end BHV ;
```

Component Instantiations

Component instantiations describe a design hierarchy. At the point of instantiation, only the external view of the component (the names, types, and directions of its ports) is visible; signals internal to the component are not visible. The component instantiation statement adds an instance and connects components, to form a netlist of instances of a design.

```
architecture INST of STRUCTURE is
  component XOR_gate
    port (I0, I1 : in STD_LOGIC; O : out STD_LOGIC) ;
  end component ;
  component AND2_gate
    port( I0, I1 : in  STD_LOGIC; O : out STD_LOGIC) ;
  end component ;

  signal A, B, Tmp1, Tmp2 : STD_LOGIC ;
  -- use default binding
begin
  U1: XOR_gate
    port map( A, B, Tmp1); -- Positional Association
  U2: AND2_gate
    port map( I0=>A, I1=>B, O=>Tmp2);
      -- Named Association
  end INST ;
```

You can use a component instantiation statement to add an instance of any component defined in a VHDL architecture. You can also pass generic constants to the lower-level components. For example:

```
library cs35 ;
use cs35.all ;
architecture instance of instance is
  component instance1
    generic ( N : integer; Y: integer);
    port ( in1,in2 : bit_vector( N downto Y);
            output : out bit_vector(N downto Y)
         );
```


© 2021 All Rights Reserved.

```
  end component;

begin
  inst1: instance1 generic map ( N => 7, Y=>  4)
    port map(in1( 7 downto 4),in2( 7 downto 4),
      output(7 downto 4));
  inst2: instance1 generic map ( N => 3, Y=>  0)
    port map(in1( 3 downto 0),in2( 3 downto 0),
      output( 3 downto 0));
end;
```

Using Default Search Order

The preceding example has not yet defined the `instance1` component. You can use any of the methods described in this section to provide this information. When HDL Compiler finds a component instantiation, it uses the following sequence to search for the entity for the component:

- Searches for a design in the working library. If an entity matches the name and port properties of the component, then HDL Compiler binds the component to the entity.
- Searches for the entities in your design library, as specified in the VHDL description (the `cs35` library in this example). If an entity matches the name and port properties of the component, then HDL Compiler uses the entity that it found.
- If HDL Compiler cannot find an entity that matches the component, then it reports a warning.

Generate Statements

The `generate` statement is a concurrent statement that describes replicated structures. You must define the statement in an architecture.

VHDL uses two methods of generation: `for` and `if`.

Using FOR Generation

The `for` method describes a regular structure. The following, example uses four one-bit full adders to construct a four-bit adder.

```
architecture GEN_FOR of FULL_ADDER_4 is
  signal X, Y, Z : STD_LOGIC_VECTOR(3 downto 0) ;
  signal Cout    : STD_LOGIC ;
  signal TMP     : STD_LOGIC_VECTOR(4 downto 0) ;
  component FULL_ADDER
    port ( A, B, C : in STD_LOGIC;
           S, Co   : out STD_LOGIC) ;
  end component ;
begin
```


© 2021 All Rights Reserved.

```
  TMP(0) <= '0' ;
  G : for I in 0 to 3 generate
      FA : FULL_ADDER port map (
           X(I), Y(I), TMP(I), Z(I), TMP(I+1) ) ;
      end generate ;
  Cout <= TMP(4) ;
end GEN_FOR ;
```

In a `generate` statement, you do not need to declare the loop variable. You can describe any concurrent statement inside a `generate` statement.

Using IF Generation

The `if` method generates a regular structure that has some irregularity.

Use a `for-generate` statement to define the width of a design, and a set of `if-generate` statements to define the beginning, middle, and ending set of connections.

The following example describes a four-bit full adder, consisting of three one-bit full adders, and one half-adder:

```
architecture GEN_FOR of FULL_ADDER_4 is
    signal X, Y, Z : STD_LOGIC_VECTOR(3 downto 0) ;
    signal Cout    : STD_LOGIC ;
    signal TMP     : STD_LOGIC_VECTOR(4 downto 1) ;
    component HALF_ADDER
        port ( A, B  : in STD_LOGIC;
                     S, Co : out STD_LOGIC) ;
    end component ;

    component FULL_ADDER
        port ( A, B, C : in STD_LOGIC;
                    S, Co   : out STD_LOGIC) ;
    end component ;

begin
    G0: for I in 0 to 3 generate
        G1: if I = 0 generate
            HA: HALF_ADDER port map (
                        X(I), Y(I),
                        Z(I), TMP(I+1) ) ;
        end generate ;
        G2: if I >= 1 and I <= 3 generate
            FA: FULL_ADDER port map (
                    X(I), Y(I), TMP(I),
                    Z(I), TMP(I+1) ) ;
            end generate ;
    end generate ;
    Cout <= TMP(4) ;
end GEN_FOR ;
```

Unlike the sequential `if` statement, the `if-generate` statement cannot use `else` or `elsif` branches.

© 2021 All Rights Reserved.

Subprograms

VHDL uses two types of subprograms: procedures and functions.

- A procedure runs as a statement, and can return one or more parameters, or no parameters.
- A function runs as an expression, and always returns just one parameter.

A special category of functions, called resolution functions, resolves multiple sources for a signal into a single value for that signal.

Functions and Procedures

Procedures can change the values of objects, if those objects are associated with the formal `in`, `out`, or `inout` mode parameters of the procedure.

Functions compute values, but unlike procedures, they do not change the value of any objects associated with the formal parameters of the function. A function must declare the type of value that the function returns.

Subprograms consist of subprogram declarations and subprogram bodies.

Subprogram declarations contain only interface information. The following are examples of subprogram declarations:

```
function Incrementer (Count: INTEGER) return INTEGER ;
    -- by default will be understood as:
    -- constant Count: in INTEGER
procedure Send ( signal CLK : in BIT ;
                                Data       : INTEGER ;
    -- by default will be understood as:
    -- constant Data : in INTEGER;
    signal Ack   : BIT ;
    signal Ready : out BIT ;
    signal Wire  : out INTEGER ) ;
```

Subprogram bodies describe the functionality of the subprograms. The following examples are subprogram (procedure and function) bodies.

```
function Incrementer (Count : INTEGER) return INTEGER is
variable TEMP: INTEGER ;
begin
    if (Count >= 255) then
        TEMP := 0 ;
    else
        TEMP := Count + 1 ;
    end if ;
    return(TEMP) ;
end Incrementer ;
```


© 2021 All Rights Reserved.

```
procedure Send ( signal CLK   : BIT ;
                                Data
                              : INTEGER ;
                                signal Ack   : BIT ;
                                signal Ready : out BIT ;
                                signal Wire  : out INTEGER ) is
begin
    wait until (CLK'event and CLK = '1' and Ack = '0') ;
    Wire  <= Data ;
    Ready <= '1' ;
    wait until (CLK'event and CLK = '1' and Ack = '1');
    Ready <= '0';
end Send ;
```

Note: When a function is called, the function call must terminate by executing a return statement, which determines the value that the function call returns.

Resolution Functions

In VHDL, more than one source can drive a signal, but you must use a `resolution` function to resolve the multiple sources into a single value for the signal.

A resolved signal is created if either a signal declaration or its subtype declaration includes a resolution function.

The following is an example of such a declaration:

```
signal NODE : WIRE_AND BIT ;
subtype RESOLVED_STD is WIRE_OR STD_ULOGIC ;
```

The first declaration uses the `WIRE_AND` resolution function to resolve the `NODE` signal. Each time an event effects the `NODE` signal, HDL Compiler calls the `WIRED_AND` function, and returns a `BIT`-type value.

The second declaration defines a resolved subtype named `RESOLVED_STD`. Signals declared using the `RESOLVED_STD` type are resolved signals.

For simulation, the resolution function can be any function, consisting of any VHDL program. But synthesis supports only a limited number of methods for handling resolution functions in hardware, such as a wired-or gate, a wired-and gate, or a tristate gate. In HDL Compiler, the resolution functions must be either wired-and, wired-or, or tristate connections. You must specify a synthesis directive to indicate which method to use.

For example, a synthesis directive is inserted in the `wired_and` function. HDL Compiler uses the `wire_and` gate to resolve the signal, regardless of the content of the function.

```
package pack_res_func is
    function wired_and ( input : bit_vector ) return bit ;
end ;
package body pack_res_func is
    function wired_and ( input : bit_vector )
```


© 2021 All Rights Reserved.

```
        return bit is
    -- pragma resolution_method wired_and
        begin
            for i in input'range loop
                if ( input (i) ='0' ) then
                    return '0' ;
                end if;
            end loop ;
                return '1' ;
        end ;
end ;

use work.pack_res_func.all;
entity res_func is
    port ( clk     : bit;
                    res_sig : out wired_and bit
            );
end ;

architecture res_func of res_func is
begin
    source_1 :
        process
        begin
            wait until clk'event and clk='1';
            res_sig <= '1' ;
            wait until clk'event and clk='1';
            res_sig <= '0' ;
        end process;
    source_2 :
        process
        begin
            wait until clk'event and clk='1';
            res_sig <= '1' ;
        end process;
end;
```

In the `STD_LOGIC_1164` package, the `STD_LOGIC` and `STD_LOGIC_VECTOR` data types are resolved types. A tristate function resolves multiple sources assigned to a signal of `STD_LOGIC` or `STD_LOGIC_VECTOR` type.

Packages

A VHDL package is a set of declarations and definitions for objects, functions, procedures, components, and attributes. This section describes the following types of packages:

- User-Defined Packages
- STANDARD and TEXTIO Packages
- IEEE STD_LOGIC_1164 Packages
- IEEE NUMERIC_BIT and NUMERIC_STD Packages

© 2021 All Rights Reserved.

Note: The package name for the Cadence `qtref` library was formerly `qtref.qtref`, but is now `qtref.qtref_pkg`. If your VHDL source code uses the old Cadence package name, you need to change it.

User-Defined Packages

A package consists of two parts: a package declaration section and a package body.

Package Declarations

The package declaration defines the interface for the package. For example:

```
library IEEE ;
use IEEE.NUMERIC_BIT.all ;
package WATCH_PKG is
   subtype MONTH_TYPE is INTEGER range 0 to 12 ;
   subtype DAY_TYPE   is INTEGER range 0 to 31 ;
   subtype BCD4_TYPE  is UNSIGNED(3 downto 0) ;
   subtype BCD5_TYPE  is UNSIGNED(4 downto 0) ;
   constant BCD5_1 : BCD5_TYPE := B"0_0001" ;
   constant BCD5_7 : BCD5_TYPE := B"0_0111" ;
     function BCD_INC (L : in BCD4_TYPE) return BCD5_TYPE ;
end WATCH_PKG ;
```

Package Bodies

A package body always has the same name as its corresponding package declaration, preceded by the reserved words `package body`.

The package body specifies the behavior of the package. Design and entities that use the package do not recognize the information in a package body. The package body inherits all declarations defined in the corresponding package declaration section.

The following example is a package body definition for the `WATCH_PKG` package:

```
package body WATCH_PKG is
   function BCD_INC (L : in BCD4_TYPE) return BCD5_TYPE is
      variable V, V1, V2 : BCD5_TYPE ;
   begin
      V1 := L + BCD5_1 ;
      V2 := L + BCD5_7 ;
      case V2(4) IS
         when '0' => V := V1 ;
         when '1' => V := V2 ;
      end case ;
      return (V) ;
   end BCD_INC ;
end WATCH_PKG ;
```


© 2021 All Rights Reserved.

The `use` clause adds functionality from a package into a VHDL description. In the preceding example, if you compiled the package into the `WATCH_LIB` library, then you need to include the following two lines, so the design recognizes the information in the package.

```
library WATCH_LIB ;
use WATCH_LIB.WATCH_PKG.all;
```

STANDARD and TEXTIO Packages

The `STANDARD` package (IEEE 1076-1987) defines useful data types, such as `INTEGER`, `BIT`, and `BOOLEAN` (see Operators on page 537).

- The `STANDARD` package is automatically visible, and does not require a `use` clause.
- The `TEXTIO` package defines types and operations, for communication with a standard programming environment. This `TEXTIO` package is used mostly for simulation. Synthesis does not require it.

IEEE STD_LOGIC_1164 Packages

The IEEE Standard Logic Package defines the following useful data types:

```
type Std_ULogic is ('U','X','0','1','Z','W','L','H','-');
type Std_ULogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic is resolved std_ulogic;
type std_logic_vector is (natural range <>) of std_logic;
SUBTYPE X01   IS resolved std_ulogic RANGE 'X' TO '1';
SUBTYPE X01Z  IS resolved std_ulogic RANGE 'X' TO 'Z';
SUBTYPE UX01  IS resolved std_ulogic RANGE 'U' TO '1';
SUBTYPE UX01Z IS resolved std_ulogic RANGE 'U' TO 'Z';
```

The logic operators, such as `and`, `nand`, `or`, `nor`, `xor`, and `not`, are overloaded to apply to the data types.

Table 15-2 describes useful functions that this type of package defines.

Table 15-2 I EEE STD_LOGIC_1164 Functions

Function	Description
`to_bit(<S>, <XMAP>)`	Converts a data object of STD_ULOGIC or STD_LOGIC type to BIT type. If the value of <S> is not '0' or '1', then this function maps it into '0' or '1' according to the function specified in <XMAP>.

© 2021 All Rights Reserved.

Function	Description
`to_StdULogic(<S>)`	Converts the value of the <S> signal from a BIT type to a data object of STD_ULOGIC type.
`rising_edge(<S>)`	Detects the leading edges of the <S> signal, which is a STD_ULOGIC type.
`falling_edge(<S>)`	Detects the falling edges of the <S> signal, which is a STD_ULOGIC type.

To use the definitions and functions of the `STD_LOGIC_1164` packages, your VHDL design must include the following statements.

```
library IEEE ;
use IEEE.STD_LOGIC_1164.all ;
```

IEEE NUMERIC_BIT and NUMERIC_STD Packages

The Standard Synthesis packages (IEEE 1076.3) include two VHDL packages: `NUMERIC_BIT` and `NUMERIC_STD`. Both packages are compiled into the IEEE library. The numerical array types in both packages use the same name, so you can use only one package at a time.

The `NUMERIC_BIT` package is based on the `BIT` type, and the `NUMERIC_STD` package is based on the `STD_LOGIC` type. Both packages interpret the `UNSIGNED` type as an unsigned binary integer, and the `SIGNED` type as a two's-complement binary integer.

This package defines two useful data types:

```
type UNSIGNED is array (NATURAL range <>) of BIT/STD_LOGIC;
type SIGNED is array (NATURAL range <>) of BIT/STD_LOGIC;
```

Arithmetic operators (such as `+`, `-`, and `*`), and comparison operators, are overloaded to apply the two data types.

To use the definitions and functions of the `NUMERIC_BIT` packages, you must include the following statements in your VHDL design:

```
library IEEE ;
use IEEE.NUMERIC_BIT.all ;
```

To use the definitions and functions of the `NUMERIC_STD` packages, you must include the following statements in your VHDL design:

© 2021 All Rights Reserved.

```
library IEEE ;
use IEEE.STD_LOGIC_1164.all ;
use IEEE.NUMERIC_STD.all
```

Table 15-3 describes useful functions that this type of package defines.

Table 15-3 IEEE NUMERIC_BIT and NUMERIC_STD Functions

Function	Description
`to_integer(<arg>)`	Converts the signed or unsigned `<arg>` object into an object of integer type.
`to_signed(<arg>, <n>)`	Converts the `<arg>` integer into a signed number `<n>` bits long.
`to_unsigned(<arg>, <n>)`	Converts the `<arg>` integer into an unsigned number `<n>` bits long.
`resize(<arg>, <n>)`	Changes the size of the unsigned or a signed number `<arg>` to a number `<n>` bits long.
`rising_edge(<S>)`	Detects the leading edges of the `<S>` signal, which is a BIT data type.
`falling_edge(<S>)`	Detects the falling edges of the `<S>` signal, which is a BIT data type.

Arithmetic Operations

HDL Compiler performs arithmetic operations on signed or unsigned numbers in your VHDL design, such as:

```
shift_left (<arg>, <n>)
shift_right (<arg>, <n>)
rotate_left (<arg>, <n>)
rotate_right (<arg>, <n>)
```

Configuration Declaration

Configuration declaration specifies binding information of component instances to design architectures, such as the architecture to use and/or generic constants of a specific instance, which could be inside a for-generate statement or if-generate statement. HDL Compiler supports a subset of configuration declaration commonly used in RTL synthesis. (See also Unsupported Constructs in Configuration Declaration on page 565.)

© 2021 All Rights Reserved.

Configuration declaration could be used to specify a particular architecture of an instance, for example `ripple_adder` or `carry_save_adder`.

```
CONFIGURATION cfg1 OF loopcntTop IS
     FOR rtl
     FOR loopcnt_inst1 : loopcnt
        USE ENTITY work.loopcnt(ripple_adder);
        END FOR;
        -- all unspecified instances use default binding
        END FOR;
     END CONFIGURATION;
```

Configuration declaration could associate a different value to a generic constant of an instance, for example the depth of a FIFO.

```
CONFIGURATION cfg1 OF loopcntTop IS
     FOR rtl
     FOR fifo_inst1, fifo_inst2 : fifo
        GENERIC MAP (DEPTH => 55);
     END FOR;
     FOR OTHERS: fifo
     -- applies to all other instances of fifo,
     -- except explicitly listed ones
     GENERIC MAP (DEPTH => 75);
     END FOR;
     END FOR;
     END CONFIGURATION;
```

```
CONFIGURATION cfg2 OF loopcntTop IS
FOR rtl
  FOR ALL : fifo
-- applies to all instances of fifo
GENERIC MAP (DEPTH => 55);
  END FOR;
END FOR;
END CONFIGURATION;
```

Instances in a for-generate statement could be bound to different architectures or different values of generic constants.

For example:

```
CONFIGURATION cfg1 OF loopcntTop IS
     FOR rtl
     FOR genAnd(0 to 3)
     -- applies to a for-generate block labeled 'genAnd'
     FOR add_inst : add
     USE ENTITY add(add_archA);
       END FOR;
       END FOR;
     FOR genAnd(8)
       FOR add_inst : add
       USE ENTITY add(add_archB);
       END FOR;
     END FOR;
     END FOR;
     END CONFIGURATION;
```

Nested configuration is supported.

© 2021 All Rights Reserved.

```
CONFIGURATION cfg1 OF loopcntTop IS
  FOR rtl
    FOR fifo_inst1, fifo_inst2 : fifo
      USE CONFIGURATION fifo_cfg1;
    END FOR;
  END FOR;
 END CONFIGURATION;
```

Restrictions in Configuration Declaration

- Generic constants of record types are not supported.
- A configuration identifier should be unique inside a library. It cannot be used by another entity or package.

Unsupported Constructs in Configuration Declaration

- Configuration specification inside a declarative portion of an architecture is not supported.
- The configuration declarative part, such as the USE clause and ATTRIBUTE clause, is not supported.

For example, the following is not supported.

```
LIBRARY mylib;
   CONFIGURATION cfg1 OF loopcntTop IS
    -- This is the declarative part --
    USE mylib.all; -- this is not supported
  FOR str
     FOR loopcnt_3 : loopcnt
    -- This is the component specification part --
    USE CONFIGURATION work.cfg3; -- this is supported
     END FOR;
  END FOR;
END CONFIGURATION;
```

- USE OPEN in component specification is not supported.

  For example, the following is not supported.

```
CONFIGURATION cfg1 OF loopcntTop IS
  FOR str
    FOR loopcnt_3 : loopcnt
      USE OPEN; -- this is not supported
    END FOR;
   END FOR;
END CONFIGURATION;
```

Note: PORT MAP aspect in component specification is not supported.

© 2021 All Rights Reserved.

VHDL Synthesis Directives

HDL Compiler provides several synthesis directives that you can use to do the following:

- Direct the process of translating behavioral (RTL) design descriptions into gate-level descriptions.
- Map functions to hardware components.
- Specify the asynchronous set/reset signal for cycle-fixed, multiple-wait descriptions.

For example, HDL Compiler supports the following synthesis directives:

```
-- pragma translate_off
-- pragma translate_on
-- pragma synthesis_off
-- pragma synthesis_on
-- pragma resolution_method wired_and
-- pragma resolution_method wired_or
-- pragma resolution_method three_state
```

In addition, Cadence provides its own versions of these synthesis directives, that accomplish similar purposes:

```
-- quickturn translate_off
-- quickturn translate_on
-- quickturn synthesis_off
-- quickturn synthesis_on
-- quickturn wired_and
-- quickturn wired_or
-- quickturn three_state
```

A synthesis directive resembles a regular comment starting with two hyphens (--). However, if the word following these characters is `pragma` or `synopsys`, then HDL Compiler interprets the remaining comment text as a directive, not a comment.

HDL Compiler uses three groups of synthesis directives:

- Synthesis Off and On Directives
- Resolution Function Directives
- Asynchronous Set/Reset Directives

Synthesis off and on Directives

These directives stop and start synthesis of a gate-level netlist from the VHDL source design. You can use `synthesis_off` and `synthesis_on` anywhere in your design description.

The syntax is:

© 2021 All Rights Reserved.

```
-- pragma synthesis_off
-- pragma synthesis_on
```

HDL Compiler cannot synthesize some VHDL statements, such as `FILE` declaration and file-access subroutines in the `TEXTIO` package. These statements, however, are very useful for simulation and debugging purposes.

This directive masks out those segments which HDL Compiler should not synthesize, but that a simulator should process. Translation into a gate-level description is enabled at the beginning of each VHDL source file.

The following example is a design description for a multiplexer and its test bench. A single architecture describes both processes. Synthesis does not need the test bench, so this example includes it in the VHDL code, but sets a synthesis_off directive, so HDL Compiler does not synthesize it.

```
library IEEE ;
use IEEE.NUMERIC_BIT.all ;
entity MUX is
   port (X0i : buffer UNSIGNED(3 downto 0) ;
         X1i : buffer UNSIGNED(3 downto 0) ;
         Z2i : buffer UNSIGNED(3 downto 0) ;
         SEL : buffer BIT ) ;
end MUX;
architecture BHV of MUX is
begin
   MUX2 :
   process(SEL,X0i,X1i)
   begin
      case SEL  is
         when '0' => Z2i <= X0i ;
         when '1' => Z2i <= X1i ;
      end case ;
   end process ;
-- pragma synthesis_off
   X0i <= "0011" ;
   X1i <= "0101" ;
   process
   begin
      SEL <= '0' ;
      wait for 10 ns ;
      SEL <= '1' ;
      wait for 10 ns ;
      assert (FALSE)
         report "MUXs : Testing Complete  "
         severity failure ;
   end process ;
-- pragma synthesis_on
end BHV ;
```


© 2021 All Rights Reserved.

Resolution Function Directives

These directives specify the resolution function for signals that have multiple drivers. HDL Compiler supports the following three methods:

```
--pragma tri_state
--pragma wire_or
--pragma wire_and
```

HDL Compiler uses the tristate buffer to implement the `tri_state` resolution function, `OR` gates to implement the `wire_or` function, and `AND` gates to implement the `wire_and` function.

The following example shows usage of a resolution function directive.

```
package pack_res_func is
   function wired_and ( input : bit_vector ) return bit ;
end ;

package body pack_res_func is
   function wired_and ( input : bit_vector )
      return bit is
   -- pragma resolution_method wired_and
   begin
      for i in input'range loop
         if ( input (i) ='0' ) then
            return '0' ;
            end if;
      end loop ;
            return '1' ;
   end ;
end ;

use work.pack_res_func.all;
entity res_func is
   port ( clk     : bit;
                res_sig : out wired_and bit
         );
end ;

architecture res_func of res_func is
begin
   source_1 :
      process
      begin
         wait until clk'event and clk='1';
         res_sig <= '1' ;
         wait until clk'event and clk='1';
         res_sig <= '0' ;
      end process;
   source_2 :
      process
      begin
         wait until clk'event and clk='1';
         res_sig <= '1' ;
      end process;
end;
```


© 2021 All Rights Reserved.

Asynchronous Set/Reset Directives

These directives specify the asynchronous set/reset signal for cycle-fixed design descriptions, containing multiple wait statements. The syntax is:

```
--pragma reset_high
--pragma reset_low
--pragma reset <reset_signal_name>
```

The asynchronous reset signal either sets or resets initial values to selected signals after the circuit starts or resets.

A process can include only one asynchronous reset signal. The data type of the asynchronous reset must be either `BIT` or `STD_ULOGIC`, and you can specify the asynchronous reset only in the declaration section of your VHDL design.

- The pragma reset_high directive sets the signal to active HIGH. This is the default for the VHDL reset function.
- The pragma reset_low directive sets the signal to active LOW.

The pragma reset `<reset_signal_name>` directive resets all signals, from their initial value, to the current value of the specified `<reset_signal_name>` signal.

Using these directives, you can instruct HDL Compiler to set or reset selected registers, after the asynchronous reset signal is set to `HIGH` or `LOW`.

The following VHDL description shows how to use an asynchronous reset directive, to specify an asynchronous reset signal (named `RESET`) at the algorithmic level. When `RESET` becomes `HIGH`, it sets all signals that have an initial value, to the current value of the `RESET` signal.

The following example sets the `CountOut` output signal to 0:

```
entity Counter is
     port ( CLOCK     : in     BIT ;
     RESET     : in     BIT ;
     CountOut  : buffer INTEGER range 0 to 15 := 0 ) ;
end Counter;

architecture Algorithm of Counter is
     -- pragma reset_high reset
begin
     process
     begin
     wait until CLOCK'event and CLOCK = '1' ;
     CountOut <= (CountOut + 1) mod 16 ;
     end process;
end Algorithm ;
```

The following is a functionally equivalent description:

```
architecture FSMD of Counter is
begin
```


© 2021 All Rights Reserved.

```
     process (RESET, CLOCK)
     begin
     if (RESET = '1') then
     CountOut <= 0;
     elsif (CLOCK'event and CLOCK = '1') then
     CountOut <= (CountOut + 1) mod 16 ;
     end if;
     end process ;
end FSMD ;
```

Enabling or Disabling Memory Synthesis

HDL Compiler automatically chooses to use the `hardmacro` memories for constructs for which they are suitable. In addition, HDL Compiler provides fine granular control of memory synthesis with two synthesis directives.

- `no_hardmacro`

  This directive disables memory synthesis for the specified signal. For example, consider a memory declared in the RTL as follows:

  ```
  reg [1024:9] mem [23:13];
  ```

  You can add the synthesis directive for this memory after the declaration as follows:

  ```
  //quickturn no_hardmacro mem
  ```

- `use_hardmacro`

  This directive forces the signal to be synthesized into hardmacro memories. The usage is similar to `no_hardmacro`.

Both synthesis directives are effective when memory synthesis option is turned on. These can be used with either Verilog or VHDL designs.

Syntax and Semantic Restrictions

HDL Compiler synthesis tool for VHDL designs is based on VHDL 1076-1993, which can be fully simulated, but not fully synthesized.

HDL Compiler has the following restrictions on using VHDL in synthesis:

- Accepts, but cannot operate on (and therefore ignores), the `FILE`, `REAL`, `ACCESS`, and `PHYSICAL` data types.
- Does not support `REGISTER` and `BUS` signal declarations.
- Ignores assertion statements.

© 2021 All Rights Reserved.

- Does not support the `BEHAVIOR`, `STRUCTURE`, `LAST_EVENT`, `LAST_ACTIVE`, and `TRANSACTION` attributes.
- Ignores the `after` clause in signal assignment statements.
- When you import your design into HDL Compiler, the loop index must evaluate to constants.
- When you use `wait` or `if` statements, they must include the `event` or `stable` edge detection attributes. These attributes detect the leading or falling edge of a signal. See Sequential Statements on page 544, for details.
- Ignores the initial value of a signal (or variable) in a signal (variable) declaration.
- Supports design descriptions containing multiple `wait` statements, but you must specify either a synchronous reset or asynchronous reset signal (see Wait Statements on page 545 and Asynchronous Set/Reset Directives on page 569). Otherwise, HDL Compiler might not be able to determine the power-up state in the real circuit.
- Extended identifiers are not supported.

Modeling Styles

VHDL design style can be flexible. But some modeling (or coding) styles produce better synthesis results than other styles. This section describes the coding styles that produce the best results when synthesizing for emulation.

Combinational Logic

A process without edge detecting attributes (`event` and `stable`) in `if` or `wait` statements is called a co*mbinational process*. A combinational process is usually synthesized as a combinational circuit (except on latch inferences, as described in Latches on page 572).

The following is an example of a combinational circuit:

```
process (A,B,C,D,Sel)
begin
  if (Sel = '1') then
    Z <= B + C ;
  else
    Z <= A + D ;
  end if ;
end process ;
```


© 2021 All Rights Reserved.

Latches

Flip-flops and latches are two commonly used, one-bit memory devices. A flip-flop is an edge-triggered memory device. A latch is a level-sensitive memory device.

The following example describes a level-sensitive latch:

```
 -- latch inference
signal S, Data_In, Data_Out: BIT ;
. . .
process (S, Data_In)
begin
  if (S = '1') then
    Data_Out <= Data_In ;
  end if ;
end process ;
```

The sensitivity list includes the `S` and `Data_In` signals, which are required for processes without `wait` statements. This list indicates that whenever the `S` and `Data_In` signals change, the process is evaluated.

Because the assignment to the `Data_Out` signal is hidden in a conditional clause, `Data_Out` does not change value (preserves its old value) if `S` is zero (0). `Data_Out` is updated with the value of `Data_In` if `S` equals one (1). This is the behavior of a level-sensitive latch.

In general, latches are synthesized from conditional expressions that are not completely specified, in a combinational description. Any signal or variable not driven under all conditions, becomes a latched element.

Latch inferences normally occur with `if` or `case` statements. To avoid having a latch inferred, assign a value to the signal under all conditions. For example, if you add an `else` part to the `if` statement, then the `if` statement becomes a completely specified statement, and is therefore synthesized into a combinational circuit.

Flip-Flops

A process containing edge-detecting attributes (`event` and `stable`) in `if` or `wait` statements is called a `clocked process`. HDL Compiler infers an edge-triggered flip-flop for signals in signal assignment statements. By detecting clock edges, HDL Compiler can locate where to insert flip-flops, so that the synthesized gate-level design matches the simulation behavior.

The `event` attribute on a signal is the most commonly used edge-detecting mechanism. It operates on a signal, and returns a Boolean value. The result is true if the signal shows a change in value.

The `stable` attribute is the Boolean inversion of the `event` attribute.

© 2021 All Rights Reserved.

The following example shows a basic VHDL program that describes a flip-flop. The process contains an `if` signal leading-edge statement. At each leading edge of the `CLK` signal, the content of `Data_In` is assigned to `Data_Out`. The rising edge of the clock is signified by "clk=1."

```
signal CLK, Data_In, Data_Out: BIT ;
. . .
process ( CLK )
begin
  if (CLK'event and CLK = '1') then
    Data_Out <= Data_In ;
  end if ;
end process ;
```

Synchronous Sets and Resets

A synchronous set (reset) input presets (clears) the output of a flip-flop when it is asserted. The assignments takes effect only while the clock edge is active; at all other times, the flip-flop does not detect changes on these inputs.

All signal assignments inside the edge detection clause of an `if` signal, cause the values to be assigned to flip-flops along the signal edge. HDL Compiler can model a flip-flop with synchronous reset, by detecting whether a synchronous input was set on the leading or falling edge of the signal.

```
signal CLK, Data_In, Data_Out, S_RST: BIT ;
. . .
process ( CLK )
begin
  if (CLK'event and CLK = '1') then
    if (S_RST = '1') then
      Data_Out <= '0' ;
    else
      Data_Out <= Data_In ;
    end if ;
  end if ;
end process ;
```

Note: The `S_RST` and `Data_In` signals need not be on the sensitivity list of the process. Their changes do not result in any action inside the edge-detection clause of the if signal.

Asynchronous Sets and Resets

D-type flip-flops sometimes require asynchronous set or reset inputs. Asynchronous set (reset) inputs clear the output of flip-flops, when they are asserted independently of the clock.

The following VHDL program example describes a D-type flip-flop, with an asynchronous reset input:

```
signal CLK, A_RST, Data_In, Data_Out: BIT ;
...
```


© 2021 All Rights Reserved.

```
process (CLK, A_RST)
begin
  if (A_RST = '0') then
    Data_Out  <= '0' ;
  elsif (CLK'event and CLK = '1') then
    Data_Out  <= Data_In ;
  end if ;
end process ;
```

The entity now has an extra input, the `A_RST` port, which resets the D flip-flop asynchronously. The `CLK` and `A_RST` signals must be in the sensitivity list. Whenever an event occurs on either the `CLK` or `A_RST` signal, the statements inside the process are executed.

HDL Compiler first tests whether the `A_RST` signal value is `1` (active-`HIGH` asynchronous reset). If so, then the flip-flop output is reset to 0. Otherwise, the value of `Data_In` is assigned to the `Data_Out` signal, at the leading edge of the `CLK` signal.

You can describe a flip-flop that has more than one asynchronous input, as in the following example:

```
signal CLK, RST, PRST, EN: BIT ;
signal Data_In, Data_Out: BIT ;
process (CLK, PRST, RST, EN)
begin
  if (PRST = '1') then
    Data_Out <= '1' ;
  elsif (RST = '1') then
    Data_Out <= '0' ;
  elsif (CLK'EVENT and CLK = '1') then
    if (EN = '1') then
      Data_Out <= Data_In ;
    end if ;
  end if ;
end process ;
```

This circuit example performs a prioritized testing of the `PRST` signal, with respect to the `RST` signal. The `PRST` signal is tested before the `RST` signal. If the `PRST` signal is active, then the flip-flop is preset to `1`, regardless of the state of the `RST` input.

If the `PRST` signal is inactive, then the flip-flop resets to `0` when the `RST` input is asserted. Otherwise, the `Data_In` is assigned to `Data_Out` synchronously, when the `EN` signal is asserted.

Registers

Various types of registers are used in a circuit. The following VHDL program example infers a basic 4-bit register:

```
 -- 4-bit basic register
signal CLK, ASYNC: BIT ;
signal Din, Dout: BIT_VECTOR(3 downto 0) ) ;
. . .
```


© 2021 All Rights Reserved.

```
process (CLK, ASYNC)
begin
  if (ASYNC = '1') then
    Dout <= "1100";
  elsif (CLK'event and CLK = '1') then
    Dout <= Din ;
  end if ;
end process ;
```

The clock pulse enables all flip-flops, so that the information currently available at the four inputs, can be transferred into the 4-bit register. The register is preset to an initial value of `1100` asynchronously, when the `ASYNC` signal becomes `1`.

You can use a `load` signal to load input data into a register. When the `LOAD` input is 1, the inputs transfer into the register on the next clock pulse. When the `LOAD` input is 0, inputs are inhibited, so the D flip-flops maintain their present value.

If the technology library contains flip-flops with enable, then HDL Compiler can use the library to build the circuit. If it does not find these flip-flops, then HDL Compiler uses flip-flops and multiplexers to synthesizes a circuit that meets the behavior of the description.

```
-- 4-bit parallel load register
signal CLK, ASYNC, LOAD: BIT ;
signal Din, Dout: BIT_VECTOR(3 downto 0) ;
. . .
process (CLK, ASYNC)
begin
  if (ASYNC = '1') then
    Dout <= "1100";
  elsif (CLK'EVENT and CLK = '1') then
    if (LOAD = '1') then
      Dout <= Din ;
    END IF ;
  END IF ;
end process ;
```

If you declare a range of values for a register, such as:

```
reg [2:0] div
```

You cannot assign a `register` value outside of the declared range, such as:

```
wire div[3]
```

Asynchronous Counters

Counters are sequential logic circuits that proceed through a well-defined sequence of states. An asynchronous counter does not use a synchronizing clock pulse to control state changes. The most common type of asynchronous counter is the binary ripple counter. A binary ripple counter consists of a series connection of complementing flip-flops (T type), with the output of each flip-flop connected to the clock input of the next higher-order flip-flop. The flip-flop holding the least significant bit, receives the incoming count pulses.

© 2021 All Rights Reserved.

```
-- 4-bit binary counter
signal CLK, RESET: BIT ;
signal COUNT: BIT_VECTOR(3 DOWNTO 0) ;
. . .
process (CLK, COUNT, RESET)
begin
  if RESET = '1' then
    COUNT <= "0000";
  else
    if CLK'event and CLK = '1' then
      COUNT(0) <= not COUNT(0) ;
    end if ;
    if COUNT(0)'event and COUNT(0) = '0' then
      COUNT(1) <= not COUNT(1) ;
    end if ;
    if COUNT(1)'event and COUNT(1) = '0' then
      COUNT(2) <= not COUNT(2) ;
    end if ;
    if COUNT(2)'event and COUNT(2) = '0' then
      COUNT(3) <= not COUNT(3) ;
    end if ;
  end if ;
end process ;
```

Synchronous Counters

If all flip-flops of a counter are controlled by a common clock signal, then it is a synchronous counter. In a synchronous counter, the storage elements simultaneously examine their inputs and determine new outputs. This is the preferred way to build counters.

For example, the following program example describes a synchronous counter with asynchronous reset, synchronous load, and up or down count. When the `Load` signal equals `1`, then the count sequence is disabled, and the external input is transferred into the counter. If the `Load` signal is `0` and the `Count` signal is `1`, then the circuit operates as a counter. Depending on the value of the `Updown` signal, the clock pulses cause the state of the register to either count up or count down. If both `Load` and `Count` are `0`, then the clock pulses do not change the state of the register.

```
-- 4-bit synchronous counter
signal CLK, RESET : BIT ;
signal Load, Count, UpDown  : BIT ;
signal DataIn  : INTEGER range 0 to 15 ;
signal REG  : INTEGER range 0 to 15 := 0 ;
. . .
process (CLK, RESET)
begin
  if RESET = '1' then
    REG <= 0 ;
  elsif CLK'event and CLK = '1' then
    if Load = '1' then
      REG <= DataIn ;
    else
      if Count = '1' then
```


© 2021 All Rights Reserved.

```
        if UpDown = '1' then
          REG <= (REG + 1) mod 16 ;
        else
          REG <= (REG - 1) mod 16 ;
        end if ;
      end if ;
    end if ;
  end if ;
end process ;
```

Tristate Buffers

Besides 0 and 1, digital systems can have a third signal value: the *high-impedance* state, usually denoted by the `Z` symbol. When the output of a gate is in a high-impedance state, the gate acts as if it were disconnected from the output. Gates that can be placed in such a state are called *tristate* gates. They can produce outputs of the following three values: `0`, `1`, and `Z`.

In addition to its conventional inputs, a tristate gate has one more input called *output enable*. If this input is not asserted, then the output is in its high-impedance state, and the gate is effectively disconnected from the output wire.

If the output enable is asserted, then the data inputs of a gate determine its output.

The predefined types of the `STANDARD` package do not include a type that describes the high-impedance value. To specify the high-impedance value in VHDL, you must use the `STD_LOGIC` multi-value logic type in the `STD_LOGIC_1164` package. When a signal is assigned the `Z` value, the output of the tristate gate is disabled. HDL Compiler synthesizes a tristate buffer between the `Din` and `Dout` signals. The `OE` signal controls the state of the tristate buffer. In the `if` clause, both `Din` and the `OE=0` condition can be full expressions.

```
library IEEE ;
use IEEE.STD_LOGIC_1164.all ;

architecture IMP of Tri_State_Buf is
signal Din, OE, Dout : STD_LOGIC ;
begin
  . . .
  process(OE, Din)
  begin
    if (OE = '0') then Dout <= 'Z';
    else Dout <= Din;
    end if;
  end process;
end IMP ;
```

Another handy notation of a tristate buffer is to use a conditional signal assignment, as in the following example:

```
architecture Implement of Tri_State_BUF is
begin
  Dout <= Din when OE = '1' else 'Z' ;
end Implement ;
```


© 2021 All Rights Reserved.

Buses

You can use tristate gates, instead of multiplexers, to construct a bus system. The outputs of four buffers are connected, to form a single bus line. The control inputs to the buffers, `S(0)` and `S(1)`, determine which one of the four normal inputs (`R0-R3`) communicates with the bus line. The circuit designer must guarantee that no more than one buffer is in the active state at any time. Only one of the connected tristate buffers can have access to the bus line; all other buffers must be maintained in a high-impedance state.

```
library IEEE ;
use IEEE.STD_LOGIC_1164.all ;

entity BUS_EX is
  port ( S : in  STD_LOGIC_VECTOR(1 downto 0) ;
         OE : buffer STD_LOGIC_VECTOR(3 downto 0) ;
         R0, R1, R2, R3 : in STD_LOGIC_VECTOR(7 downto 0) ;
         BusLine : out STD_LOGIC_VECTOR(7 downto 0) ) ;
END BUS_EX ;

architecture Implement of BUS_EX is
begin
  process (S)
  begin
    case (S) is
      when "00" => OE <= "0001" ;
      when "01" => OE <= "0010" ;
      when "10" => OE <= "0100" ;
      when "11" => OE <= "1000" ;
      when others => null ;
    end case ;
  end process ;
  BusLine <= R0 when OE(0) = '1' else 'ZZZZZZZ' ;
  BusLine <= R1 when OE(1) = '1' else 'ZZZZZZZ' ;
  BusLine <= R2 when OE(2) = '1' else 'ZZZZZZZ' ;
  BusLine <= R3 when OE(3) = '1' else 'ZZZZZZZ' ;
end Implement ;
```

Normally, simultaneous assignment to a signal (such as `BusLine` in the preceding example) is not allowed at the architectural level, unless a resolution function is provided. However, in VHDL, the `STD_LOGIC` or `STD_LOGIC_VECTOR` data types are already resolved (a resolution function is provided in `STD_LOGIC_1164`). Therefore, signals declared with these two data types can have multiple drivers.

Moore Machines

In the Moore model of sequential circuits, the outputs are functions only of the present state. You can represent a Moore model of a sequential circuit as a state diagram, with the circles marked to show the current state and output, and directed edges marked with the input only.

The following example shows a VHDL description for the Moore machine. In this example, the asynchronous reset signal clears the content of the `O` register, and resets the finite state machine to its initial state. At each leading edge of the `CLK` signal, the value of the next state

© 2021 All Rights Reserved.

is assigned into the current state. The rest of the statements describe the combinational logic, which generates the next state and the outputs of the design. The outputs are computed based on the current state only, so the assignments of output signals cannot depend on the input signals.

```
entity MOORE is
  port ( CLK : in  BIT ;
         RST : in  BIT ;
         I   : in  BIT ;
         O   : out BIT ) ;
end MOORE ;

architecture Implement of MOORE is
begin
  process (CLK, RST, I)
    type STATE_TYPE is (ST0,ST1,ST2,ST3);
    variable State,Next_State : STATE_TYPE ;
  begin
    if (RST = '1') then
      O <= '0' ;
      State := ST0 ;
    elsif (CLK'event and CLK = '1') then
      State := Next_State ;
    end if ;
    case State is
      when ST0 =>
        O <= '0' ; -- output depends on present state
        if (I = '0') then  Next_State := ST0 ;
        else               Next_State := ST1 ;
        end if ;
      when ST1 =>
        O <= '0' ;
        if (I = '0') then  Next_State := ST0 ;
        else               Next_State := ST2 ;
        end if ;
      when ST2 =>
        O <= '1' ;
        if (I = '0') then  Next_State := ST3 ;
        else               Next_State := ST2 ;
        end if ;
      when ST3 =>
        O <= '1' ;
        if (I = '0') then  Next_State := ST3 ;
        else               Next_State := ST0 ;
        end if ;
    end case ;
  end process ;
end Implement ;
```

Asynchronous Mealy Machines

In the Mealy model, the outputs are functions of both the present state and current inputs. The outputs can change if the inputs change during the clock pulse period. The outputs might momentarily show false values, because of the delay encountered from the time that the inputs change, to the time that the flip-flop outputs change. To synchronize an asynchronous

© 2021 All Rights Reserved.

Mealy machine, the inputs of the sequential circuit must be synchronized with the clock, and the outputs must be sampled only during the clock-pulse transition.

In a state transition diagram representation of a Mealy machine, both the input and output values are included along the directed lines between the circles. The following example shows the corresponding VHDL program:

```
entity ASYNC_MEALY is
   port ( CLK                : in  BIT ;
         RST               : in  BIT ;
         I                 : in  BIT ;
         O                 : out BIT  ) ;
   end ASYNC_MEALY ;

   architecture Implement of ASYNC_MEALY is
   begin
      process (CLK, RST, I)
      type STATE_TYPE is (ST0,ST1,ST2,ST3,ST4) ;
      variable State,Next_State : STATE_TYPE ;
   begin
      if (RST = '1') then
         State := ST0;
      elsif (CLK'event and CLK = '1') then
         State := Next_State ;
      end if ;

   case STATE is
      when ST0 =>
         if (I = '0') then  O <= '0';  Next_State := ST1 ;
         else               O <= '1';  Next_State := ST0 ;
         end if ;
      when ST1 =>
         if (I = '0') then  O <= '0';  Next_State := ST1 ;
         else               O <= '0';  Next_State := ST2 ;
         end if ;
      when ST2 =>
         if (I = '0') then  O <= '0';  Next_State := ST0 ;
         else               O <= '0';  Next_State := ST3 ;
         end if ;
      when ST3 =>
         if (I = '0') then  O <= '0';  Next_State := ST3 ;
         else               O <= '1';  Next_State := ST4 ;
         end if ;
      when ST4 =>
         if (I = '0') then  O <= '1';  Next_State := ST2 ;
         else               O <= '0';  Next_State := ST0 ;
         end if ;
   end case ;
  end process ;
end Implement ;
```

At each leading edge of the `CLK` signal in this example, the value of the next state is assigned into the current state. The combinational section describes the combinational logic, for generating both the next state and the outputs of the design. HDL Compiler generates the output signals, based on the value of the input signals.

© 2021 All Rights Reserved.

Algorithmic State Machines

You can represent the behavior of your design as a flow-chart description, such as an Algorithmic State Machine (ASM). An ASM chart contains three basic elements: the state box, the decision box, and the conditional output box.

- A state in the control sequence is indicated by a state box (a rectangular box).
- The decision box, which describes the effect of an input on the system, is a diamond-shaped box that has two or more exit paths.
- The conditional output box is represented as an oval box. The input path of a conditional box must originate from one of the exit paths of a decision box. The operations listed inside the conditional box are executed during a specified state, if the input condition is satisfied.

The following example is a VHDL program for an Algorithmic State Machine that calculates the Greatest Common Divider (GCD) of two numbers.

```
architecture FSMD of GCD is
begin
  Main :
  process (RESET, CLOCK)
  variable X, Y: INTEGER range 0 to 255 ;
  type S_Type IS (ST0 , ST1  );
  variable State          : S_Type := ST0 ;
  variable TMP_0, TMP_1   : BOOLEAN ;
  begin
    if (RESET = '1') then
      OUT <= 0 ;
      State := ST0 ;
    elsif (CLOCK'event and CLOCK = '1') then
      case State is
        when ST0 =>
          if (START = '1') then
            X := XI ;
            Y := YI ;
            State := ST1 ;
          else
            State := ST0 ;
          end if ;
        when ST1 =>
          if (X /= Y) then
            if (X < Y) then
              Y := Y - X ;
            else
              X := X - Y ;
            end if ;
            State := ST1 ;
            else
              OUT <= X ;
              State := ST0 ;
            end if ;
      end case ;
    end if ;
```


© 2021 All Rights Reserved.

```
  end process Main ;
end FSMD ;
```

Cycle-Fixed Algorithm Using Multiple Waits

You can use sequential statements in VHDL to describe any algorithm. When modeling an algorithm, HDL Compiler uses signal assignment statements, to communicate externally. It also uses `wait` statements, to synchronize the circuit execution with outside events.

You can use VHDL to describe an algorithmic behavior, if you properly insert the `wait` statements to delimit the clock boundaries.

The following VHDL example describes how the Greatest Common Divider (GCD) of two numbers is calculated:

```
entity GCD is
       port ( CLOCK : in  BIT ;
              RESET : in  BIT ;
              XI,YI : in  INTEGER range 0 to 255 ;
              READY : in  BIT ;
              OUT   : out INTEGER range 0 to 255 := 0 ) ;
end GCD ;
architecture BHV of GCD is
-- mebs reset_high RESET
begin
  Main : process
  variable X, Y : INTEGER range 0 to 255 ;
  begin
    wait until(CLOCK'event and CLOCK = '1' and READY = '1');
    X := XI ;
    Y := YI ;
    while (X /= Y) loop
      wait until(CLOCK'event and CLOCK = '1');
      if (X < Y) then
        Y := Y - X ;
      else
        X := X - Y ;
      end if ;
    end loop ;
    OUT <= X ;
  end process Main ;
end BHV ;
```

The program includes two `wait` statements. When the `READY` signal is `HIGH` at a leading edge of the `CLOCK` signal, then the two eight-bit signals move into the `X` and `Y` local variables. The `while ... loop` statement computes the GCD of the two numbers.

Note: Because a wait statement is inserted in the while statement, every iteration executes in exactly one clock cycle. When the computation is over, the result moves to the OUT output port.

© 2021 All Rights Reserved.

Synchronous Reset in Multiple-Wait Descriptions

When a synchronous reset signal is asserted (active-`HIGH` or active-`LOW`), the synthesized design is forced to the state that is the equivalent of the beginning of the process. Some of the registers are also either cleared, or preset to certain values. In VHDL, you need to test a synchronous reset signal after each `wait` statement.

```
entity GCD is
         port ( CLOCK : in  BIT ;
              S_RESET : in  BIT ;
              XI,YI   : in  INTEGER range 0 to 255 ;
              READY   : in  BIT ;
              OUT     : out INTEGER range 0 to 255 := 0 ) ;
end GCD ;

architecture BHV of GCD is
begin
  Main : process
  variable X, Y : INTEGER range 0 to 255 ;
  begin
    s_reset_loop: loop
      wait until(CLOCK'event and CLOCK = '1' and READY = '1');
      exit s_reset_loop when s_reset = '1';
      X := XI ;
      Y := YI ;
      while (X /= Y) loop
        wait until(CLOCK'event and CLOCK = '1');
        exit s_reset_loop when s_reset = '1';
        if (X < Y) then
          Y := Y - X ;
        else
          X := X - Y ;
        end if ;
      end loop ;
      OUT <= X ;
    end loop s_reset_loop;
    OUT <= 0;
  end process Main ;
end BHV ;
```

Asynchronous Reset in Multiple-Wait Descriptions

In a design description with multiple `wait` statements, you must use directives that specify the asynchronous reset signal. After detecting the directive, HDL Compiler connects the asynchronous reset line to the state registers. This connection sets the state registers to the initial state, after the reset signal is asserted. An asynchronous reset directive starts with `pragma`, followed by `reset_high` `reset_low`, indicating that it is first an active-`HIGH` signal, and then the asynchronous reset signal.

The following is an example of using asynchronous reset in a multiple-wait description.

```
entity GCD is
       port ( CLOCK : in  BIT ;
              RESET : in  BIT ;
```


© 2021 All Rights Reserved.

```
              XI,YI : in  INTEGER range 0 to 255 ;
              READY : in  BIT ;
              OUT   : out INTEGER range 0 to 255 := 0 ) ;
end GCD ;

architecture BHV of GCD is
-- pragma reset_high RESET
begin
  Main : process
  variable X, Y : INTEGER range 0 to 255 ;
  begin
    wait until(CLOCK'event and CLOCK = '1' and READY = '1');
    X := XI ;
    Y := YI ;
    while (X /= Y) loop
      wait until(CLOCK'event and CLOCK = '1');
      if (X < Y) then
        Y := Y - X ;
      else
        X := X - Y ;
      end if ;
    end loop ;
    OUT <= X ;
  end process Main ;
end BHV ;
```

Support for VHDL 2008

When compiling RTL code with VHDL 2008 constructs, use the following options:

```
hdlImport -v2008
```

Or

```
vavhdl -v2008
vavhdl -v200x
```

The following constructs from the 1076-2008 standard for VHDL are supported:

- integer_vector (section 16.3 and 5.3.2.3)
- External names (section 8.7, partial support)
- C Style Comments (section 15.9)
- Elsif/else generate and case generate (section 11.8)
- Condition Operators (section 9.2.9)
- Enhanced Bit String Literals (section 15.8, vs section 13.7 in 1993 LRM)
- Sequential Conditional Assignment (section 10.5.3 and 10.5.4)
- Slices in Array Aggregates (section 9.3.3)

© 2021 All Rights Reserved.

- Context Declarations (section 13.3)
- RISING_EDGE and FALLING_EDGE Functions (section 16.3 and 5.2.6)]
- MAXIMUM and MINIMUM Functions (section 16.3 and 5.3.2.4)
- Unconstrained Element Type (section 5.1 and 6.3)
- Alias Declarations (section 6.6)
- Generic Function/Procedure (section 4, partial support)

INTEGER_VECTOR [5.3.2.3 and 16.3]

INTEGER_VECTOR is defined in section 5.3.2.3. Predefined array types are described in section 16.3.

The value of the predefined type INTEGER_VECTOR is a one-dimensional array of the predefined type INTEGER, indexed by values of the predefined subtype NATURAL.

```
type INTEGER_VECTOR  is array (NATURAL  range <>)  of INTEGER;
```

The operations to be performed on INTEGER_VECTOR are listed in section 16.3.

Sample code:

```
entity test is
       port(a_in0, a_in1, b_in0, b_in1, c_in0, c_in1 : in integer;
            out1, out2, out3, out4, out5 : out boolean);
end test;

architecture arch of test is
       signal a_in, b_in, c_in: integer_vector(0 to 1);
       signal sinv: integer_vector(0 to 3);
       signal sin1, sin2: integer;
       signal sel: integer range 0 to 1 := 0;
       begin
            a_in <= a_in0 & a_in1;
            b_in <= b_in0 & b_in1;
            c_in <= c_in0 & c_in1;
            sin1 <= a_in0 + b_in1;
            sin2 <= a_in1 - b_in0;
            sinv <= (sin1 & c_in) & sin2;
            out1 <= a_in /= b_in;
            out2 <= (a_in & b_in) > sinv;
            out3 <= c_in <= (sin1 & sin2);
            out4 <= a_in < sinv(1 to 2);
            out5 <= (a_in(sel) & b_in(0)) >= c_in;
       end arch;
```


© 2021 All Rights Reserved.

External names [8.7]

External names are defined in section 8.7. An external name denotes an object declared in the design hierarchy containing the external name. It is a hierarchical reference to the object defined at the place denoted by the external name from the current scope.

Following are the limitations in support for external names:

- Only external names to signals are supported. The following syntax is not supported:

  ```
  relative_pathname ::= { ^ . } partial_pathname
  ```

- HDL Compiler does not support external names that contain block names in the hierarchical path.
- HDL Compiler can verify the correctness of an external name if it denotes an object in the current design unit. Else, use tools like Xcelium to verify the external name.

Sample code:

```
entity e is
   port ( S1 : out integer);
   end e;
   architecture a_e of e is
    begin
      -- Absolute pathname referring to signal S in top, the design entity that
      -- is the root of the design hierarchy.
      S1 <= <<signal .top.S1 : integer>>;
    end a_e;
   entity top is
        port ( S1, S2 : out integer);
   end top;
   architecture top_a of top is
    begin
      S1 <= 5;
      I1 : entity work.e port map (S2);
    end top_a;
```

C-Style Comments [15.9]

Earlier versions of VHDL had single-line comments, starting with the characters `--` and extended to the end of the line. VHDL 2008 keeps this style of comment, and adds delimited comments, starting with the characters `/*`, extending to the closing characters `*/`. The opening and closing characters can be on different lines, or on the same line. Moreover, you can include VHDL code on the line after the closing characters.

Following is an example of comments:

```
/* This is a comment header that describes the purpose of the design unit.
   It contains all you ever wanted to know, plus more. */
```


© 2021 All Rights Reserved.

An occurrence of two adjacent hyphens within a delimited comment is not interpreted as the start of a single-line comment. Similarly, an occurrence of a solidus character immediately followed by an asterisk character within a single-line comment is not interpreted as the start of a delimited comment. Moreover, an occurrence of a solidus character immediately followed by an asterisk character within a delimited comment is not interpreted as the start of a nested delimited comment. However, note that comments should not be nested.

For example, the following syntax is incorrect:

```
-- Here is the start of the comment: /* A comment extending
    over two lines */
```

The use of characters `*/` on the second line to close the comment started on the first line, is not correct. It makes the orphaned delimiter `*/` on the third line illegal.

```
/* Comment out the following code:
   signal count_en : bit; /* counter enable */
*/
```

Synthesis pragmas can be used within delimited comment. However, synthesis pragmas should be used in a single line within a delimited comment:

```
/* pragma_keyword pragma_name [optional_pragma_value] */
```

Conformal supports C-style comments. So, you can verify the netlist generated by HDL Compiler for a design that uses C-style comments by using Conformal 14.10.100 to compare it with the RTL source code.

Sample RTL code with single-line and multi-line comments:

```
architecture arch of test is
begin
    bv_out1 <= a_bs & b_bs;
    /* single-line comment */
    bv_out2 <= c_bs & bv_out1;
    /* multi-line
       comment */
end arch;
```

Sample RTL code with pragmas in comments:

```
entity test is
    port(a_bs, b_bs : in bit;
         c_bs  : in bit;
         bv_out1 : out bit_vector(0 to 1));
end test;

architecture arch of test is
begin
    bv_out1 <= a_bs & b_bs;
    /* pragma synthesis_off */
    bv_out2 <= c_bs & bv_out1;
    /* pragma synthesis_on */
end arch;
```


© 2021 All Rights Reserved.

elsif/else generate and case generate [11.8]

Generate statements are extended with `elsif` and `else generate` in VHDL 2008. The `case generate` statement is also added. The `generate_label` and `alternative_label` are also supported. Signals defined in `block_declarative_part` of `generate_statement_body` are supported. It is recommend that `alternative_label` added in that statement.

Sample code:

```
entity test is
     port ( in1,in2,in3 : bit_vector( 0 to 7);
            output:out bit_vector(0 to 7)
          );
end;
architecture arch of test is
    constant const1: integer:= 0;
    constant const2: integer:= 4;
    begin
         l2: if l2_s1: ((const2-const1) > 5) generate
                signal s1 : std_logic;
                begin
                     output <= in1;
             elsif ((const2-const1) = 0) generate
                begin
                     output <= in2;
             else l2_s3: generate
                signal s3 : std_logic;
                begin
                     output <= in3;
             end generate;
    end arch;
```

```
entity test is
     port ( in1,in2,in3,in4 : bit_vector( 0 to 7);
          output:out bit_vector(0 to 7)
          );
     end;
  architecture arch of test is
     constant const1: integer:= 0;
     constant const2: integer:= 4;
   begin
       l2: case (const2-const1) generate
           when ls_s1: 1|2|3 =>
           signal s1 : std_logic;
             begin
                  output <= in2;
             when 4 =>
             signal s2 : std_logic;
             begin
                  output <= in1;
             when ls_s3: 5|6|7 =>
             begin
                  output <= in3;
             when l2_s4: others =>
             signal s4 : std_logic;
             begin
                  output <= in4;
```


© 2021 All Rights Reserved.

```
            end generate;
   end arch;
```

Condition Operator [9.2.9]

The condition operator is new operators defined in VHDL-2008 LRM, section 9.2.9.

The condition operator `??` is predefined for type BIT defined in package STANDARD (see 16.3). Conversion of a value of type BIT converts '1' to TRUE and '0' to FALSE. The conversion operator may be overloaded for other types.

```
function "??" (anonymous: BIT) return BOOLEAN;
```

The condition operator is defined for type STD_ULOGIC defined in package STD_LOGIC_1164 (see 16.7). Conversion of a value of type STD_ULOGIC converts '1' and 'H' to TRUE and all other values to FALSE.

Some examples are:

```
    signal cs1, ncs2, cs3 : std_ulogic;
if ?? (cs1 and not cs2 and cs3) then
...
end if;
```

The condition operator implicitly applied, if any, is either the predefined operator for type BIT or an overloaded operator, determined as follows. If, without overload resolution (see 12.5), the expression is of type BOOLEAN defined in package STANDARD, or if, assuming a rule requiring the expression to be of type BOOLEAN defined in package STANDARD, overload resolution can determine at least one interpretation of each constituent of the innermost complete context including the expression, then the condition operator is not applied. Otherwise, the condition operator is implicitly applied, and the type of the expression with the implicit application shall be BOOLEAN defined in package STANDARD.

```
    if cs1 and cs3 and alu_op = pass then ... - illegal
```

The result of the “=” comparison is of type boolean, which cannot be mixed with the std_ulogic operand for the and operator. The conversion is only applied to the overall condition. We would have to write the condition as:

```
    if cs1 = '1' and cs3 = '1' and alu_op = pass then ...
```

Condition operator are not supported from Conformal and CAN’T be verified using Conformal 14.10.100. We need to write testbench and verify the result netlist using simulator, or write function equivalent test case for verifying with Conformal 14.10.100.

Sample code:

RTL code with explicit condition operator:

© 2021 All Rights Reserved.

```
entity tri_state is
    port (ena: in bit;
          input: in std_logic_vector(7 downto 0);
          output: out std_logic_vector(7 downto 0));
end tri_state;
architecture tri_state of tri_state is
begin
    output <= input when not(?? ena) else
              (others=>'Z');
end tri_state;
```

RTL code with implicit condition operator:

```
entity tri_state is
    port (ena: in bit;
          input: in std_logic_vector(7 downto 0);
          output: out std_logic_vector(7 downto 0));
end tri_state;
architecture tri_state of tri_state is
begin
    output <= input when (not ena) else
              (others=>'Z');
end tri_state;
```

Enhanced Bit String Literals [15.8 (versus section 13.7 in 1993 LRM)]

Enhanced Bit String Literals is introduced in VHDL-2008 LRM, section, 15.8 Bit string literals.

Here's the syntax difference between enhanced bit string literals and ordinary bit string literals.

Enhanced bit string literals:

```
bit_string_literal ::= [ integer ] base_specifier " [ bit_value ] "
bit_value ::= graphic_character { [ underline ] graphic_character }
base_specifier ::= B | O | X | UB | UO | UX | SB | SO | SX | D
```

Ordinary bit string literals:

```
bit_string_literal ::= base_specifier "[ bit_value ]"
bit_value ::= extended_digit { [ underline ] extended_digit }
base_specifier ::= B | O | X
```

In VHDL-2008, bit string literals are enhanced.

© 2021 All Rights Reserved.

- Bit string literals may have an explicit width,
- Bit string literals may be declared as signed or unsigned,
- Bit string literals may include meta-values ('U', 'X', etc.)
- Bit string literals may be interpreted as a decimal integer if the base specifier is D.

Here are some examples:

```
variable S : std_logic_vector(5 downto 0);
  begin
    S := 6x"0f";-- specify width 6
    S := 6x"XF";-- means "XX1111"
    S := 6SX"F";-- "111111" (sign extension)
    S := 6Ux"f";-- "001111" (zero extension)
    S := 6sb"11";-- "111111" (binary format)
    S := 6uO"7";-- "000111" (octal format)
```

Note that within bit string literals it is allowed to use either upper or lower case letters, i.e. F or f.

The result netlist of test case using enhanced bit string literals produced by HDL Compiler can be verified by comparing with RTL source code using Conformal 14.10.100. Enhanced bit string literals has been supported in Conformal.

Sample code:

```
entity bit_string_liter is
    port(clk,reset : in std_logic;
         digit : out std_logic_vector(6 downto 0));
end bit_string_liter;

architecture bit_string_liter of bit_string_liter is
begin
        process (clk,reset)
        variable temp: integer ;
    begin
      for temp in 0 to 15 loop
    case temp is
        when 0 => digit <= 7X"7F" ;     --"1111111" Lengths
        when 1 => digit <= 7uX"F" ;     --"0001111" extend unsigned
        when 2 => digit <= 7Ux"0F";     --"0001111" reduce unsigned
        --when 3 => digit <= 7UX"8F";   --"10001111" error
        when 4 => digit <= 7sX"F" ;     --"1111111" extend signed
        when 5 => digit <= 7SX"CF";     --"1001111" reduce signed
        --when 6 => digit <= 7SX"8F";   --"10001111" error
        when 7 => digit <= 7sx"-X";     --"---XXXX" -,x,z handling
        when 8 => digit <= 7d"127";     --"1111111" decimal values
        when 9 => digit <= "111"&X"F";  --7F standard
        --when 10=> digit <= X"7F";     --7F standard error with length
        when 11=> digit <= b"100_1X11"; --nomal B with _
        when 12=> digit <= "000"&X"F";  --nomal X
        when 13=> digit <= "0"&o"77";   --nomal o
        when 14=> digit <= "111"&X"7";  --nomal x
        when 15=> digit <= 7SB"0H1";    --"00000H1"
        when OTHERS => null;
```


© 2021 All Rights Reserved.

```
    end case;
      end loop;
    end process ;
end bit_string_liter;
```

Sequential Conditional Assignment [10.5.3 and 10.5.4]

Sequential conditional assignment includes conditional signal assignment and selected signal assignment, which are defined in VHDL-2008 LRM, section 10.5.3 and 10.5.4 respectively. In HDL-ICE, following syntax is supported:

```
conditional_signal_assignment ::=
    conditional_waveform_assignment
selected_signal_assignment ::=
    selected_waveform_assignment
```

The following syntax is not supported as force & release assignment is not ready yet.

```
conditional_force_assignment
selected_force_assignment
```

keyword “unaffect” is acceptable in conditional signal assignment.

Sample code:

```
entity test is
  port (
    sel1, sel2  : in bit_vector(0 to 1);
    a,b,c,d: in bit_vector(0 to 1);
    o1, o2    : out bit_vector(0 to 1));
end test ;

architecture arch of test is
begin
  process (all)
  begin
    o1 <= a when (sel1 = "00") else
          b when (sel1 = "01") else
          c when (sel1 = "10") else
          d;
    with sel2 select
        o2 <= a when "00",
              b when "01",
              c when "10",
              d when others;
```


© 2021 All Rights Reserved.

```
    end process;
  end arch;
```

Slices in Aggregate Phase 2 [9.3.3]

An aggregate of an n-dimensional array type, where “n” is greater than 1, is written as a one-dimensional aggregate. The index subtype of this one dimensional aggregate is provided by the first index position of the array type. While the expression specified for each element association is an (n–1)-dimensional array or array aggregate, called a subaggregate. A string or bit string literal is allowed as a subaggregate in place of any aggregate of a one-dimensional character type array.

There can be three types of slices in aggregates:

Example of slices with subaggregate and element can be written as follows:

```
library ieee;
use ieee.std_logic_1164.all;
package pkg is
  type DUMMY0 is array(integer range <>) of std_logic_vector(3 downto 0);
end package;

library ieee;
use ieee.std_logic_1164.all;
use work.pkg.all;
entity test is
 port (
  x : in DUMMY0(0 to 2);
  y : in std_logic_vector(3 downto 0);
  z : out DUMMY0(0 to 3) );
end test;

architecture module of test is
begin
 z <= (x, y);
end module;
```

Example of slices with nested subaggregate can be written as follows:

```
library ieee;
use ieee.std_logic_1164.all;
package pkg is
  type DUMMY0 is array(integer range <>) of std_logic_vector(7 downto 0);
end package;

library ieee;
use ieee.std_logic_1164.all;
use work.pkg.all;
entity test is
  port (
   x : in std_logic_vector(3 downto 0);
   z : out DUMMY0(0 to 2));
end test;
architecture module of test is
```


© 2021 All Rights Reserved.

```
begin
  z <= ((x,x) , (x,x) , (x,(x)) );
end module;
```

Example of slices with nested subaggregate and operator can be written as follows:

```
library ieee;
use ieee.std_logic_1164.all;
package pkg is
  type DUMMY0 is array(integer range <>) of std_logic_vector(7 downto 0);
end package;
```

```
library ieee;
use ieee.std_logic_1164.all;
use work.pkg.all;
entity test is;
  port (
   x1 : in std_logic_vector(3downto 0);
   x2 : in std_logic_vector(1 downto 0);
   x3 : in std_logic_vector(2 downto 0);
   z : out DUMMY0(0 to 2));
end test;
```

```
begin
 z <= ((x1,x1),
    (not(x2),x3,x3),
    (not(x2),x2,not(x1)));
end module;
```

Generic Function/ Procedure

VHDL-2008 LRM, section 4 Subprograms support generic list within subprogram declarations. Subprogram declarations declare a procedure or a function.

Note: See section "Subprograms" on page 557 for more information.

HDL Compiler partially supports generic list within subprogram declarations as the following are not supported:

- Generic type is not supported yet.
- A generic reference to another generic declared in the generic list is also not supported yet.

A procedure with a generic list takes the following form:

```
procedure identifier
generic (...)
parameter (...) is
       declarations
begin
       statements
end procedure identifier;
```

A function with a generic list takes the following form:

© 2021 All Rights Reserved.

```
function identifier
generic (...)
parameter (...) return result_type is
       declarations
begin
       statements
end function identifier;
```

If the subprogram header contains the reserved word generic, a generic list, and no generic map aspect, the subprogram is called an uninstantiated subprogram. In case of uninstantiated subprograms, the generic list is part of the subprogram specification. Thus, if we separate the declaration, we must write the generic list and parameter list in the specification, and then repeat both together with the body. Uninstantiated subprogram cannot be called directly.

For a procedure, the instantiation takes the following form:

```
procedure identifier is new uninstantiated_procedure_name
generic map (...);
```

For a function, the instantiation takes the following form:

```
function identifier is new uninstantiated_function_name
generic map (...);
```

In both cases, the identifier is the name for the subprogram instance, and the generic map supplies actual generics for the formal generics defined by the uninstantiated subprogram. If all of the formal generics have defaults, we can omit the generic map to imply use of the defaults. Once we have instantiated the subprogram, we can then use the instance name to call the instance.

Following example illustrates how to instantiate a generic function/procedure:

```
library ieee;
use ieee.std_logic_1164.all ;
package pkg is
   function graycode_bin_to_gray
      generic (W : integer  := 7)
      parameter (x : in std_logic_vector(W-1 downto 0))
   return std_logic_vector;
end pkg ;

package body pkg is
  function graycode_bin_to_gray
      generic (W : integer := 7 )
      parameter (x : in std_logic_vector(W -1 downto 0))
  return std_logic_vector is
     variable graycode_bin_to_gray_DUMMY0 : std_logic_vector(W-1 downto 0) ;
```


© 2021 All Rights Reserved.

```
  begin
      graycode_bin_to_gray_DUMMY0 := x NOR (X SLL 1) ;
      return graycode_bin_to_gray_DUMMY0;
  end graycode_bin_to_gray;
end pkg ;

library ieee;
use work.pkg.all;
use ieee.std_logic_1164.all ;
entity test is
  port (
    o1 : out std_logic_vector(31 downto 0) ;
    i : in std_logic_vector(31 downto 0) ;
    o2 : out std_logic_vector(3 downto 0) ;
    i2 : in std_logic_vector(3 downto 0) ;
  clk : in std_logic ) ;
end test ;

architecture module of test is
  function graycode_bin_to_gray_32 is new work.pkg.graycode_bin_to_gray
     generic map(W=>32) ;
  function graycode_bin_to_gray_4 is new work.pkg.graycode_bin_to_gray
     generic map(W=>4) ;
begin
  o1 <= graycode_bin_to_gray_32(i) ;
  o2 <= graycode_bin_to_gray_4(i2) ;
end module;
```

Slices in Array Aggregates [9.3.3.3]

Slices in array aggregates are defined in VHDL-2008 LRM, section 9.3.3.3 Array aggregates.

In earlier versions of VHDL, an aggregate could only be formed from individual elements. In VHDL-2008, the rules are extended to allow form an aggregate from a mixture of individual elements and slices of the array. For example, a bit_vector array aggregate could be written as follows:

```
    ('0', "1001", '1')
```

This forms a 6-element vector from the single element '0', the vector value "1001" and the single element '1'. The vector value forms a slice of the final aggregate value. The effect is similar to concatenating the element and array values.

© 2021 All Rights Reserved.

A signal assignment target can be wrote in the form of an aggregate of signal names, the names can be a mixture of element-typed signals and array-typed signals. Elements of the right-hand-side value are assigned to the matching signals or signal elements. For example:

```
signal a, b, sum : unsigned(15 downto 0);
signal c_out : std_ulogic;
...
(c_out, sum) <= ('0' & a) + ('0' & b);
```

Named association also can be used in array aggregates. If one want to include an association for a slice of the aggregate, a range of index values can be specified for the slice. For example:

```
variable status_reg : bit_vector(7 downto 0);
variable int_priority, cpu_priority : bit_vector(2 downto 0);
variable int_enable, cpu_mode : bit;
( 2 downto 0 => int_priority,
6 downto 4 => cpu_priority,
3 => int_en, 7 => cpu_mode ) := status_reg;
```

Slices in array aggregates are not supported from Conformal and CAN’T be verified using Conformal 14.10.100. We need to write testbench and verify the result netlist using simulator, or write function equivalent test case for verifying with Conformal 14.10.100.

Sample code:

```
entity test is
  port (
         A, B : in unsigned(7 downto 0);
         Y: out unsigned(0 to 7);
         CarryOut: out std_logic
        );
end test;
architecture a of test is
begin
  (CarryOut, Y) <= ('0' & a) + ('0' & b);
end;
```

Context Declarations [13.3]

Context Declarations is introduced in VHDL-2008 LRM, section, 13.3 Context Declarations.

VHDL-2008 provides a new form of design unit, a context declaration, in which people can gather a collection of library and use clauses. A context declaration can be refered to before

© 2021 All Rights Reserved.

a design unit, rather than having to repeat the collection of library and use clauses. The form of a context declaration is

```
context identifier is
... -- library clauses, use clauses and context references
end context identifier;
```

Within a context declaration, library and use clauses can be wrote in the same form as in a context clause preceding a design unit. A declared context can be refered to with a context reference of the form

```
    context context_name;
```

or, if one wish to refer to several context declarations:

```
    context context_name, context_name, ...;
```

Context reference can be wrote in the context clause preceding a design unit, or nested within another context declaration. In each case, the context reference is equivalent to replacement by the list.

Context Declarations are not supported from Conformal and CAN'T be verified using Conformal 14.10.100. We need to write testbench and verify the result netlist using simulator, or write function equivalent test case for verifying with Conformal 14.10.100.

Sample code:

```
context ctx_rtl is
  library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;
end context ctx_rtl;

library work;
context work.ctx_rtl;
entity test is
  port ( en,CI: in std_logic;
         A, B : in unsigned(7 downto 0);
         Y    : out unsigned(7 downto 0);
         CO   : out std_logic
        );
end test;

architecture a of test is
    signal Y9: unsigned(8 downto 0);
begin
```


© 2021 All Rights Reserved.

```
  Y9 <= ('0' & A) + ('0' & B) + CI ;
  Y <= Y9(7 downto 0);
  CO <= Y9(8);
end;
```

RISING_EDGE and FALLING_EDGE Functions (section 16.3 and 5.2.6)

RISING_EDGE and FALLING_EDGE functions have been introduced in VHDL-2008 LRM, section, 16.3 and 5.2.6.

```
function RISING_EDGE (signal S: BOOLEAN) return BOOLEAN;
function FALLING_EDGE (signal S: BOOLEAN) return BOOLEAN;
```

The function RISING_EDGE applied to a signal S of type BOOLEAN is TRUE if the expression“S'EVENT and S” is TRUE, and FALSE otherwise. The function FALLING_EDGE applied to a signal S oftype BOOLEAN is TRUE if the expression “S'EVENT and not S” is TRUE, and FALSE otherwise.

```
function RISING_EDGE (signal S: BIT) return BOOLEAN;
function FALLING_EDGE (signal S: BIT) return BOOLEAN;
```

The function RISING_EDGE applied to a signal S of type BIT is TRUE if the expression “S'EVENT and S = '1'” is TRUE, and FALSE otherwise. The function FALLING_EDGE applied to a signal S of type BIT is TRUE if the expression “S'EVENT and S = '0'” is TRUE, and FALSE otherwise.

Note: Rising_edge and Falling_edge are not supported from Conformal and cannot be verified using Conformal 14.10.100. Write the testbench and verify the result netlist using simulator, or write a function-equivalent test case for verifying with Conformal 14.10.100.

Sample code:

```
entity counter is
    port(clk : in bit;  -- or boolean
         digit: out integer range 0 to 9);
end counter;

architecture counter of counter is
begin
    count:process (clk)
        variable temp: integer range 0 to 10;
    begin
        if rising_edge(clk) then   --  or falling_edge
            temp := temp+1;
            if (temp=10) then temp :=0 ;
```


© 2021 All Rights Reserved.

```
            end if;
        end if;
        digit <= temp;
    end process count;
end counter;
```

MINIMUM and MAXIMUM Functions (section 16.3 and 5.3.2.4)

Minimum and Maximum functions have been introduced in VHDL-2008 LRM, section, 16.3 and 5.3.2.4.

HDL Compiler now supports Minimum/Maximum function for bit, boolean, integer, bit_vector, boolean_vector and integer_vector. Minimum and Maximum are defined for all scalar, discrete array types, and numeric std_logic type.

```
function minimum (L, R: T) return T;
function maximum (L, R: T) return T;
```

The MINIMUM function returns the value of L if L < R, or the value of R otherwise. The MAXIMUM function returns the value of R if L < R, or the value of L otherwise. For both operations, the comparison is performed using the predefined relational operator for the type.

Minimum and Maximum are also defined for single dimensional array types, AT, whose element type, ET, is a scalar.

```
function minimum (L: AT) return ET;
function maximum (L: AT) return ET;
```

The values returned by these functions are defined as follows.

— The MINIMUM operation returns a value that is the least of the elements of L. That is, if L is a null array, the return value is E'HIGH. Otherwise, the return value is the result of a two-parameter MINIMUM operation. The first parameter of the two-parameter MINIMUM operation is the leftmost element of L. The second parameter of the two-parameter MINIMUM operation is the result of a single-parameter MINIMUM operation with the parameter being the rightmost (L'LENGTH – 1) elements of L.

— The MAXIMUM operation returns a value that is the greatest of the elements of L. That is, if L is a null array, the return value is E'LOW. Otherwise, the return value is the result of a two-parameter MAXIMUM operation. The first parameter of the two-parameter MAXIMUM operation is the leftmost element of L. The second parameter of the two-parameter MAXIMUM operation is theresult of a single-parameter MAXIMUM operation with the parameter being the rightmost (L'LENGTH – 1) elements of L.

Sample code:

© 2021 All Rights Reserved.

```
entity MINMAX_COMP is
  generic (N: integer := 3);
  port (
        a_bit    : in bit;
        b_bit    : in bit;
        a_bit_v  : in bit_vector(0 to N);
        b_bit_v  : in bit_vector(0 to N);
        Z_bit_1  : out bit;
        Z_bit_1c : out bit;
        Z_bit_2  : out bit;
        Z_bit_2c : out bit;
        Z_bit_v1 : out bit_vector(0 to N);
        Z_bit_v1c: out bit_vector(0 to N);
        Z_bit_v2 : out bit_vector(0 to N);
        Z_bit_v2c: out bit_vector(0 to N));
end MINMAX_COMP;
architecture STRUCT of MINMAX_COMP is
begin
  process (all)
  variable v1 : bit_vector(0 to N);
  variable v2 : bit_vector(0 to N);
  variable v3 : bit_vector(0 to N);
  begin
    v1 := b"0101";
    v2 := b"0000";
    v3 := b"1111";
    Z_bit_1   <= MINIMUM(a_bit_v);
    Z_bit_v1  <= MINIMUM(a_bit_v,b_bit_v);
    Z_bit_1c  <= MINIMUM(v1);
    Z_bit_v1c <= MINIMUM(a_bit_v,v2);
    Z_bit_2   <= MAXIMUM(a_bit_v);
    Z_bit_v2  <= MAXIMUM(a_bit_v,b_bit_v);
    Z_bit_2c  <= MAXIMUM(v1);
    Z_bit_v2c <= MAXIMUM(a_bit_v,v3);
  end process;
end STRUCT;
```

Unconstrained Element Types (section 5.1 and 6.3)

Unconstrained Element Type have been introduced in VHDL-2008 LRM, section, 5.1 and 6.3.

© 2021 All Rights Reserved.

HDL Compiler now partially supports unconstrained element type for array composite types, while record type is not included.

HDL Compiler only supports array subtype with no index constraint and the element subtype is either not a composite subtype or is an unconstrained composite type. Unconstrained record subtype is not supported.

A composite subtype is said to be fully constrained if

- It is an array subtype with an index constraint and the element subtype is either not a composite subtype or is a fully constrained composite type, or
- It is a record subtype and each element subtype is either not a composite subtype or is a fully constrained composite subtype.

A composite subtype is said to be partially constrained if it is neither unconstrained nor fully constrained.

In IEEE 1076-1987 and 1076-1993, types and subtypes had to be fully constrained in the declaration of the type.

In IEEE 1076-2008, types and subtypes can be unconstrained and the constraining happens when there are objects which have been defined of that type. For example:

```
   type my_partially_constrained1 is array (natural range <>) of std_logic_vector(0
to 7);
   type my_partially_constrained2 is array (natural range <>) of std_logic_vector(0
to 3);
    --subtype S1 is my_partially_constrained1(0 to 1);
    --subtype S2 is my_partially_constrained2(0 to 1);
signal reg8 : my_partially_constrained1(0 to 1);
signal reg4 : my_partially_constrained2(0 to 1);
```

In IEEE 1076-1987 and 1076-1993, the signal `reg8` and `reg4` must have been accompanied with two types (`my_partially_constrained1`, `my_partially_constrained2`). In IEEE 1076-2008, the signal `reg8` and `reg4` can be accomplished with one type (`my_unconstrained`). This can also be done for arrays as shown in the following example:

```
type my_unconstrained is array (natural range <>) of std_logic_vector;
    --subtype S3 is my_unconstrained(0 to 1)(0 to 7);
    --subtype S4 is my_unconstrained(0 to 1)(0 to 3);
    signal reg8 : my_unconstrained(0 to 1)(0 to 7);
signal reg4 : my_unconstrained(0 to 1)(0 to 3);
```


© 2021 All Rights Reserved.

Alias Declarations (section 6.6)

VHDL now supports writing an alias for an object and also specifying a subtype with which to view the object. For example:

```
signal s : bit_vector(15 downto 0);
alias bigendian_s : bit_vector(0 to 15) is s;
```

In earlier versions of VHDL, the rules for aliases of arrays were unclear and prohibited aliases of multidimensional array objects. VHDL-2008 clarifies the rule and removes the restriction making the following example legal:

```
type bit_matrix is
array (natural range <>, natural range <>) of bit;
signal s : bit_matrix(15 downto 0, 15 downto 0);
alias bigendian_s : bit_matrix(0 to 15, 0 to 15) is s;
```

Deviations from VHDL 2008 LRM

This section specifies the deviations from IEEE Standard VHDL LRM 2008.

Using Unparenthesized Unary Logical Operators

According to IEEE Standard VHDL LRM 2008, section 9.1, unary operator and its operand should be a parenthesized expression. So, the following example is not allowed. However, other compiler tools, such as RTL Compiler and Conformal allow such unparenthesized unary logical operators. To maintain compatability with such tools, HDL Compiler also permits unparanthesized unary logical operators.

```
library ieee;
use ieee.std_logic_1164.all;
entity test is
port(a : in std_logic_vector(3 downto 0);
     b : in std_logic;
     c : out std_logic );
end test;
architecture arch of test is
begin
     c <= and a and b ;
end arch;
```


© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

16

Compiling Designs for Simulation Acceleration Using IXCOM

In the **I**ncisive **X**accelerator **COM**piler (IXCOM) flow, the `vlan`, `vhan`, `ixcom` tools are used during the design analysis and compilation phases. These tools generate the intermediate database provided to *xrun* for compilation. The resultant database is referred as the Xcelium or *xrun* snapshot. This snapshot can be run in Software Simulation mode and Simulation Acceleration mode.

In the **Software Simulation mode**, the complete design is run on the *xrun* simulator.

To compile a **VHDL or mixed-language** design for Software Simulation mode, first analyze the Verilog and VHDL source files using the `vlan` and `vhan` tools, respectively. Then, run the `ixcom` command, specifying the analyzed source files and relevant command-line options. IXCOM passes all input to the `vhelab` tool for design elaboration.

To compile a **Verilog-only** design for Software Simulation mode, provide the source files and command-line options with the `ixcom` command. IXCOM invokes the required tools for:

- Source analysis and design elaboration
- Source transformation and code generation
- Writing the relevant design information to the required internal databases

Finally, IXCOM runs `make` to generate the *xrun* or Xcelium snapshot.

In the **Simulation Acceleration mode**, one part of the design is run on the *xrun* simulator and the other part is run on the emulator. In addition to all the processing that the IXCOM compiler does for the Software Simulation mode, in the Simulation Acceleration mode, IXCOM also invokes the back-end compilers to generate the emulator configuration data. The back-end compilation process is described in Chapter 9, “Compiling Designs for In-Circuit Emulation with xeCompile.”

© 2021 All Rights Reserved.

Important

The content of this chapter pertains only to SA mode.

The following diagram illustrates the flows for compiling and running Mixed-HDL designs using IXCOM flow in both Software Simulation and Simulation Acceleration modes. The steps specific to the Simulation Acceleration mode are marked in blue colored dotted lines.

Figure 16-1 Compiling and Running Mixed-Language Designs

© 2021
All Rights Reserved.

Overview of the IXCOM Compilation Process

The compile methodology of IXCOM is similar to that of other compiled-code simulators. It consists of multiple steps (multi-step compilation) for a mixed-HDL or VHDL design and a single step (single-step compilation) for a Verilog-only design:

For a mixed-HDL or VHDL Design, use the multi-step compilation process:

1. Analyze the VHDL and Verilog parts of the design separately. (Use `vlan` and `vhan` commands.) See Analyzing Mixed-Language Designs on page 609.
2. Compile the entire design to generate the simulation snapshot. (Use `ixcom` command.) See Compiling Mixed-Language Designs on page 626.

For a Verilog-only Design, use the single-step compilation process:

➡ Compile the entire design to generate the simulation snapshot. (Use `ixcom` command only; separate `vlan` and `vhan` steps are not required.)

The `ixcom` command supports various `vlan` command options to enable single-step compilation. When using multi-step compilation, do not use the options categorized as Analyzer/Compiler options with the `ixcom` command. Specify these options with the `vlan` command instead.

After the design has been compiled, use the resulting snapshot to run the design and debug it for errors and improving performance.

Important

The snapshot is platform-specific. This means that if the compilation was done on Linux, the snapshot would be Linux-specific, and at run time, the *xrun* and xeDebug tools accessing this snapshot should also be run on Linux.

This chapter contains the following topics on using IXCOM to analyze and compile your design:

- Analyzing Mixed-Language Designs
- Compiling Mixed-Language Designs
- Compiling for Hardware Using IXCOM
- Compiling SystemC Designs
- Using Specman Elite with IXCOM
- Compiling Testbenches Separately

© 2021 All Rights Reserved.

- Determining File Type by Source Filename Extension
- Compiling Behavioral Constructs
- Compiling Behavioral Interface Constructs
- Fine-Tuning Behavioral Performance
- System Task Mapping in IXCOM flow
- Executing Non-Synthesizable Code in the Emulator
- Generating Pulses
- Pragma Handling in IXCOM
- Reading and Writing Signals in DUT Modules
- Using Plus Arguments
- Support for Encryption in SA Mode
- Support for Strengths in SA Mode
- Handling of UDPs by IXCOM
- Support for Verilog Configuration Blocks
- Support for Overriding VHDL Generics from the IXCOM Command Line
- Support for specify Blocks in SA Mode
- Enabling Run-time Access of For-generate Blocks and Instance Arrays Using Original RTL Names
- Direct Access to DUT Memory Arrays and Registers
- Using the marg Library
- C and Verilog Interface marg Functions
- SystemVerilog DPI marg Functions
- Specman marg Functions
- Importing External Netlists in the SA Mode

© 2021 All Rights Reserved.

Analyzing Mixed-Language Designs

To analyze a mixed-HDL design, perform the following steps:

1. Set up a `libmaprc` file to associate the desired physical libraries to logical library names, for example:

   ```
   VERILOG => /asic/verilog/verilog_lib
   ```

   See Associating Logical Libraries to Physical Directories Using libmaprc File on page 617 for more detail.

2. Analyze the Verilog sources to one or more library using `vlan`. (The default library is `WORK`.)

Tip

`IXCOM_COMPILE` is a predefined or built-in macro that IXCOM always evaluates to be true:

```
`ifdef IXCOM_COMPILE
  initial $display("IXCOM flow");
`else
.........
`endif
```

You need not use `define IXCOM_COMPILE 1`.

3. Analyze the VHDL sources to one or more library using `vhan`. (The default library is `WORK`.)

You can mix VHDL and Verilog throughout a design, with a few caveats. These caveats are described as guidelines in the following sections:

- Instantiating Verilog in VHDL
- Instantiating VHDL in Verilog

Instantiating Verilog in VHDL

Instantiating a Verilog module in a VHDL architecture is like instantiating any other VHDL component: write a component declaration for the Verilog module, then instantiate the component. Use the `vhshell` utility to automatically generate the component declaration for the Verilog. (Refer to vhshell on page 104 for more information on this utility.)

© 2021 All Rights Reserved.

Binding a VHDL Instance to a Verilog Configuration

A component instance or direct instance in a VHDL module can be bound to a Verilog module by using any of the bindings, such as direct instance, component instance using default binding, using a configuration specification, or using a configuration declaration.

A component instance in VHDL can be bound to a Verilog configuration by using either a configuration specification in a VHDL architecture or a VHDL configuration declaration.

Consider the following examples that illustrate the binding of a component instance in VHDL to a Verilog configuration.

// Verilog configuration

```
config vlg_conf;
    design lib1.vlgMod;
    default liblist lib1 lib2 ;
endconfig
```

// VHDL component instance

```
Inst1 : vlgMod generic map (g1, g2)  port map (p1, p1);
```

// VHDL configuration specification

```
for inst1 : vlgMod use configuration lib1.vlg_conf;
```

Keep the following points in mind while binding a component instance in VHDL to a Verilog configuration:

- A VHDL instance cannot be bound to a Verilog configuration using direct instantiation, such as the following:

  ```
  // vlgMod cannot be a Verilog configuration. It must be VHDL configuration.
  Inst1 : configuration vlgMod generic map (g1, g2)  port map (p1, p1);
  ```

- A VHDL instance cannot be bound to a Verilog configuration through a component instance by using the default binding, such as the following:

  ```
  // vlgMod can not be a Verilog configuration. It must be a VHDL entity.
  Inst1 : vlgMod generic map (g1, g2)  port map (p1, p1);
  ```

- Use of Verilog escaped names in VHDL is not supported.

Instantiating VHDL in Verilog

IXCOM supports the defacto_standard escaped identifier mechanism to explicitly bind a VHDL component instantiated in a Verilog module:

```
\library_name.configuration instance-name (port_list);
\library_name.entity(architecture) instance-name (port_list);
```


© 2021 All Rights Reserved.

The VHDL component is specified either with a *configuration* or an *entity(architecture)* pair. If the *(architecture)* parameter is omitted, the last compiled architecture is used. The <*library_name*> specifies the logical library containing the VHDL design unit. By default, it is `WORK`. To change the <*library_name*>, specify it with the `-work` option.

For more detail, see the following sections:

- Searching Specific Libraries Using the -L library Search Switch
- Searching Specific Libraries Using the ‘uselib Compiler Directive for Verilog
- Design Binding and Search Rules in IXCOM
- Cadence xmsim-Compatible Search for IXCOM
- Instantiating VHDL Design Units in Verilog
- Mapping VHDL to Verilog Values
- Mapping Verilog to VHDL Values

Searching Specific Libraries Using the -L library Search Switch

As an alternative to the previous syntax, you can specify just the VHDL entity name in a standard Verilog instantiation. Then, pass one or more `-L` *`library`* options to `ixcom` to search a set of analyzed libraries for the desired component. This option can be used to find any component (VHDL or Verilog) not explicitly bound to a particular design unit. The compiler will first search the library of the design unit that instantiated the component, then search the list of `-L` libraries.

For example, if you have two components instantiated as follows in Verilog:

```
\ramlib1.sram(behav) s1 (Clk, Addr, Data, WE, Q1);
sram s2 (Clk, Addr, Data, WE, Q2);
ixcom -top <top_level_design_unit> [other_compiler_options] -L ramlib2
```

A library (*`ramlib1`*) is specified in the instantiation of component *`s1`*, so only that library is searched. There is no library specified in the instantiation of component *`s2`*, so the library, *`ramlib2`* (specified with the `-L` *`ramlib2`* option), is searched.

Searching Specific Libraries Using the ‘uselib Compiler Directive for Verilog

The `` `uselib `` compiler directive specifies the libraries that should be searched for the definitions of modules and User Defined Primitives (UDPs) instantiated in a design description that does not include their definitions. Each `` `uselib `` directive explicitly defines the library search that resolves the instances that follow it until the compiler encounters another `` `uselib `` directive, which completely redefines the search. An empty `` `uselib ``

© 2021 All Rights Reserved.

directive makes the preceding `` `uselib `` directives ineffective. The functionality associated with these `` `uselib `` extensions is tied to the `cds.lib` and `hdl.var` files used by *xrun*.

For details about the `` `uselib `` compiler directive, refer to the "*The `uselib Compiler Directive*" section in the *Xcelium Direct Invocation: Verilog Compilation* User Guide available in the Xcelium documentation set.

IXCOM does not support the following IUS extensions of the `` `uselib `` syntax:

```
`uselib lib = <logical_lib_name>
`uselib view = <view_name>
```

You need to manually modify these *xrun*-supported `` `uselib `` extensions and convert these to the corresponding Verilog-XL style syntax that IXCOM supports, as follows:

```
`uselib dir = <dir_name>
`uselib file = <file_name>
`uselib libext = <ext_name>
```

Following is the mapping of `` `uselib lib `` to the corresponding `` `uselib dir `` extension:

The `lib` version takes the logical library name as an argument.

The `dir` version takes the name of the physical directory (corresponding to that logical directory name, as specified in the `cds.lib` file) as the argument.

The `` `uselib view `` extension maps to the `` `uselib libext `` extension based on the contents of the `cds.lib` file.

Design Binding and Search Rules in IXCOM

Binding is the process of selecting the design units that should be instantiated at each node of the hierarchy.

IXCOM uses the following rules for module binding:

- For the top-level module, IXCOM searches for the modules in the following order:
  - **a.** Verilog module
  - **b.** VHDL module
  - **c.** SystemC module
- For binding VHDL modules, IXCOM searches for the specified VHDL module in the following order:
  - **a.** In the current library

© 2021 All Rights Reserved.

b. In the parent library

c. In all the libraries specified in the `libmaprc` file, if `-relax` option is passed

d. In the libraries mentioned with the `use` clause

e. If no VHDL module is found, IXCOM searches for a Verilog module in the following order:

- In the parent library
- In all the libraries, starting with the first entry in the `libmaprc` file, if no Verilog module is found in the parent library

f. If no Verilog module is found, IXCOM searches for a SystemC module in the shared libraries specified with the `-loadsc` option.

Note: The *`current`* library refers to the library specified with the `-w` <*`library`*> option, and the *`parent`* library refers to the library that contains the parent module.

The following table summarizes the search order that IXCOM follows for locating the specified module:

Parent/ Child	VHDL/ VHDL	Verilog/ Verilog	Verilog/ VHDL	VHDL/ Verilog
Parent library	2	1	1	1
Current library	1	-	-	-
Set by `-L` option	-	3	2	2
Set by `libmaprc`	4	4	3	3
Set by `` `uselib ``	3	-	-	-
Set by the `-v` or `-y` option	-	2	-	-

Note:

- When parent/child is VHDL/VHDL, use the `-relax` option for using `libmaprc`. The `-L`, `-v`, or `-y` option cannot be used.
- `` `uselib `` is used only when parent/child is VHDL/VHDL.
- Declare all libraries when using the `-L` option.

© 2021 All Rights Reserved.

- If a library is declared through `libmaprc`, or the options, such as `-L`, `-v`, or `-y`, IXCOM sets the priority from left to right or from top to bottom.
- If the `-v` and `-y` option are used together, priority is set from left to right declaration.

- For binding Verilog modules, IXCOM searches for the specified Verilog module in the following order:
  a. In the parent library
  b. In all the libraries specified in the `libmaprc` file, if `-relax` option is passed
  c. Searches for a VHDL module if no Verilog module is found, in the following order:
    - In the parent library
    - In all the libraries specified in the `libmaprc` file
  d. Searches for a SystemC module if no VHDL module is found in the parent library.
- For binding SystemC modules, IXCOM:
  - Searches for the specified SystemC module in the parent library.
  - Implements the same bind routine implemented for a Verilog module.

  Note: In case of a SystemC module, the parent and the current libraries are same.
- When `-L` *`<library>`* option is passed, then IXCOM searches for:
  - The VHDL module in the library(s) specified with the `-L` option.
  - A Verilog module if no VHDL module is found, in the library(s) specified with the `-L` option.
  - The Verilog module in the library(s) specified with the `-L` option.
  - A VHDL module if no Verilog module is found, in the library(s) specified with the `-L` option.

  Note: You can specify multiple libraries with the `-L` option.
- When the `-relax` option is passed to IXCOM, the VHDL modules are searched in all the libraries, starting with the first entry in the `libmaprc` file.

© 2021 All Rights Reserved.

IXCOM Adhering to the Xcelium Search Order for Binding

For default binding with respect to instance binding or the library search order, IXCOM follows the Xcelium search order for the process of selecting a particular binding unit if you specify the `-iesDefaultBinding` option of the `ixcom` command.

Note: If the component is not bound, IXCOM generates an error message while Xcelium generates a warning.

Search Order for SystemVerilog Top-Level Modules

The following list defines the library-search order followed for the SystemVerilog top-level modules as defined in Xcelium tool when the `-iesDefaultBinding` option is used:

1. If the module is not already loaded, the libraries are searched. If the module is present, it is used, else the libraries are searched in the following order:
2. Looks for the first library as follows:
   a. If no library has been specified with the `-L` option, the first library is set to parent library.
   b. If the parent library exists in the libraries specified with the `-L` option, first library is set to the parent library.
   c. Else, first library is set to the first library specified with the `-L` option.
3. Searches in the first library as found above. If the module is found, uses it and does not search further.
4. Looks up the libraries specified with the `-L` option. f the module is not found, and libraries have been specified with the `-L` option, starts the search from the library next to the first library set above in a cyclic fashion. If it finds the module, uses it, and does not search further.
5. Continues to look up in the `libmaprc` file. If the module is not found, and libraries have been specified with the `-L` option, scans the libraries in the `libmaprc` file.
   - Else if, multiple definitions are found, generates an error message,
   - Else uses the found definition and exits the look-up process.
6. Tries to find the module in the `libmaprc` file. If it is not able to find it, and the `-L` option is not specified, it searches in libraries in the `libmaprc` file, from library next to the first library set above in a cyclic fashion. First, the SystemVerilog modules in all libraries are looked up, and then the VHDL units are searched.

© 2021 All Rights Reserved.

If the module is found, uses it, else an error is generated.

Search Order for VHDL Parent Modules

Following are the steps for library search when parent module is VHDL. To bind component instances to compiled design units in the libraries, the elaborator:

1. Uses explicit binding indications
2. If no explicit binding indication exists, the elaborator tries to bind the component to the following design units in the specified order:
   a. A design unit made visible with a USE clause given to the architecture instantiating the component.
   b. A design unit made visible with a USE clause given to the entity of the architecture instantiating the component.
   c. A design unit available in the library into which the component was compiled.
   d. A design unit in the work library.

   The `ixcom -relax` option can also be used to relax the strict default binding search order. The search order used with the `-relax` option is the same as mentioned in step 2. The `-relax` option also searches for a design unit in a library defined in the `libmaprc` file — that is, if a binding has not been found, the elaborator opens the `libmaprc` file and searches all the libraries that are defined in the file that have not been searched.
3. If a module is not found in VHDL, the elaborator tries to bind the component to a Verilog module in the following order:
   a. Selects a design unit available in the library into which the component was compiled.
   b. Selects a design unit made visible with a USE clause given to the architecture instantiating the component.
   c. Selects a design unit made visible with a USE clause given to the entity of the architecture instantiating the component.
   d. Selects a design unit made visible through library mapping defined in `libmaprc`. If, however, a design unit is available in more than one libraries, the elaborator would error out.

© 2021 All Rights Reserved.

Cadence xmsim-Compatible Search for IXCOM

The `-ncsim -xmsim` option with ixcom can be used to enable Cadence xmsim-compatible searching of entities and modules not found in `-L` libraries. Libraries specified in the `libmaprc` file are searched for VHDL or Verilog components not found in libraries specified with `-L` options. Libraries are searched in the order specified in the `libmaprc` file.

Instantiating VHDL Design Units in Verilog

VHDL design units can be instantiated in Verilog only if the VHDL ports that connect to Verilog signals are of supported types and vectors of these types. Other data types cause errors at compile time.

Note: The VHDL signals cannot be forced from Verilog modules.

Mapping VHDL to Verilog Values

When a VHDL signal drives a Verilog port (`input` or `inout` to Verilog), the VHDL values are mapped to the Verilog values. The mapping of these values is described in detail in the *Xcelium Direct Invocation: Verilog Compilation User Guide* and the *Xcelium Direct Invocation: VHDL Compilation User Guide*. Both of these user guides are provided in the Xcelium documentation set.

Mapping Verilog to VHDL Values

When a Verilog signal drives a VHDL port (`input` or `inout` to VHDL), the Verilog values are mapped to the VHDL values. The mapping of these values is described in detail in the *Xcelium Direct Invocation: SImulation* User Guide provided in the Xcelium documentation set.

Associating Logical Libraries to Physical Directories Using libmaprc File

The logical library concept, an essential aspect of the mixed-HDL design flow, gives you a way to organize VHDL and Verilog design units into directory structures that can be easily referenced in VHDL statements and read by downstream tools.

The `libmaprc` file associates a logical library name with a physical directory path containing the actual analyzed design units. You can use one or more `libmaprc` files—the files are all read to generate a list of logical library definitions referenced during analysis and compilation. `libmaprc` files are required by the vhan analysis tool, the ixcom compiler, and other utilities like vhls.

© 2021 All Rights Reserved.

libmaprc File Syntax

Here is the basic syntax for the `libmaprc` file:

```
<logical-library-name> => <physical-directory-path>
```

<*logical-library-name*> is not case sensitive, but <*physical-directory-path*> is case sensitive. For example:

```
RAMLIB => ./axis_compile_libs/ramlib
```

The logical library name RAMLIB (not case sensitive) is referenced in VHDL source code or in `-work` options to various mixed-HDL tools.

Note: The last directory in each directory path does not need to exist—it will be created automatically.

Comments in the libmaprc File

The **-** and **#** characters are used to delineate a comment in the `libmaprc` file. For example:

```
RAMLIB => ./axis_compile_libs/ramlib -- A comment
# Another comment
```

include Keyword in the libmaprc File

Use the `include` keyword in your `libmaprc` file to include the contents of another `libmaprc` file at a specified point in the file. For example:

```
RAMLIB => ./axis_compile_libs/ramlib
-- Following statement includes contents of another libmaprc file
include /net/designs/ASIC/romlibs/libmaprc
```

Using libmaprc Files

The easiest way to use a libmaprc file is to copy the *<IXCOM_INSTALL_HOME>/share/vxe/*`packages/libmaprc` file to your local project directory; then add your own libraries to the file. The `libmaprc` file in the local directory will be found first by mixed-HDL tools.

Alternatively, you can specify a particular `libmaprc` file using the `-libmaprc` option to vhan, vlan, ixcom, or other mixed-HDL tool:

```
-libmaprc <libmaprc_file>
```

You can specify multiple `-libmaprc` options. The *libmaprc_file(s)* are read in the order specified to generate a list of logical libraries.

© 2021 All Rights Reserved.

Note: If the same logical library is listed more than once in the `libmaprc` files, the first entry will take precedence.

Searching for libmaprc Files

If no `-libmaprc` options are specified, specific directory locations are searched for `libmaprc` files, in the following order:

1. Current directory
2. `$HOME` environment variable directory
3. $~~XE_INSTALL_DIR~~IXCOM_INSTALL_HOME environment variable directory

Note: All `libmaprc` files found in these locations are read, in order to generate a list of logical libraries. If the same logical library is listed more than once in the `libmaprc` file, the first entry will take precedence.

Default libmaprc File

The default `libmaprc` file is located in the $~~XE_INSTALL_DIR~~IXCOM_INSTALL_HOME/`packages` directory. Here are the contents of this file:

```
STD => $XE_INSTALL_DIRIXCOM_INSTALL_HOME/packages/lib/STD
IEEE => $XE_INSTALL_DIRIXCOM_INSTALL_HOME/packages/lib/IEEE
SYNOPSYS => $XE_INSTALL_DIRIXCOM_INSTALL_HOME/packages/lib/SYNOPSYS
AXIS => $XE_INSTALL_DIRIXCOM_INSTALL_HOME/packages/lib/AXIS
NCUTILS => $XE_INSTALL_DIRIXCOM_INSTALL_HOME/packages/lib/NCUTILS
```

This file contains logic library associations for VHDL system libraries.

You can copy this file to your project directory and add your own design libraries as follows, for example:

```
WORK => ./axis_compile_libs/work
```

Alternatively, use the `include` keyword to include the <~~*XE_INSTALL_DIR*~~IXCOM_INSTALL_HOME>/`packages/libmaprc` file.

For example:

```
include <XE_INSTALL_DIRIXCOM_INSTALL_HOME>/packages/libmaprc
<your_libraries_here>
```

System Libraries Referenced in the libmaprc File

The software installation includes the following VHDL system libraries, which are referenced in the default `libmaprc` file:

© 2021 All Rights Reserved.

- STD
- IEEE
- SYNOPSYS
- AXIS
- NCUTILS

The default compiled system libraries are located in:

<~~XE_INSTALL_DIR~~IXCOM_INSTALL_HOME>/packages/lib

The source code for the system libraries is located in:

<~~XE_INSTALL_DIR~~IXCOM_INSTALL_HOME>/packages/src

Associating Multiple Logical Libraries to Same Physical Directory

You can associate more than one logical library to the same physical directory.

For example:

```
WORK => ./axis_compile_libs/work
RAMLIB => ./axis_compile_libs/work
```

Note: A design unit (VHDL or Verilog) analyzed to a physical directory will override any design units that already exist in that directory, even if a different logical library was specified.

Working with Multiple libmaprc Files Using vhlib

The `vhlib` utility is useful when you are working with multiple `libmaprc` files. It provides a listing of all logical-library-to-physical-directory associations, as well as prints the `libmaprc` file from which each association was obtained. Here is the syntax for the `vhlib` utility:

```
vhlib [<logical_library_name>] [-libmaprc <libmaprc_file>] [-n]
```

If specified with no options, `vhlib` prints the logical-library-to-physical-directory associations of all `libmaprc` files specified with options or all `libmaprc` files found in the search order described in Searching for libmaprc Files on page 619. It also prints the full path to the `libmaprc` file in which each association was obtained.

If <*logical_library_name*> is specified, only the information for that particular library is specified.

© 2021 All Rights Reserved.

If `-libmaprc` *`libmaprc_file`* is specified, `vhlib` prints the logical-library-to-physical-directory associations in the specified *libmaprc* file only, rather than all `libmaprc` files found.

Specifying `-n` causes only the logical library association to be printed and not the `libmaprc` file information.

For example:

```
vhlib RAMLIB
..
-- file:  /net/designs/ASIC1/libmaprc
RAMLIB => /net/designs/ASIC1/ramlib
..
```

Converting cds.lib Files to libmaprc Format

The `cds.lib` files are used with Xcelium utilities, and have the following syntax for mapping logical libraries to physical libraries:

```
DEFINE <logical_library> <physical_library_path>
```

For example:

```
    DEFINE lib1 ./lib
    DEFINE lib2 ./pci_lib
```

The above syntax is a little different from that of the libmaprc file. (libmaprc File Syntax) Therefore, the `cds2libmap` utility is provided to help convert existing `cds.lib` files to equivalent libmaprc files.

To use the `cds2libmap` utility, use the following command:

```
cds2libmap [-h] <cdslib_file> [-o <output_file>]
```

The <*`output_file`*> is the equivalent libmaprc file.

See cds2libmap on page 97 for more information.

Note: `cds2libmap` utility can also be used to convert IXE specific `va.lib` files to equivalent libmaprc files.

Keep the following points in mind when using the `cds2libmap` utility:

- The `cds2libmap` utility needs to be run individually on all the relevant `cds.lib` files.
- For a particular `cds.lib` file, the `cds2libmap` utility generates a single libmaprc file that contains all the `DEFINE`s resolved from the `cds.lib` file and all the included files.

© 2021 All Rights Reserved.

- All the relative paths in the `cds.lib` file (and included `cds.lib` files) are resolved and dumped as absolute paths. Thus, if the design or part of the design is moved to a different location, the conversion utility must be run again on the `cds.lib` files.
- The `cds2libmap` utility resolves the standard locations of the predefined libraries, such as IEEE and SYNOPSYS in the `cds.lib` file to the corresponding location in the IXCOM hierarchy.
- Multiple definitions for the same logical library are resolved to a single definition according to the `cds.lib` resolution semantics. Thus, the last definition overrides the older definitions.

Optimizing Library File Processing

IXCOM parses library files and directories specified with the `-v` and `-y` options files, multiple times, to identify the undefined modules instantiated in the source or other library files. Depending upon the arrangement of the library files and directories, availability of undefined modules and underlined scan order, the compilation time can be significant.

You can speed up the compilation time by using the `-enableLibFileCaching` option of the `vlan` or `ixcom` command.

The arrangement of the library files and design size determine the improvement possible in compilation time, and the improvement is more in scenarios that require multiple iterations, and the files do not contain undefined module definitions at one or more iteration level.

Note: Ensure that the `-v` and `-y` options are specified with this option for optimizing compilation time.

Files containing compiler directive or macros, `` `define ``, `` `ifdef ``, `` `else ``, `` `elseif ``, `` `endif ``, `` `undef ``, or `` `undefall `` are not cached and these are parsed in all the scan passes.

Enabling Concurrent Execution of vlan Sessions

Large designs might be written in multiple analyzer sessions, which when executed sequentially require subsequent sessions to wait for the previous sessions to complete before starting. This results in a significant amount of analysis time. For improving compile time for large designs, you can enable concurrent execution of the analyzer sessions, only for `vlan`.

Note: This feature is available only for SystemVerilog designs in the multi-step compilation flow.

To enable the analyzer sessions to run concurrently, specify the `vlan` sessions to be run in parallel within a block using a set of `start` and `stop` commands as follows:

© 2021 All Rights Reserved.

- `vlan -parallelBlock start` marks the **start** of the **parallel block**.
- `vlan -parallelBlock stop` marks the **end** of the **parallel block**.

This feature is relevant for multiple `vlan` sessions with and without package dependency between sessions.

Considerations for Using Parallel Blocks

Keep the following points in mind regarding the parallel blocks:

- A parallel block cannot be specified in multiple shell sessions.
- A parallel block cannot span multiple terminals, shells, or scripts.
- Multiple parallel blocks can be created but these parallel blocks will execute sequentially.
- A parallel block containing a `vlan` command is not supported with the `-conformal` option that is used for the LEC flow.
- Parallel blocks cannot be nested.

Recommendations for Using Parallel Blocks

Try to balance each session in term of files and time. Having a single session to analyze most of the files and the rest of the sessions to analyze just a small part of the design will not give desired result.

As far as possible, create independent sessions with no interdependent packages. Then, each session runs independently, without blocking the execution.

Restrictions while Using Parallel Blocks

Concurrent execution of the `vlan` sessions is not supported in the following cases:

- Imported netlist flow
- When VHDL packages are used in SystemVerilog designs
- When VHDL analyzer (`vhan`) is used

  If a `vhan` command is included in the parallel block, the execution of the parallel block is halted until `vhan` execution is completed. If the `vhan` command refers to a package in the parallel block, the analysis process fails.

© 2021 All Rights Reserved.

Example for Using Parallel Blocks

Consider the following example.

The following commands run sequentially:

```
vlan -sv pkg.v foo.v
vlan -sv pkg2.v boo.v
vlan -sv pkg3.v koo.v
```

In the above commands, session 3 will start executing only after session 2 is over. Similarly, session 2 will start executing only after session 1 is over.

Add the following `vlan` commands to encapsulate the existing `vlan` sessions to create a block of sessions that can run concurrently:

vlan –parallelBlock start

```
vlan -sv pkg.v foo.v
vlan -sv pkg2.v boo.v
vlan -sv pkg3.v koo.v
```

vlan –parallelBlock stop

In the above example, sessions 2, 3, and 4 execute simultaneously.

By default, the log files are generated in the `WORK/64bit/_<sessionId>_vlan.log` directory. A `server.log` file is generated to log the status of a parallel block.

The `vlan` command in the parallel block creates the log file inside `WORK/64Bit/` as `_<sessionId>_vlan.log` if the `-l` command is not specified in the command line.

The following `vlan` sessions cannot execute in the same parallel block.

Commands to generate a library and use it in a `vlan` session cannot be written in the same parallel block. The following vlan commands in **bold** font cannot be in the parallel block.

```
$(VLAN) -m mapfile.v -work libdbs
$(VLAN) -parallelBlock start
$(VLAN) -sv top.v -z libdbs/64bit/ +debug
$(VLAN) -sv bot.v
$(VLAN) -sv dut.v
$(VLAN) -parallelBlock stop -l server.log
```

Following sequence of commands is not supported because nested parallel blocks are not allowed:

vlan -parallelBlock start

```
vlan -sv pkg.v foo.v
vlan -parallelBlock start
vlan -sv pkg2.v boo.v
vlan -sv pkg3.v koo.v
vlan -parallelBlock start
vlan -sv pkg3.v koo.v
```


© 2021 All Rights Reserved.

```
vlan -parallelBlock stop
```

Parallel Invocation of HDL-ICE Analyzer

For designs that have multiple libraries, and for which the HDL-ICE analysis sessions might take significant time, it is useful to run these analysis sessions in parallel and optimize the compilation time. IXCOM supports the parallel invocation of the analysis steps of HDL-ICE Compiler with the use of the `-parallelHdliceAnalyze [<nJobs>]` option of the `ixcom` command.

IXCOM generates the `vavlog` and `vavhdl` commands for each library and merges all the sessions of one library in a single session. Thus, all the modules and packages for a library are collected and analyzed with a single `vavlog` command.

With the `-parallelHdliceAnalyze [<nJobs>]` option, IXCOM invokes the above `vavlog` commands in parallel by using the following command:

```
make -j <nJobs> -f xc_work/xc_make vavlog_parallel
```

The <*nJobs*> argument is optional. If it is not specified, IXCOM compares the number of the available cores on the machine and the number of vavlog sessions that are required, and sets the lesser number as <*nJobs*>. If the specified <*nJobs*> value exceeds the number of available cores on the machine, the number of available cores on machine is set as <*nJobs*>.

If the `-parallelHdliceAnalyze` option is specified multiple times, IXCOM honors the last specification with a corresponding INFO message.

IXCOM generates a target rule, `vavlog_parallel`, in the `xc_work/xc_make` Makefile as:

```
vavlog_parallel: vavlog_1 vavlog_2 vavlog_3 vavlog_4 vavlog_5
```

It also creates a target rule for each library in `xc_work/xc_make` as follows:

```
vavlog_1:
vavlog $(VAVLOG_OPTIONS) -append_log $(VALIB) +ixcom -Work IXCOM_TEMP_LIBRARY \
       -File xc_work/v/vavlog-IXCOM_TEMP_LIBRARY-verilog.f

vavlog_2:
vavlog $(VAVLOG_OPTIONS) -append_log $(VALIB) +ixcom -Work lib1 \
       -File xc_work/v/vavlog-lib1-verilog.f

vavlog_3:
vavlog $(VAVLOG_OPTIONS) -append_log $(VALIB) +ixcom -Work lib2 \
       -File xc_work/v/vavlog-lib2-verilog.f

vavlog_4:
vavlog $(VAVLOG_OPTIONS) -append_log $(VALIB) +ixcom -Work lib3 \
       -File xc_work/v/vavlog-lib3-verilog.f

vavlog_5:
vavlog $(VAVLOG_OPTIONS) -append_log $(VALIB) +ixcom -Work lib4 \
       -File xc_work/v/vavlog-lib4-verilog.f
```


© 2021 All Rights Reserved.

Then, IXCOM invokes HDL-ICE Compiler as follows:

```
make -f xc_work/xc_make vlog_serial
make -f xc_work/xc_make vlog_parallel -j <nJobs>
make -f xc_work/xc_make vaelab_only
```

In the `ixcom.log` file, an INFO message is generated to indicate the number of job slots used for the parallel invocation.

Note: You can also achieve this parallel invocation by manually modifying the `xc_work/xc_make` Makefile.

Compiling Mixed-Language Designs

During compilation, IXCOM elaborates the design using the analyzed VHDL/Verilog design units. Then, the design is automatically partitioned into synthesizable and non-synthesizable parts. The synthesizable parts can run on the emulator; non-synthesizable parts can run only in the software simulator. See Compiling for Hardware Using IXCOM on page 634.

Following are the commands used for analyzing and compiling a mixed-language design.

```
vlan <file1.v> <file2.v>
vhan <file1.vhd> <file2.vhd>
ixcom -top <top_tb> -ua +dut+<top_dut>
```

Here is the basic syntax of the `ixcom` command:

```
ixcom -top top_level_VHDL_Verilog_unit_name \
      [-ua +dut+<entity_or_module> other_options]
```

The `ixcom` command performs the following tasks:

- Compiles the modules in the source files specified with the `.v` extension for a Verilog-only design. (IXCOM detects the file types according to the filename extensions. See Determining File Type by Source Filename Extension on page 659.)
- Elaborates the design.
- Generates the corresponding binary source code of the design.
- Compiles the design into a simulation snapshot that can be run on the emulator if the `-ua` option is given with the `ixcom` command.

The `ixcom` command supports all standard Verilog and/or VHDL options. In addition, it supports command-line options for integrating PLIs and the options for compiling. The `ixcom` command options are of following types:

- Dash options - beginning with a dash (-) symbol

© 2021 All Rights Reserved.

- Plus arguments - beginning with a plus (+) symbol (See Using Plus Arguments on page 805)

Handling Time Unit and Precision Settings

The Verilog language enables a way to specify the time unit/precision per module. In VHDL, the time unit/precision needs to be set according to the tool used.

- For VHDL: Pass the `-vhdl_time_precision` <*`time_precision`*> to IXCOM. For example, `-vhdl_time_precision 1ns`. IXCOM sets this time precision for all VHDL modules.
- For Verilog: If a module does not have the time unit/precision, you can set the time unit/precision by using the `-timescale` <*`unit/precision`*> option of the `ixcom` command.

The global minimum time unit/precision is set after considering both the VHDL and Verilog modules in the design.

Note: The same time unit and time precision settings should be specified for IXCOM and *xrun*. For example, the `uxc_hdl.var` file, which is used to pass options to *xrun*, should not include a separate setting if you have already passed these options to IXCOM.

Partitioning Designs into RTL (Hardware) and Behavioral (Software)

IXCOM compiler analyzes and elaborates the entire design, running RTL checks, to determine the synthesizable and non-synthesizable portions. It automatically partitions the non-synthesizable portions to be run in the software partition and the synthesizable portions to be run on the hardware emulator.

RTL Synthesis Checks

The RTL synthesis checks provide information on the synthesizability of the design. There are two types of RTL synthesis checks:

- Ignore:

  These checks report information about the design constructs or elements that are non-synthesizable and ignored by the compiler. Constructs falling in this category are reported with the `RTL_IGNORE_` prefix.

- Unsupported:

© 2021 All Rights Reserved.

These checks report information about design constructs or elements that are non-synthesizable and unsupported. Constructs falling in this category are reported with the `RTL_UNSUPPORTED_` prefix.

For RTL synthesis checks, use the `-rtlchk` option of the `ixcom` command.

The `-rtlchk` option has three values: `warn`, `off`, and `error`. Of these, `-rtlchk error` is the default setting. So, by default, all modules in the DUT module hierarchy (specified using the `+dut+` <*`module_name`*> option) are considered synthesizable, and partitioned to emulator. If a specified module does not pass the RTL checks, an error is generated and compilation stops. You can change this default behavior by specifying the value as `off` (completely switching off the RTL checks) or `warn` (a warning is generated and the specified module and its entire hierarchy are partitioned to the software side).

Based on the value specified for the `-rtlchk` option (`warn, error`, or `off`), IXCOM takes the following actions:

- `-rtlchk error`: Displays a message and aborts compilation.

  The non-synthesizable design constructs are reported with the `RTL_UNSUPPORTED_` prefix in the message and compilation is aborted.

  Note: If you intend to run `tbrun` or `autorun`, use the `-rtlchk error` option so that design violations are addressed at the compile time, pushing design modules to the software partition by using the `-rtlchk warn` option could lead to incorrect results in the simulation.

- `-rtlchk warn`: Displays a message and the non-synthesizable module is not pushed to the software partition.

  The non-synthesizable design constructs that are not supported are reported with the `RTL_UNSUPPORTED_` prefix in the message.

  If the compiler generates any messages with the `RTL_IGNORE_` prefix, such constructs are ignored in the module. Note that for such ignored constructs, the module is not partitioned to the software-simulator side for such ignored constructs found.

  > **Note:** The compilation aborts if the top-level DUT module (specified using the `+dut+` option) is pushed to the software simulator side.

- `-rtlchk off`: Disables the automatic partitioning of the design.

  When the `-rtlchk off` option is specified, no design partitioning takes place. IXCOM passes all the modules under the `+dut+`<*`mod_name`*> hierarchy to HDL-ICE Compiler without conducting any RTL synthesis checks. Then, if HDL-ICE Compiler detects any non-synthesizable modules, it generates an error message and aborts compilation.

© 2021 All Rights Reserved.

Example - RTL Synthesis Checks

Following is an example of how IXCOM checks for RTL synthesis and reports errors and warnings as appropriate.

Consider the following code.

```
module top();
  reg clk = 0;
  wire [7:0] hwcount;
  wire [7:0] swcount;
  always #5 clk =~ clk;
  dut dut0(clk, hwcount, swcount);
  initial $monitor($time,,,hwcount,,,swcount);
  initial #100 $finish;
endmodule

module dut(clk, hwcount, swcount);
  input clk;
  output [7:0] hwcount;
  output [7:0] swcount;
  sw sw0(clk, swcount);
  hw hw0(clk, hwcount);
endmodule

module hw(clk, count);
  input clk;
  output [7:0] count;
  reg [7:0] count = 0;
    always @(posedge clk)
    begin
      count <= count + 1;
    end
endmodule

module sw(clk, count);
  input clk;
  output [7:0] count;
  reg [7:0] count = 0;
    always @(posedge clk)
    begin
      @(negedge clk);
      count <= count + 1;
    end
endmodule
```

In the Verilog example above, the `sw` module is instantiated under `dut` module, and it is not synthesizable. By default, (`-rtlchk error`), if a module fails the RTL subset checks, the IXCOM compiler generates an error and aborts. Thus, the following `ixcom` command will fail.

```
$ ixcom -clean -ua +dut+dut test.sv
    ...
    ...
    ...
    Top level design units:
       top
    Design hierarchy summary:
                                    Instances Unique Splits
        Modules:                          4      4      4
    Error! in file test.sv at line 35 in module sw    [RTL_UNSUPPORTED_EVENT]
```


© 2021 All Rights Reserved.

```
        unsupported event control statement type in rtl mapper.
     ...
        Info                        [RTL_VIOLATION]
            There are total of 1 module violated RTL subset checking.
      (sw)
      Please remodel this module to meet RTL requirements for Palladium Z1
logic
      mapping.
...
```

The `sw` module failed the RTL subset check because the unsupported event control statement used. It is recommended that in such cases, you remodel the module so that it passes the subset checks. Else, there can be the following negative side effects:

- xeCompile scheduling failure due to I/O congestion
- Functional failures due to race conditions
- Slow performance due to heavy I/O load

Boundary modules summary: (only the modules that violated RTL subset checking are included)

```
Number of Instance * Est. I/O width = Total I/O width  (Module)
             1 *          9 bit =           9 bit   (sw)
Errors : 1 Warnings : 0 Info : 2
```

Manual Partitioning of the Design

If required, direct the partitioning of the designs manually:

- Mark parts of the synthesizable module to be run on the software side by modifying the design code and adding the required testbench callback statements (See Executing Non-Synthesizable Code in the Emulator on page 766).
- Use the `+no_rtlchk+<module_name>` option of the `ixcom` command to skip automatic partitioning for a particular module or set of modules of the design. Use this option for a module or set of modules that might work correctly on the emulator but the automatic design partitioning pushes the module to the software partition.

Compilation Log Files

A number of log files are generated during compilation. All of these files are created in the current directory by default, but you can use the `-l <logfile>` option with any of these commands to specify a different filename and/or directory path.

- `vhan.log` (VHDL only)

© 2021 All Rights Reserved.

Created by the VHDL analyzer program, `vhan`, this file contains messages relating to the syntax checking of your VHDL design.

- `vlan.log` (Verilog only)

  Created by the Verilog analyzer program, `vlan`, this file contains messages relating to the syntax checking of your Verilog design.

- `ixcom.log`

  Created by the `ixcom` command, this file contains a log of the complete compilation process along with the list of options used for the compilation process. The `ixcom.log` file also contains the following information if you are using the `-ua` option with the `ixcom` command for compiling:

  - Messages produced by the Synthesis Policy Checker
  - Signal Analysis report

Summarizing Messages for Readability and Ease of Analysis

Numerous messages of `INFO`, `WARNING` and `ERROR` types are generated by the different analyzers and compiler during IXCOM compilation. To make it easy to identify the number of times and the stage at which a particular message was generated, you can choose to generate an easy-to-read message summary by each analyzer or compiler in a separate log file.

To generate the message summary, specify the `-msgSummary` option with the required command. The message summary is generated into a separate log file, with the command name prefixed to the log-file name. For example:

```
vhan -msgSummary generates vhan.msg.log
vlan -msgSummary generates vlan.msg.log
ixcom -msgSummary generates ixcom.msg.log
```

In the single-step compilation mode, a single log file is generated with the name `ixcom.msg.log`, for all the commands executed during the IXCOM compilation, that is `vlan`, `vhan`, and `ixcom`.

In the multi-step compilation mode, the name of the log file depends on the name of the executed command. For example, for `vlan`, it is named as `vlan.msg.log`, for `vhan`, it is named as `vhan.msg.log`, and so on.

The message summary includes individual counts for every `INFO`, `WARNING`, and `ERROR` message type at the end of command execution. So, for each type of message, the log file includes the message codes and the number of times that the message code was generated.

© 2021 All Rights Reserved.

Following is a sample `ixcom.msg.log` file:

```
// NOTE: This file corresponds to message count summary from vlan, vhelab, and
ixvcg. Single step mode is not applicable for VHAN.
###########
vlan
###########
Info
********
No Message generated in this category.
Warning
***********
No Message generated in this category.
Error
***********
No Message generated in this category.
###########
vhelab
###########
Info
********
No Message generated in this category.
Warning
***********
No Message generated in this category.
Error
***********
ILL_PARAM : 1
```

Following is a sample `vhan.msg.log` file:

```
// NOTE: This file corresponds to message count summary from vhelab binary
Info
********
No Message generated in this category.
Warning
***********
No Message generated in this category.
Error
***********
ILL_PARAM : 1
```

Support for SystemVerilog XMR Referring to VHDL Objects

For mixed SystemVerilog-VHDL designs, IXCOM provides support for SystemVerilog XMRs (Cross-Module References) referring to VHDL objects of the following types:

- std_logic type objects
- std_logic_vector type objects
- Integer type
- 2-D Array of std_logic or its bit-select, or its part-select
- Field of a record type object where field type is std_logic, std_logic_vector, or integer

© 2021 All Rights Reserved.

- Field of an array of records type object where field type is std_logic, std_logic_vector, or integer

Limitations of this Support

- XMRs to VHDL records having a field of type real are not allowed.
- XMRs defined in DUT cannot be assigned in testbench.

Examples for Support for SystemVerilog XMR Referring to VHDL Objects

Following are some examples illustrating this support:

```
// VHDL type declarations

package def is
    type recT is record
        I: std_logic_vector(9 downto 0);
        J: std_logic;
    end record;

    type recRecT is record
        data : recT;
        vld : std_logic;
        end record;
    type arrayRecRecT is array (NATURAL range <>) of recRecT;
    type array2DT is array (integer range 1 to 7) of std_logic_vector(0 to 8);
    subtype array2DT1 is array2DT;
end package def;

//VHDL objects of record and Array of record types

entity tx_top is
    port( outA : in arrayRecRecD(15 downto 0);
            p1 : in recRecD;
            I1 : in integer; );
end tx_top;

architecture rtl of tx_top is
    signal sig2D : array2DT1;
begin
end;

// SV XMRs to VHDL records and Array of records

wire [1:7][0:8]dut2;
wire [0:8]dut1;
assign dut1[2] = {vtop.subcore.topInst.outA[15].data.J } ; --->> XMR to array of
                                                              // records
assign dut1[5] = {vtop.subcore.topInst.p1.data.I[2] } ;    --->> XMR to records
assign dut1[6] = {vtop.subcore.topInst.I1 } ;              --->> XMR to Integers

assign dut1 = {top_tb.i_test_module.sig2D[1] } ; --->> XMR to 2-D array
                                                   // object(bit-sel)
if (top_tb.i_test_module.sig2D[2][0:7] != 'h0)   --->> XMR to 2-D array
                                                   // object(part-sel)
assign dut2 = {top_tb.i_test_module.sig2D } ;    --->> XMR to 2-D array object
```


© 2021 All Rights Reserved.

Compiling for Hardware Using IXCOM

By default, designs are compiled to run in software simulation only. To be able to run in Simulation Acceleration modes, add the `-ua` and `+dut+<entity_module>` options to the `ixcom` command line.

For mixed-HDL or VHDL Designs:

```
ixcom -top <top_level_Verilog_configuration_name> -ua \
      +dut+entity|module other_options
```

For Verilog-only Designs:

```
ixcom -ua +dut+module other_options <design_files>
```

The `+dut+` option indicates the parts of the design that you want to be compiled for hardware. These parts will then execute in the emulator when running in Simulation Acceleration mode.

Directing Design Compilation for Hardware

When compiling the design to run on hardware, you can specify various parameters, such as emulator configuration and vision mode by using the User Data and compilation commands. These commands are listed in the *User Data Commands* and *Compiler Commands* chapters of the *VXE Command Reference Manual*. To specify these commands, include these commands in a file called `compilerOptions.qel`, and place this file in the same directory in which the design is being compiled and run. The IXCOM compiler automatically sources these commands from the `compilerOptions.qel` file while compiling the design.

By default, the IXCOM compiler compiles using the XEL `compile` command, which performs only one compilation trial. To perform multiple trials, direct IXCOM to use the `compileFind` command by setting the `compileScript` TCL variable in the `compilerOptions.qel` file, as follows:

```
set compileScript "compileFind [options]"
```

For example:

```
set compileScript "compileFind -first -normal"
```

Note: In the `compilerOptions.qel` file, if you need to specify a User Data command (for example, the `moduleFV` command) that requires naming the library, use `hdldb` for IXCOM flow, because IXCOM uses a fixed library, `hdldb`.

© 2021 All Rights Reserved.

Passing Options to HDL Compilation Utilities from the IXCOM Command Line

To pass options to the HDL compilation utilities, such as vavlog, vaelab, and vavhdl, use the `-vaelabargs`, `-vavlogargs`, and `-vavhdlargs` options, respectively, with the `ixcom` command.

For details about using these utilities, refer to the Using UNIX Commands for RTL Import and Synthesis section.

In VXE 18.1 and 18.5 versions, the `-db2` option to enable parallel synthesis was the default option. From VXE 20.05, to enable parallel synthesis while compiling the designs for SA mode, specify the following options with the `ixcom` command:

- `-vavlogargs -db2`
- `-vavhdlargs -db2`
- `vaelabargs -parallel <number>`

  Note: Use the `-legacy_bdd` option with the `vaelabargs -parallel <number>` option to enable parallel synthesis with the old synthesis engine.

For example:

```
ixcom -ua +dut+<top_level_module> <>.v -vaelabargs [vaelab_options]
```

Detection of Hardware Platform Based on xeCompile Environment

Starting from IXCOM 20.12.s002, IXCOM detects the hardware platform, such as VXE, WXE or PTM based on the xeCompile environment being used by the user when the `-ua` option is specified. IXCOM applies the corresponding hardware platform option, such as `-z1`, `-z2`, or `-ptm` during compilation automatically.

For example, suppose the following command is issued:

```
$ ixcom -clean -ua +dut+dut test.sv ...
```

Then, the following notification message is issued:

```
Info! VXE was detected from user environment. IXCOM will use this platform as the
default for hardware compilation. Option "-z1" is added to the vhelab/ixvcg.
```

Alternatively, if the user has WXE hardware platform, the following message is issued:

```
INFO: WXE xeCompile was detected from user environment. IXCOM will use this platform
as the default for hardware compilation.
```


© 2021 All Rights Reserved.

You can check the execution platform by using the `-version` option of the `xeCompile` command. The following example shows the output from different xeCompile platforms.

```
$ /lan/cva_rel/vxe21h1/red/bin/xeCompile -version
21H1_VXE, V21.02.039.s001
$ /lan/cva_rel/wxe21h1/red/bin/xeCompile -version
21H1_WXE, V21.00.039.b003
$ /lan/cva_rel/ptm21h1/red/bin/xeCompile -version
```

Prior to this enhancement, if the wrong execution platform was used, the following error message was generated:

```
Error: Version check failed. IXCOM is compatible with ( 21.00.s000 ) release(s) of
WXE xeCompile. You are currently using version 21H1_VXE, V21.02.039.s001
This version of xeCompile is compatible with the following IXCOM release(s) (
20.05.s000+, 20.12.s000+ ).
Since the release date of xeCompile is newer, it takes the precedence on the version
checking. Please also check the version string and make sure the right platform is
used.
Use -ignoreVersionCheck to downgrade the severity to a warning if needed.
```

Before this enhancement, if no specific hardware platform option, such as `-z1`, `-z2`, or `-ptm` was used, the hardware platform was assumed as `VXE`.

Messages and Information about Detected Platforms

The `INFO` messages are generated during compilation in the `ixcom.log` file and at run time for the hardware platform being used.

During compilation, the detected hardware platform is specified in the `ixcom.log` file if the following command is used:

```
$ ixcom -clean -ua +dut+dut test.sv ...
Info!      VXE was detected from user environment. IXCOM will use this platform as
the default for hardware compilation. Option "-z1" is added to the vhelab/ixvcg.
```

At run time, if the target and execution platforms do not match, a warning message is issued in the log files, such as `xrun.log`, `xcprof.out`, and `tmp/xc_run.log`. Otherwise, an `INFO` message is issued in the `tmp/xc_run.log` and `xcprof.out` files.

For instance, suppose the following commands are issued:

```
$ xrun -R
...
$ cat tmp/xc_run.log
...
```

Then, the following message is generated:

```
INFO: target platform:Z1, execution platform:Z1
...
```


© 2021 All Rights Reserved.

Switching the Platforms

This enhancement facilitates changing the hardware platform for stage compilation from `VXE` to `WXE` and `WXE` to `VXE` if the initial IXCOM compilation was via platform detection, that is, the `-z1` or `-z2` option were **not** explicitly specified.

Note that these are the only changes supported for hardware platforms. All other combinations of platform changes by user environment result in an error during version checks.

For example, suppose the following commands were used:

```
$ ixcom -clean -ua +dut+dut test.sv          # user has VXE environment.
...
$ ixcom -target xeCompile -z2                # user has WXE environment now.
...
```

Then, the following messages will be displayed:

```
Info!      WXE was detected from user environment.
Info!      xc_work/rt.info is updated.
Warning!   xeCompile platform is updated using WXE and the IXCOM front end
compilation was based on VXE. Please note that runtime performance could be
suboptimal.
...
```

Switching the Platforms During Staged Compilation

With the `-target` option of the `ixcom` command, you can recompile the design at xeCompile stage. The `-target` option also enables you to switch the execution platform by simply setting up the environment and recompiling the xeCompile stage.

IXCOM supports the following combinations of <*targetPlatform*>:<*executionPlatform*> options. The `-z1:z2` option means that the design should be compiled with target platform on VXE but the design should run on WXE. If user environment does not match with the platform specified in the option, IXCOM generates an error message.

```
-ptm:ptm
-ptm:z1
-ptm:z2
-z1:z1
-z1:z2
-z2:z2
```


© 2021 All Rights Reserved.

Following are some cases for switching execution platforms using the `-target` option. Case 1 is supported. Cases 2 and 3 are not supported and an error message is issued for these cases.

- Case 1:

  The design is compiled with two different xeCompile platforms. For the first `ixcom` command, VXE is assumed as the execution platform. However, the final execution platform is changed by the second `ixcom` command. In this case, the execution platform is updated and the design compilation proceeds successfully. Note that -z1:z2 option is required to change the execution platform by recompiling the design from xeCompile stage.

  A warning message is issued to indicate that the result is suboptimal for changing the platform.

  ```
  $ ixcom ... [rest of the options] (user has VXE setup)
  ```

  Later, WXE is used for recompilation.

  ```
  $ ixcom -target xeCompile                      /*(user has WXE setup) */
  $ ixcom -target xeCompile -z1:z2               /*(user has WXE setup) */
  ```

- Case 2:

  If the platform is explicitly specified using an option, such as -z1, -z2, or -ptm, any changes of platform by subsequent `ixcom` commands via the user environment are not allowed, and an error message is generated.

  ```
  $ ixcom -z1 -target hdlice ... [rest of the options]
  ...
  $ ixcom -target xeCompile      /*(user has WXE environment) */
  ```

- Case 3:

  Using one platform to import the design and using another platform after importing the design is not supported.

  ```
  $ ixcom … -target hdlice                      /*(user has VXE setup) */
  $ ixcom -target xeCompile:designImport        /*(user has VXE setup) */
  $ ixcom -target xeCompile:skipImport          /*(user has WXE setup) */
  ```

Modifying the IXCOM Compilation Flow

By default, IXCOM invokes `xeCompile` to run the `ixcom.qel` script located at *<VXE_INSTALL_HOME>*`/share/vxe/etc/qel/ixcom.qel`. In some rare cases, you

© 2021 All Rights Reserved.

might need to change the commands executed by `xeCompile`, and editing commands in the `compilerOptions.qel` file might not be sufficient for this change. Then, you can copy the `ixcom.qel` file to the compilation directory, rename it to `compile.qel`, and make the necessary changes as described below. When `xeCompile` executes the `ixcom.qel` script and locates a `compile.qel` file in the compilation directory, it starts executing the `compile.qel` file instead of running the rest of the `ixcom.qel` script.

Caution

Use a compile.qel file only if there is no alternative. If you create compile.qel based on ixcom.qel, and in a future VXE release, the ixcom.qel is modified, your compile.qel will be based on an obsolete ixcom.qel, and might not work properly.

To modify the IXCOM compilation flow, use the following method:

1. Create the `compile.qel` file in the IXCOM compilation directory. You can use the `ixcom.qel` or `xeCompile.key` files as a template to create the `compile.qel` file.

   For example:

   ```
   cd <my_compile-dir>
   cp <VXE_INSTALL_HOME>/share/vxe/etc/qel/ixcom.qel compile.qel
   ```

   If the `ixcom.qel` file is used as a template, the following section should be removed from the `compile.qel` file:

   ```
   if {[file exists compile.qel]} {
       source compile.qel
       exit
   }
   ```

   Note: The `compilerOptions.qel` file overrides the User Data in the `compile.qel` file if both files are present in the IXCOM compilation directory.

2. Modify the `compile.qel` file as per your requirements.

3. Compile the design files using the `ixcom` command.

Automatic Selection of Protocol for SA Transfer

While compiling the design, you can select the protocol you want to use by using the following command:

```
compilerOption -add {SAProtocol <CLASSIC|MODULAR>}
```

By default, the protocol is set to `CLASSIC`. Specify this command in the `compilerOptions.qel` file that is passed with the `ixcom` command.

© 2021 All Rights Reserved.

The protocol for SA transfer depends on the number of PI and PO, and the number of domains that will be occupied by your design. If there are large number of PI and PO, the design might not fit into one domain. One domain provides physical locations for 36K PI and 96K PO. If the design cannot fit into one domain, the etcompiler automatically sets the protocol to `MODULAR`. The `CLASSIC` protocol assumes that only one domain is used. Note that because the `MODULAR` protocol uses more than one domain, it is a little slower than `CLASSIC`.

The etcompiler might use `MODULAR` protocol even if the number of I/O is less than the limit supported by the hardware for `CLASSIC`. But it will run slower than with `CLASSIC` protocol. On the other hand, if the I/O limit is close to the hardware limit, selecting `CLASSIC` might still work.

Hardware utilization limits and PI/PO counts determine if MODULAR protocol would be used:

- The highest utilization limit might set the protocol to `MODULAR` for multi-domain compiles when >36K PI signals or >96K PO signals.
- The middle utilization might set the protocol to `MODULAR` for >16K PI and >64K PO.
- The lowest utilization might set the protocol to `MODULAR` for > 16K PI or > 16K PO.

Compiling SystemC Designs

IXCOM supports SystemC with any combination of Verilog and VHDL files. You can also simulate SystemC-only designs using this process.

SystemC supports a limited set of data types at the boundary between a Verilog or VHDL component and a SystemC model. Supported type combinations are listed in Table 16-1 on page 641.

Supported compilers are GCC G++ 4.1.2 on Linux.

Supported Types

Table 16-1 on page 641 lists SystemC, Verilog and VHDL types supported by SystemC at the boundary. Any SystemC type from this list can be connected to any Verilog or VHDL type from the list. The only requirement is that the bit size of the connected ports must be the same.

© 2021 All Rights Reserved.

Note: User-defined VHDL types are not supported on the SystemC boundary.

Table 16-1 SystemC, Verilog, and VHDL types supported by SystemC level 2

SystemC Types	Verilog Types	VHDL Types
bool	net (scalar and vector)	bit
sc_bit	reg (scalar and vector)	std_[u]logic
sc_logic	integer	boolean
[unsigned] char		bit_vector
[unsigned] short		std_[u]logic_vector
[unsigned] int		array of boolean
[unsigned] long		
sc_[u]int<N>		
sc_big[u]int		
sc_bv<N>		
sc_lv<N>		

Developing and Simulating SystemC Designs

This section describes a procedure to develop and simulate different kinds of mixed-HDL/ SystemC designs.

Cadence recommends that you write a makefile to automate this flow. You will find examples of makefiles in `<IXCOM_INSTALL_HOME>/share/vxe/gift/tutorial/systemc/`.

Note: For using SystemC, only the standard IXCOM environment is required; you do not need to do any special environment settings.

Designing and Coding Recommendations

When you are designing and coding for SystemC:

© 2021 All Rights Reserved.

- Define cross-boundary ports using any of the types listed in Table 16-1.
- Add the `NCSC_MODULE_EXPORT` macro for modules to be instantiated in Verilog or VHDL.
- Define an instance of a Verilog or VHDL component in SystemC as a `sc_foreign_module`.
- Do not use global variables of type `SC_MODULE`.
- You do not need any special markers for SystemC modules instantiated in Verilog or VHDL.
- If you need to pass Verilog parameters or VHDL generics to the instantiated SystemC module, use the method described in the SystemC Simulation User Guide in the Xcelium documentation set.
- Create a SystemC wrapper for each Verilog or VHDL component instantiated in SystemC. Details about the wrapper coding can be found in Instantiating Verilog/VHDL under SystemC on page 644 .

Compiling and Linking SystemC Components

To compile and link your HDL and SystemC components:

- Choose the design libraries with which your SystemC modules will be associated.
- Use `ncsc_run` to analyze SystemC files that contain SystemC modules to be instantiated in Verilog or VHDL.
- Separately analyze each mixed-HDL component. For VHDL files, use `vhan` and for Verilog-only files, use `vlan`.

Compiling SystemC Designs

To compile a SystemC design, follow the same process as for standard mixed-HDL flow elaboration, with the addition of the option, `-loadsc`. Here is an example command line:

```
ixcom -top <top_module> -loadsc ./libncsc.so
```

You must specify all SystemC top modules (not instantiated in HDL) with the `-top` option. The constructors of SystemC modules are started twice: once during elaboration, and again at run time.

If you specify `sc_main` as a *`top_module`*, the `sc_main()` global function is started; otherwise, `sc_main()` is not used.

© 2021 All Rights Reserved.

Using the sc_main() Global Function

If you pass the `-top sc_main` option to the elaborator, as described in the preceding section, the `sc_main()` global function is started. Here is the syntax of the function:

```
sc_main(int argc, char *argv[]) {
  ...
}
```

The `sc_main` function is called before any other SystemC modules instantiated in HDL or specified as `-top` modules.

To pass the arguments to the `sc_main` function, it is necessary to use the `+systemc_args+"<args>"` option of the `ixcom` command. During elaboration, *argv* contains all these options specified for `ixcom`. During run time, *argv* contains all plus options specified during elaboration and all options passed to *xrun*.

All elaborator arguments specified with a `+` option are accessible in `sc_main()` by means of *argc* and *argv*, both during elaboration and at run time. It is not necessary to repeat these options at run time.

SystemC and Mixed Modules as Top Modules

A SystemC module can be the topmost module of the design.To enable it, declare this module using the `NCSC_MODULE_EXPORT` macro. The second argument of the macro is the name of the top module that should be specified with the `-top` option on the `ixcom` command line.

You can have multiple top modules, some in Verilog/VHDL and others in SystemC.

Simulating SystemC Designs (*xrun*)

SystemC designs are simulated just like standard mixed-HDL flow designs with *xrun*.

To pass the arguments to the `sc_main` function, it is necessary to use the `+systemc_args+"<args>"` option of the `ixcom` command.

Instantiating SystemC under Verilog/VHDL

Verilog and VHDL instantiate a SystemC model using the same syntax used to instantiate a corresponding HDL instance.

On the SystemC side, declare the instantiated module types using the following special macro:

© 2021 All Rights Reserved.

```
NCSC_MODULE_EXPORT(<systemc_module_name>,
                   <verilog_module_name>|<vhdl_module_name>);
```

The first argument to the macro is the class name of the `sc_module` that you want to instantiate. The second argument specifies the HDL module name.

There are two reasons for showing two different arguments (<*verilog_module_name*>|<*vhdl_module_name*>) in the macro:

- VHDL is not case sensitive, while Verilog and SystemC are case sensitive. Therefore, the case of VHDL component and port names (for named connections) does not need to match the case of the SystemC counterparts. The case of Verilog and SystemC identifiers, however, must be identical.
- Not all SystemC module (type) names can be used in Verilog. For example, templatized module names (such as `my_and<4>`) cannot be used in Verilog.

Some SystemC module names cannot be used in the `XSC_EXPORT_MODULE` macro. For example, if a SystemC module name is templatized, it can use the comma separator. The comma character (**,**) is considered by the macro to be a separator. To work around this restriction, you might need to add a special `typedef` declaration to alias the SystemC module type name to one that can be used in the macro.

Every SystemC module you instantiate under Verilog or VHDL must have a constructor that takes only one argument of type `sc_module_name`. Alternatively, you can use the standard macro `SC_CTOR()`, which declares such a constructor.

Note: You must specify `NCSC_MODULE_EXPORT` in the `.cpp` file.

A warning is issued if the symbolic name is not specified for the cross-boundary SystemC port, or if the symbolic name is specified, but does not match the corresponding port name on the HDL-side. The symbolic name must match even if you are using positional binding of ports. The symbolic name is required for visibility of the ports in waveform viewers. If a mismatch exists, a warning is issued by the elaborator.

Instantiating Verilog/VHDL under SystemC

To instantiate a Verilog module or VHDL component in the SystemC module, you need to create a shell. Use the xmshell utility to create this shell or follow the method given below.

Declare a new wrapper class derived from the class `sc_foreign_module`. This class is derived, in turn, from the standard SystemC `sc_module` class and is a Cadence extension to the SystemC library `libxsc`. The name of the wrapper class must match the Verilog module name or VHDL entity name. The wrapper class definition can be written manually or by using the `xmshell` utility.

© 2021 All Rights Reserved.

The rules for writing wrappers are:

- Define the corresponding C++ class derived from the predefined class `sc_foreign_module`. The macro `SC_FOREIGN_MODULE` can be used to achieve the same functionality.

  For example:

  ```
  SC_FOREIGN_MODULE(Mult)
  ```

- Declare the ports of the wrapper as follows:

  The order of the ports must match exactly the order of the formal ports in the HDL module (or entity) declaration.

  The wrapper port names must be exactly the same as the names of the formal ports.

  Port length in bits must be the same as in HDL.

- Define a constructor of the wrapper module as follows:

  `SC_CTOR`(Mult): *port-names-list* {}

  where *port-names-list* looks like this:

  port-field-name(“*actual-port-symbolic-name*”), …

  For example:

  ```
  SC_CTOR(Mult) : a("a"), b("b"), clk("clock") {}
  ```

Support for Multiple SystemC Shared Libraries

The IXCOM compiler supports multiple SystemC shared libraries created by the `ncsc` command. This support ensures that the SystemC libraries obtained from multiple sources are also supported. Another benefit of supporting multiple SystemC libraries is that it enables you to partition the SystemC code in multiple libraries. These SystemC shared libraries are created using the `ncsc` command and are passed to IXCOM by using the `-loadsc` option.

For example, to load multiple SystemC shared libraries, `libncsc_model1.so`, `libncsc_model2.so`, and `libncsc_model3.so`, the `ixcom` command is:

```
ixcom -loadsc libncsc_model1.so -loadsc libncsc_model2.so [other_ixcom_options]
```


© 2021 All Rights Reserved.

Using Specman Elite with IXCOM

You can use IXCOM to compile both Verilog and mixed-language DUT in the Specman environment, with a Specman ***e*** testbench.

This section discusses the following:

- Compiling and Running a Verilog Design in the Specman Environment
- Compiling and Running a Mixed-Language Design in the Specman Environment

Compiling and Running a Verilog Design in the Specman Environment

In the Specman environment, use IXCOM to compile a Verilog or mixed-language DUT and *xrun* to compile the Specman ***e*** testbench and simulate the DUT. To compile a Verilog design in the Specman environment, you need a Specman-to-DUT reference file and a Specman stub file. The Specman-to-DUT reference file, typically named as `specman_hdl_ref.v`, is required by IXCOM whereas the Specman stub file, typically named as `specman.v`, is required by *xrun*.

For example, suppose you have a Verilog DUT, *`xor.v`*, a Specman ***e*** testbench, *`xor_verify.e`*, and a sample script file, *`xc.t`* that contains the simulation acceleration commands.

A sample script file, *`xc.t`*, contains the following commands:

```
run 200ns
xc on -run
run
exit
```

Perform the following steps to compile and run the DUT with the Specman ***e*** testbench:

1. Create a Specman-to-HDL reference file manually based on how each DUT signal is used in Specman testbench. Refer to Accessing DUT Signals from Non-HDL Interfaces for details about creating such a file. For example:

   ```
   $export_read(xor_try.clk);
   $export_deposit(xor_try.inp_xor);
   $export_read(xor_try.out_xor);
   ```

   Note: You can get the list of HDL-referenced signals from Specman testbench by using the `dump hdl_ref` command in the Specman tool. Specify the `specman` command to generate the Specman-to-DUT reference file `specman_hdl_ref.v`, as shown:

   ```
   specman -command "load xor_verify.e; dump hdl_ref -verilog_top"
   ```

   A sample `specman_hdl_ref.v` file is given below:

© 2021 All Rights Reserved.

```
module specman_hdl_ref;
      initial begin
          $export_spexsim(xor_try.clk);
          $export_spexsim(xor_try.inp_xor);
          $export_spexsim(xor_try.out_xor);
      end
endmodule
```

Be sure to modify the `specman_hdl_ref.v` based on the way each signal is used in the Specman testbench before continuing the next step. Modify the `specman_hdl_ref.v` file if the port bindings for the DUT signals are ***e*** string variables instead of the tick (`) notations, because some of the DUT access signals might not be extracted from the testbench.

2. Generate the Specman stub file, `specman.v` for *xrun*, as shown:

```
specman -command "load xor_verify.e; write stubs -xmvlog"
```

3. Specify the `ixcom` command to compile the Verilog DUT and the Specman-to-DUT reference file, `specman_hdl_ref.v`, as shown:

```
ixcom -ua +dut+xor_try +iscDelay+clk_gen xor.v specman_hdl_ref.v
```

4. Specify the `xrun` command to compile the Specman testbench and the Specman stub file. This command also elaborates the DUT objects with Specman, as shown:

```
xrun -elaborate xor_verify.e specman.v -f xc_work/xrun.f
```

5. Specify the `xrun` command to run the design, as shown:

```
xrun +dpvcl -R -input xc.t
```

In the above command, the `+dpvcl` option is required to enable IXCOM to capture changes in signal values by Value Change Callback (VCL). This is because Specman modifies the DUT signal values directly in the *xrun* space.

Compiling and Running a Mixed-Language Design in the Specman Environment

In the Specman environment, use IXCOM to compile a mixed-language DUT, which can contain SystemVerilog, Verilog, and VHDL portions. Also, use *xrun* to compile the Specman ***e*** testbench and simulate the DUT. For a mixed-language design, you need a Specman-to-DUT reference file and a Specman stub file. The Specman-to-DUT reference file, typically named as `specman_hdl_ref.v`, is required by IXCOM whereas the Specman stub file, typically named as `specman.v`, is required by *xrun*.

For example, suppose you have a VHDL DUT, *<xor>.vhd*, a Verilog file, *clk_gen.v*, and a Specman ***e*** testbench, *xor_verify.e*. You also have a sample script file, *xc.t* that contains the following simulation acceleration commands:

© 2021 All Rights Reserved.

```
run 200ns
xc on -run
run
exit
```

Perform the following steps to compile and run the VHDL DUT with the Specman ***e*** testbench:

1. Create a Specman-to-HDL reference file manually based on how each DUT signal is used in Specman testbench. Refer to Accessing DUT Signals from Non-HDL Interfaces for details about creating such a file. For example:

   ```
   $export_read(xor_try.clk);
   $export_deposit(xor_try.inp_xor);
   $export_read(xor_try.out_xor);
   ```

   Note: You can get the list of HDL-referenced signals from Specman testbench by using the `dump hdl_ref` command in the Specman tool. Specify the `specman` command to generate the Specman-to-DUT reference file `specman_hdl_ref.v`, as shown:

   ```
   specman -command "load xor_verify.e; dump hdl_ref -verilog_top"
   ```

   Be sure to modify the `specman_hdl_ref.v` based on the way each signal is used in the Specman testbench before continuing the next step. Modify the `specman_hdl_ref.v` file if:

   - The target signal is only forced or released but not read or written in the Specman testbench, then change `$export_spexsim` to `$export_frcrel`.
   - If the port bindings are for the DUT signals are ***e*** string variables instead of the tick (`) notations, then some of the DUT access signals might not be extracted from the testbench.

2. Generate the Specman stub file, `specman.v` for *xrun*, as shown:

   ```
   specman -command "load xor_verify.e; write stubs -xmvlog"
   ```

3. Specify the `vlan` and `vhan` commands to analyze the Verilog and VHDL source files, as shown:

   ```
   vlan specman_hdl_ref.v
   vlan clk_gen.v
   vhan xor.vhd
   ```

4. Specify the `ixcom` command to compile the VHDL DUT and the Specman-to-DUT reference file, `specman_hdl_ref.v`, as shown:

   ```
   ixcom -ua +dut+xor_try +iscDelay+clk_gen -top xor_try -top specman_hdl_ref
   ```

5. Specify the `xrun` command to compile the Specman testbench and the Specman stub file. This command also elaborates the DUT objects with Specman, as shown:

   ```
   xrun -elaborate xor_verify.e specman.v -f xc_work/xrun.f
   ```

6. Specify the `xrun` command to run the design, as shown:

   ```
   xrun +dpvcl -R -input xc.t
   ```


© 2021 All Rights Reserved.

Compiling Testbenches Separately

When compiling designs with IXCOM, you have the option to partition the design in such a manner that you can compile the testbench portion and elaborate the entire design separately using Xcelium tools, while the DUT is compiled using IXCOM.

Depending on the design hierarchy, any of the following are possible:

- The testbench and DUT are separate top-level modules (Specman, UVM, or TBA)
- The testbench instantiates the DUT
- The testbench is instantiated within the DUT

Accordingly, you can use the following options of the `ixcom` command:

- If the DUT and testbench are separate top-level modules or hierarchies like UVM or TBA, you can compile the DUT with the IXCOM compiler, followed by a separate step to compile the testbench and elaborate the complete design (testbench along with IXCOM-instrumented DUT) with Xcelium tools. Refer to Compiling Designs with SystemVerilog Constructs in SA Mode. For a scenario in which two independent top-level modules exist for the DUT and the testbench, and the DUT top-level module has I/O (inout) ports at the boundary of the software and hardware, refer to the Support for Bi-Directional Ports (inout) at the DUT-Testbench Boundary for Two-Top-Level Modules section.

  When you compile the design using IXCOM, the software and hardware portions of the design are compiled in parallel. For large designs with very large DUT portions, the hardware compilation can take longer than the time required for testbench compilation. In such cases, to proceed to the step of testbench compilation, without waiting for the DUT compilation to complete, you can choose to compile the testbench in parallel, and, thereby, minimize the overall compilation time. For this, you need to use the `-tbCompile` option as described in the Compiling Testbenches in Parallel section for details.

- If the DUT and testbench are in the same hierarchy and if the DUT is instantiated within the testbench, use the `+xcDesignTop` option to compile the testbench separately using Xcelium tools and compile the DUT using IXCOM. Refer to Compiling Top-Level Testbenches Using +xcDesignTop for more information.

Linking DUT with Multiple Testbenches

IXCOM enables you to compile the testbenches separately from the DUT, thereby, enabling you to link a DUT with multiple testbenches, without needing to recompile the DUT with each testbench. In this case, compile the DUT using the `ixcom` command and the testbenches using the `xrun` command.

© 2021 All Rights Reserved.

If the DUT and testbench are in separate hierarchy, compile the DUT hierarchy using IXCOM and the testbench hierarchy using Xcelium. After that, link the testbench and the IXCOM-instrumented DUT together using the Xcelium elaboration step.

If the DUT and testbench are in the same hierarchy, use the `+xcDesignTop` option of the `ixcom` command.

Following are the advantages of using the separate testbench and DUT compilation (SVTB compilation) method:

- Sharing the DUT with multiple testbenches, reducing compilation time and improving the efficiency of the compilation process:

  Compiling the DUT is a time-consuming process. You might want to compile the DUT with multiple testbench environments without recompiling the DUT every time. If there are multiple testbenches that need to be compiled with the DUT, compilation time increases. Using the SVTB compilation method, you can compile the DUT and the testbench separately, and later link the separately compiled DUT and testbench. Therefore, in case of multiple testbenches, you only need to compile the different testbenches, whereas the DUT needs to be compiled only once.

- Using ***xrun*** to compile HVLs that cannot be mapped to the emulator:

  Modules written in certain HVLs, such as SystemC, ***e***, and Vera cannot be mapped to the emulator, and are, therefore, used for testbenches only. Using the SVTB compilation method or the `+xcDesignTop` option, you can direct IXCOM to skip the compilation of the testbench modules written in these HVLs.

See SystemVerilog Testbench Constructs (SVTB) section for details.

Following is the syntax of the `ixcom` command for compiling the DUT and testbench separately when the DUT design hierarchy is not within the testbench:

```
ixcom -ua +dut+DUT  [options] -nccompile-xmcompile
xrun -f xc_work/xrun.args tb.sv -top Tb [options]
```

Following is the `ixcom` command.with the `+xcDesignTop` option:

```
ixcom [ixcom_options] \
     +xcDesignTop+<hier_ref_to_DUT_instance_in_testbench>=<DUT_module_name>
```

Here, <*hier_ref_to_DUT_instance_in_testbench*> is the instance name of the DUT in the testbench and <*DUT_module_name*> is the name of the top-level module in the DUT.

© 2021 All Rights Reserved.

Compiling Testbenches in Parallel

This section describes the process of compiling testbench as part of IXCOM compilation process by using the `-tbCompile` <*file*> option of the `ixcom` command.

Note: The `-tbCompile` option is case insensitive.

Typically, the following steps are taken to compile the design using IXCOM:

1. Analyze design (DUT) files

   ```
   vlan <source_files>
   vhan <source_files>
   ```

2. Elaborate design (DUT)

   ```
   ixcom -ua +dut +<design_top> ... -xmcompile
   ```

3. Compile testbench and elaborate the final design to include DUT:

   ```
   xrun -elaborate <testbench_files> -top <testbench_top> -f xc_work/xrun.f
   ```

Step 3 in the above process is performed after completing step 2 in a sequential order. In large designs, both steps 2 and 3 can take significant time (many hours). Using the `-tbCompile` option enables the IXCOM compiler to invoke part of these compile stages in parallel to minimize overall compilation time. Therefore, you can specify the command in step 3 above as a part of design compilation in step 2. So, the compilation steps using the `-tbCompile` option are as follows:

1. Analyze design (DUT) files

   ```
   vlan <source_files>
   vhan <source_files>
   ```

2. Elaborate design (DUT)

   ```
   ixcom -ua +dut +<design_top> ... -xmcompile -tbCompile <tbCompile.cmd>
   ```

   Here, the <*tbCompile.cmd*> file includes the following command:

   ```
   xrun -elaborate <testbench files> -top <testbench top> -f xc_work/xrun.f
   ```

The `-tbCompile` option can also be specified in the one-step compilation for SystemVerilog- and Verilog-only designs.

Creating the Testbench Compilation Command File

The testbench compilation command specified in the file that is passed as an argument to the `-tbCompile` option must be executable in the compilation environment used by IXCOM compiler. The syntax of the commands must be compatible to Bourne (sh) shell.

© 2021 All Rights Reserved.

A command can be specified in multiple lines by using the back-slash character (\) for line continuation as required by the shell environment:

```
xrun -elaborate \
     -f xc_work/xrun.f \
...
```

Reference to environment variables is allowed:

```
xrun -elaborate \
     -f xc_work/xrun.f \
     -incdir ${IXCOM_INSTALL_HOME}/share/uxe/etc/ixcom \
...
```

Multiple commands can be specified in this file separated by new line. Empty lines and comment lines in the file for readability. A line starting with “#” symbol is treated as comment line. For example:

```
# Testbench compile script
echo $PATH
xrun -f xc_work/xrun.f \
       -elaborate tb.sv -top Tb \
       -timescale 1ns/1ns
```

The command file can also execute other shell scripts using Bourne (sh) shell. For example:

```
# tbCompile.sh
csh -f tbCompile.csh
```

with C-Shell (csh) commands to do the Testbench compile.

Note: For complete list of command syntax and capabilities allowed by Bourne (sh) shell, refer to the shell command documentation (`man sh`).

Example to Illustrate the Use of the `-tbCompile` Option

This section provides a simple example to demonstrate the use of `-tbCompile` option. The example below uses the `-xcDesignTop` option. The steps are the same for other use models like UA-TBA and UVM that use options like the `-xmcompile` option.

```
// File: tb.sv
module Tb;
parameter Sz = 4;
    reg clk;
    reg [Sz-1:0] din;
    wire [Sz-1:0] dout;

initial
  begin
       clk = 0;
       din = 0;
       #50 $finish;
  end
Dut #(Sz) dut(clk, din, dout);
    always
    #1 clk = ~clk;
    always @(negedge clk)
```


© 2021 All Rights Reserved.

```
    begin
         din = din + 1;
         $display("%m: din = %d, dout = %d [ %b ]", din, dout, dout);
    end
endmodule // Tb

// File: dut.sv
module Dut(clk, din, dout);
    parameter Sz = 4;
    input clk;
    input [Sz-1:0] din;
    output reg [Sz-1:0] dout;
    always @(posedge clk)
    begin
         dout = din + 1;
    end
endmodule // Dut

// File: tbCompile.cmd

xrun -elaborate tb.sv -top Tb -f xc_work/xrun.f
```

Following are the commands to compile the `Dut` design using IXCOM and elaborate with separately compiled `Tb` testbench in the above example.

```
#1 Analyze DUT file(s)
vlan -sv dut.sv

#2 Compile DUT using IXCOM
ixcom -clean -ua +dut+Dut  -top Dut +xcDesignTop+Tb.dut=Dut

#3 Complie Testbench using Xcelium and elaborate with pre-compiled DUT
xrun -elaborate tb.sv -top Tb -f xc_work/xrun.f
```

The same design and testbench can be compiled using the `-tbCompile` option as shown below:

```
#1 Analyze DUT file(s)
vlan -sv dut.sv

#2 Compile DUT using IXCOM and elaborate it with testbench separately using the
-tbCompile option
ixcom -clean -ua +dut+Dut  -top Dut +xcDesignTop+Tb.dut=Dut \
      -tbCompile tbCompile.cmd
```

Log File for the Testbench Compilation Command File

During testbench compilation, the captured data is directed to standard output and error into the `xc_work/tbCompile.log` file. The IXCOM log file includes the time stamp for testbench compilation as it does for other stages as follows:

```
...
// tb compile started at : Thu Oct 12 15:32:22 2017
Please refer to xc_work/tbCompile.log for the TB compile log.
// tb compile ended at : Thu Oct 12 15:32:22 2017
Info [FLOW_SUCCESS]
Successfully built target (tb compile)
...
```


© 2021 All Rights Reserved.

Note: You can redirect output and error messages into a command-specific log file by using the tool option and/or standard Bourne shell syntax. For redirection commands, the log file specified in the commands will have higher precedence over the default log file, which is the `xc_work/tbCompile.log` file. For Xcelium compilation commands, it is recommended that you use `-l <log_file>` option to capture the command output.

Staged Compilation with the -tbCompile Option

If you use `-tbCompile <file>` option for testbench compilation, the commands specified in the `<file>` are executed in a new thread during the `swmodel` stage. You can compile the testbench separately. There are no changes for staged compilation, which is invoked with the `-target <stage>` option.

NOTE: Refer to Compiling Designs in Stages using IXCOM Compiler section for details about staged compilation.

Compiling Top-Level Testbenches Using +xcDesignTop

Consider an example design that includes a DUT with the top module as, *adder* and a testbench, `mytb`.

The DUT is instantiated in the testbench with the name, *mydut* as shown below:

```
module mytb;
    <statements>;
    dutmodule dutinst(a, b, z);
    <statements>;
    <statements>
endmodule
```

To compile using the `+xcDesignTop` option, perform the following steps:

1. Specify the `ixcom` command with the following options to compile the DUT, *adder*:

   ```
   ixcom -ua +dut+dutmodule dutmodule.v +xcDesignTop+mytb.dutinst=dutmodule
   ```

2. Specify the `xrun` command with the following options to compile the testbench separately:

   ```
   xrun -elaborate mytb.v -top mytb -f xc_work/xrun.f
   ```

   In the above command, you can replace the testbench, `mytb.v` with a testbench written in any other HVL if the name of the testbench module remains the same as `mytb`, and the hierarchical path from the testbench to the DUT instance remains the same as `mytb.dutinst`.

© 2021 All Rights Reserved.

Linking DUT with Multiple Testbenches Using Xcelium Tools

The IXCOM compiler generates `xmelab.f` and `xmsim.f` option files in the `xc_work` directory for elaboration and simulation, respectively. These option files are useful when instead of using *xrun*, the individual Xcelium tools - *xmvlog* (or *xmvhdl*), *xmelab*, and *xmsim* are used to compile and elaborate the testbench and other non-synthesizable modules. For example, consider a design with the following hierarchy:

Here, `counter_sw` and `counter_hw` modules are described in VHDL while the other modules are described in Verilog. IXCOM compiles the `hw_top` and `counter_hw` modules for hardware. The `top`, `sw_top`, and `counter_sw` modules are compiled separately using `xmvlog` and `xmvhdl`. Then, the entire design is elaborated together using `xmelab`, followed by simulation using `xmsim`.

Following are the steps of this use flow:

1. Create a library mapping file *`mylibmaprc`* to be used by IXCOM, as shown below.

   ```
   include <IXCOM_INSTALL_HOME>/share/vxe/packages/libmaprc
   dut_vhdl_work  => ./dut_vhdl_work
   dut_vlog_work  => ./dut_vlog_work
   ```

   For more information, see Associating Multiple Logical Libraries to Same Physical Directory on page 620.

2. Analyze the DUT modules using `vhan` and `vlan`.

   ```
   vhan counter_hw.vhd -libmaprc mylibmaprc -work dut_vhdl_work
   vlan hw_top.sv -libmaprc mylibmaprc -work dut_vlog_work
   ```

3. Compile the DUT modules using `ixcom`.

   ```
   ixcom +xcDesignTop+top.hw_top=hw_top -nccompile-xmcompile +dut+hw_top \
         -top hw_top -ua -libmaprc mylibmaprc -work dut_vlog_work
   ```

4. Create *xrun* library file `uxc_cds.lib` to be used by Xcelium tools.

   ```
   DEFINE tb_work ./tb_work
   ```

   Here, `tb_work` is the logical library name and `./tb_work` is physical library directory name.

   Note: The filename `uxc_cds.lib` cannot be changed because it is included automatically in the elaboration and simulation steps.

© 2021 All Rights Reserved.

5. Analyze the testbench modules using `xmvhdl` and `xmvlog`.

```
mkdir tb_work
xmvhdl counter_sw.vhd -f xc_work/xmsim.f -work tb_work
xmvlog sw_top.sv top.v -f xc_work/xmsim.f -work tb_work
```

6. Use `xmelab` to combine and elaborate the testbench and DUT modules, and generate the snapshot.

```
xmelab top -f xc_work/xmelab.f -timescale 1ns/1ns
```

7. Use `xmsim` to simulate the snapshot.

```
xmsim top -f xc_work/xmsim.f -input hw_run.tcl
```

Limitations When Using +xcDesignTop

- The `+xcDesignTop` option does not support bidirectional signals, which require instrumentation of both the testbench and the DUT. Use the `+dump_bidir` option of the `ixcom` command to dump such bidirectional signals that cause IXCOM to quit compilation. Then, use the `+xcBidir` option to generate additional instrumentation to enable support for bidirectional signals with the `+xcDesigntop` option.
- The `+xcDesignTop` option does not support use of the configuration name in the component instantiation statement in the testbench if the top-level DUT module is written in VHDL. You must use an entity name instead of the configuration name. The *<DUT_module_name>* in `+xcDesignTop` must also refer to the entity name.
- This use model is intended to elaborate the testbench (using Xcelium tools) with the pre-compiled DUT (using the IXCOM tools) separately. Therefore, you must address design-unit-binding issues, if any. For example, compilation of testbench files that include re-definition of the DUT module might lead to unintended design-unit binding in the final elaborated design. You must specify explicit binding to resolve such issues by using language or tool capabilities, such as configuration, multiple logical libraries, etc.
- The `NL_HOTSWAP` keyword must be set to `on` if the `+xcDesignTop` option has been used for external netlists. Refer to Importing External Netlists in the SA Mode on page 898 for details about importing external netlists and use of `NL_HOTSWAP` keyword.
- IXCOM compiler modifies `for-generate` scope names in the DUT partition to present compiled design in HDL form for linking (or binding) into testbench hierarchy. If any hierarchical references to DUT objects in the testbench do not match the scope names generated by the IXCOM compiler, use the `-rtlNameForGenerate` option of the `ixcom` command. Refer to Using the -rtlNameForGenerate option for Run-time Access section for details.
- If both DUT and testbench need to import the same package, the same package should not be specified while compiling testbench with Xcelium in the second step, that is the Xcelium compilation step.

© 2021 All Rights Reserved.

- Parameterization of DUT is not supported with the `+xcDesignTop` option.

Support for Bi-Directional Ports (inout) at the DUT-Testbench Boundary for Two-Top-Level Modules

When two independent top-level modules exist for the DUT and the testbench, the DUT top-level module has I/O (inout) ports at the boundary of the software and hardware. This use flow supports bidirectional ports at the hardware-software boundary in two independent top-level modules and the software partition is compiled with Xcelium and the hardware partition is compiled with IXCOM.

Note: Xcelium 19.09_s003, 19.10_a001, or higher versions must be used for this feature.

For this use flow, use the `+xcBidirTBHierRef+<hierrerf>=<DUT_TOP>` option of the `ixcom` command with the `+xcBidir` option.

<*hierrref*> specifies the full testbench-software-side hierarchical path in which the hardware (DUT) side I/O (inout) port is referred using OOMRs inside the testbench-software hierarchy.

<*DUT_TOP*> specifies the RTL top-level unit that is supposed to be a part of the DUT-hardware top of the design partition.

Example of Support for Two-Top-Level Modules with +xcBidirTBHierRef Option

Following is an example that illustrates the `+xcBidirTBHierRef` option.

```
// sw_top.sv: testbench-software top-level module
module sw_top();
    reg enable;
    reg data_in;
...
    assign hw_top.enable = enable;
    assign hw_top.ioPort = ((enable == 1'b1) ? data_in : 'bz); // I/O port reference
in testbench
...
endmodule
```

```
// hw_top.sv: DUT top-level unit which has an I/O port named ioPort
module hw_top( enable, ioPort);
    input wire enable;
    inout ioPort;
    reg w;
...
    sub_dut I(ioPort, enable, w);
...
endmodule
```


© 2021 All Rights Reserved.

```
// sub_dut.sv: DUT hierarchy
module sub_dut(io_port, enable, w);
    inout io_port ;
    input enable;
    input w;
    assign io_port = ((enable == 1'b0) ? w : 'bz);
...
endmodule
```

Following are the commands to compile the above example:

- Two-Step Compilation:

```
ixcom -ua +sv +dut+hw_top +xcBidir \
      +xcBidirTBHierRef+sw_top=hw_top +timescale+1ns|1ns \
      -xmcompile -clean hw_top.sv sub_dut.sv
xrun -elaborate sw_top.sv -top sw_top -f xc_work/xrun.f \
     -timescale 1ns/1ns -sv -access +rwc
```

- Two-Step Multi-Snapshot Incremental Elaboration (MSIE) Compilation:

```
ixcom +dut+hw_top +xcBidir +xcBidirTBHierRef+sw_top=hw_top \
      +timescale+1ns|1ns hw_top.sv sub_dut.sv -ua +sv -xmcompile -clean
xrun -elaborate -access rwc -timescale 1ns/1ns -incrtop \
     sw_top -top sw_top -f xc_work/xrun.f
     -primtop hw_top sw_top.sv
```

To run the above example, use one of the following commands:

```
xrun -R -input <input_tcl>
```

Or

```
xeDebug <input_tcl>
```

- Multi-Step MSIE Compilation

```
#IXCOM Step
ixcom +dut+hw_top +xcBidir +xcBidirTBHierRef+sw_top=hw_top \
      +timescale+1ns/1ns hw_top.sv sub_dut.sv -ua +sv -xmcompile -clean
#HREF Step
xrun top.sv -incrtop top -top top -primtop dut -primtop \
     xc_top -f xc_work/xrun.f -genhref href.txt -cva_bidir_support \
     -ixcom_dut_info xc_work/nc_duttb_spec_file -primtop IXC_GFIFO \
     -primtop ixc_time -primtop _ixc_isc -timescale 1ns/1ns
#PRIM Compile
xrun -elaborate -access rwc -cva_bidir_support \
     -ixcom_dut_info xc_work/nc_duttb_spec_file -mkprimary -f xc_work/xrun.f \
     -incrbind top -href href.txt -snapshot dut_s \
     -top xc_top_externs xc_top_externs.sv
#INCREMENTAL Compile
xrun -elaborate -access rwc -cva_bidir_support \
     -ixcom_dut_info xc_work/nc_duttb_spec_file \
     -primsnap xc_ncwork.dut_s:module -top top top.sv \
     -f xc_work/xrun.f -timescale 1ns/1ns
```


© 2021 All Rights Reserved.

Note: MSIE is an Xcelium feature designed to facilitate re-elaborate small portions of designs that change frequently. Refer to the *Multi-Snapshot Incremental Elaboration* document in the *Xcelium documentation set* for details about MSIE.

Limitations of the Two-Top-Level-Modules Support for Bidirectional Ports

- Multiple DUT top-level modules are not supported. Therefore, multiple hardware top-level modules with bidirectional ports are not supported.
- Mixed-language and VHDL portions are not supported.
- The `-extbind` option is not supported in MPIC and MSIE.
- The `+xcBidirTBHierRef` option should be used only once in the command line and with fixed hierarchy.
- If the DUT top-level modules include multiple bidirectional ports and are used at different hierarchy levels of the testbench, Xcelium cannot find the object during elaboration, and generates an error. Multiple DUT ports used in different scopes are not supported with the `+xcBidirTBHierRef` option.
- Multi-dimensional bidirectional ports having their bits assigned in different testbench-scopes are not supported.
- Multi-dimensional bidirectional ports having port dimensions greater than two in either packed or unpacked or both dimensions are not supported and a warning message is issued.

Determining File Type by Source Filename Extension

In the absence of Verilog mode options (`+v1995`, `+v2000`, or `+sv`), IXCOM determines the file type for a design element at the file level, based on the filename extension. Each recognized file type has a built-in, predefined set of file extensions. For each file type, there is a command-line option that you can use to change, or add to, the list of file extensions mapped to a given language.

Options for Verilog Design Compilation Mode

- `+sv`: Forces SystemVerilog compilation
- `+v1995`: Turns off new Verilog-2001 keywords
- `+v2000`: Enables new Verilog-2001 keywords

© 2021 All Rights Reserved.

Options for File-Extension-Based Compilation Mode

- `-default_ext` <*file_type*>: Specifies Verilog file type or language version for unrecognized file extension(s)
- `-sysv_ext` [+]<*extns*>: Adds or overrides extensions for SystemVerilog sources
- `-vlog95_ext` [+]<*extns*>: Adds or overrides extensions for Verilog95 sources
- `-vlog_ext` [+]<*extns*>: Adds or overrides extensions for Verilog sources

The optional + symbol is used to add new extensions to the previous list of options. `<extns>` indicates one or more file extensions. Use a comma to separate multiple extensions.

For example:

```
-vlog_ext .v2k,.vLog
-vlog_ext +.v2k,.vLog
```

If any of the Verilog mode options are specified, it means that you have explicitly specified the language (subset) to be used. Then, file-extension based compilation is not used.

The following table shows the recognized file types, the set of built-in, predefined file extensions for each language type, and the command-line option you can use to change, or add to, the list of file extensions mapped to a given language.

File Type	File extension	Compiler option
Verilog	`.v, .V, .vp, .VP, .vs, .VS`	`-vlog_ext`
Verilog 1995	`.v95, .V95, .v95p, .V95P`	`-vlog95_ext`
SystemVerilog	`.sv, .SV, .svp, .SVP, .svi, .svh, .vlib, .VLIB`	`-sysv_ext`

For example, the default file extensions for Verilog files are `.v`, `.vp`, `.vs`, `.V`, `.VP`, and `.VS`. For Verilog files with other extensions (for example, `.rtl` and `.vg`), it is essential to specify that these are valid extensions for Verilog files. Do this with the `-vlog_ext` option.

Additionally, the `-default_ext` option can be used to specify the file type for the respective filename extensions that are not specified by the built-in or user-specified extensions.

Note: By default, if no corresponding file type is set for a filename extension, the file type is set to Verilog-2001.

© 2021 All Rights Reserved.

The argument to the `-default_ext` option is a string that represents a file type. For example, if the string specified with the `-default_ext` option is `systemverilog`, the file type is taken as SystemVerilog HDL.

Consider the following example that illustrates the filename-based compilation in IXCOM:

```
// File: top.sv
    reg [3:0] r;
    module Top; // sv module
           initial $unit::r = 11;
           Mid mid();
    endmodule // Top
// File: mid.v
    //
    module Mid;
           localparam p=1;
           Bot bot();
    endmodule // Mid
// File: bot.v95
    //
    module Bot;
           reg [3:0] int;
           inital int = 2;
    endmodule // Bot
// File: tip.sv
    // shares same compilation-unit
    module Tip;
           initial
           #1 display ($unit::r);
    endmodule // Tip
```

The following `xrun` command is used for the above example:

```
xrun top.sv mid.v bot.v95 tip.sv
```

Then, the Verilog version for each of the modules in the above example is determined as follows:

- `Top` - Verilog version set to SystemVerilog
- `Mid` - Verilog version set to Verilog-2000
- `Bot` - Verilog version set to Verilog-1995
- `Tip` - Verilog version set to SystemVerilog

Details of setting file types based on filename extensions are provided in the *Changing the Default Set of File Extensions* section of the *Customizing xrun* chapter of the *xrun User Guide* provided as a part of the Xcelium documentation set.

© 2021 All Rights Reserved.

Compiling Behavioral Constructs

This section describes the way IXCOM behavioral compiler extends the Verilog RTL subset in the DUT for acceleration in an emulator.

During behavioral compilation, the supported behavioral `always` procedures in the DUT are mapped into synthesizable RTL-compliant `always` procedures. Behavioral compilation is not enabled by default. You need to explicitly specify the modules that should undergo behavioral compilation. Other IXCOM transformations, such as interface compilation or support for SystemVerilog assertions, can also internally enable behavioral compilation with selective support.

A Verilog or SystemVerilog `always` procedure is a candidate for behavioral compilation if all of the following conditions hold true:

- The `always` procedure has only explicit sensitivity list.
  - An `always_comb` or `always_latch` procedure does not have a sensitivity list, and is, thus, unqualified for behavioral compilation.
  - An `always @(*)` procedure has an inferred sensitivity list, and is, thus, unqualified for behavioral compilation.
- The `always` procedure contains only synthesizable RTL and supported behavioral constructs.
- There is no hierarchical reference to any register or data type declared inside the behavioral `always` procedure.

Specified behavioral modules are transformed one at a time and each behavioral procedure is mapped into a synthesizable RTL-compliant procedure, which has the following basic characteristics:

- All `@` events in a single `always` procedure are identical and synthesizable.
- All supported behavioral constructs are replaced with synthesizable operations.
- The number of operations between consecutive `@` events are bounded and can be completed in a single internal behavioral clock cycle.

Enabling Behavioral Compilation

To enable behavioral compilation for a module in the DUT, use either of the following options:

© 2021 All Rights Reserved.

- Specify the module with the `+bc+<mod_name>` option of the `ixcom` command. The `<mod_name>` refers to the name of the DUT module that should undergo behavioral compilation.

  For example:

  ```
  ixcom test.sv +dut+top +bc+test1 +bc+test2 ...
  ```

  With the above command, behavioral compilation will be enabled for the DUT modules `test1` and `test2` inside DUT `top`.

- Add the following `map_bc` directive in an `initial` statement within the required module:

  ```
  $ixc_ctrl("map_bc");
  ```

Supported Behavioral Constructs

Behavioral compiler extends the Verilog RTL subset in the DUT. This section describes the extended non-RTL constructs and their basic restrictions. Behavioral extensions are limited to non-RTL control constructs and static data types. Dynamic data types, such as class objects and dynamic arrays are not supported by behavioral compiler.

Non-RTL and Multiple Events

Behavioral compilation supports `always` procedure with general non-RTL sensitivity list and multiple events as listed below:

- Event control (@) statements

  An event control statement can have any number of event expressions, edge-sensitive events, level-sensitive events, or both.

  - For edge-event expressions, the operand of posedge or negedge should be a 1-bit scalar or a 1-bit constant bit select of a simple bit vector.
  - For level-sensitive events, the following conditions must hold true:
    - The event expression should not contain any function calls.
    - The data type of the expression should be either a 1-bit scalar or a packed bit-vector type.

  Example:

  ```
  wire wclk;        // 1-bit scalar
  reg rclk;         // 1-bit scalar
  reg [3:0] clks;   // simple bit vector
  always begin      // multiple event controls supported
       @(wclk)...            // 1-bit scalar supported
       @(clks[3:1]) ...      // simple bit vector type supported
  ```


© 2021 All Rights Reserved.

```
     @(posedge rclk) ...   // 1-bit scalar supported
     @(negedge clks[0]     // 1-bit const bit-select of packed bit vector
                           //supported
     or wclk)...           // mixed edge- and level-sensitive events supported
end
```

- Intra-assignment event control

  Intra-assignment event control is supported in blocking assignments. It is not supported in non-blocking assignments.

- Wait statements

  The `wait` expression must not contain any function calls.

```
integer x;
function integer f1; ...
always begin
      wait(x);        // wait expression supported
      wait(f1(x)) ... // wait expression containing function call not supported
....
end
```

  Note: An `always` procedure beginning with a `wait` statement as the example above is supported. However, it can cause a race condition leading to hardware-software mismatch. Avoid using such an `always` procedure in DUT.

- Multiple events

  - An `always` procedure can have any number of event-control statements. Event expressions can be identical or distinct.
  - An `always` procedure can have any number of `wait` statements

  Example:

```
always begin
   wait(x) ...
   if (!done) @(negedge y);
   @(posedge clk) ...
end
```

- An asynchronous `always` procedure that does not reference any objects in its sensitivity list is transformed as a behavioral process to honor the specified event expressions.

  - If any of the event expressions are referenced in a pure-RTL asynchronous `always` procedure, the `always` procedure is not transformed, and follows combinational logic behavior during synthesis. This means that the event expressions would be inferred from actual references.
  - If none of the event expressions are referenced in the `always` procedure, the `always` procedure follows the LRM semantics in accordance to the specified event expressions.

© 2021 All Rights Reserved.

Example:

```
always @(a, b, e) // event expressions referenced in always
   d = a + b + c; // no transformation: combinational logic behavior with
                  // inferred @(a,b,c)
always @(x)       // event expression not referenced in always
    h = f & g;    // will be transformed: no change to sensitivity list @(x)
```

Timing Controls

Time-delay control statements (`#delay`) and intra-assignment delays in blocking assignments are supported through interface compilation with the `+iscDelay` option or `map_delays` directive if the time delays are a part of a supported clock model. For more information, refer to Enabling Interface Compilation in IXCOM and Manually Modeling Clocks in SA Mode sections.

Multiple Drivers

Assignments to the same register in multiple `always` procedures are supported when these are either all blocking or all non-blocking assignments.

Passing a register as an output or inout argument to a subroutine call is considered as a blocking assignment to the register.

Avoid mixing blocking and non-blocking assignments for the same register because it might result in compilation errors during the IXCOM compilation and HDL-ICE Compiler synthesis phase and/or lead to simulation and emulation mismatch.

When multiple `always` procedures are active and drive the same register at the same time, in emulation, one of the `always` driver wins and overrides the driving values from all other active drivers. Typically the lexically last `always` procedure in the source code is the winning driver.

For example, suppose you have two `always` procedures writing to register counter as follows.

```
always @(posedge clock1) begin
   counter = counter + 1;   // driver 1
   $display("Driver 1: ", counter);
end
always @(posedge clock2) begin
   counter = counter + 2;   // driver 2
   $display("Driver 2: ", counter);
end
```

If only driver `1` is active, the register counter is incremented by 1. If only driver `2` is active, the register counter is incremented by 2. However, when both drivers are active, only one driving

© 2021 All Rights Reserved.

value persists. If the current value of counter is `10` and both drivers become active, then driver `1` displays `11` and driver `2` displays `12`, but the final value of counter is either 11 or 12, depending on the implementation.

To specify that a register should be serialized when there are multiple drivers, use the following `bcs` directive in an `initial` statement in the behavioral module.

```
$ixc_ctrl("bcs" [, <var_name>]+);
```

Suppose, the following `bcs` directive is used in the counter example:

```
initial $ixc_ctrl("bcs", counter);
```

A serialized ordering of driver `1`, followed by driver `2` would result in driver `1` displaying `11` and driver `2` displaying `13` with the final value of counter as `13`.

Named Event

Named event variable declarations in the DUT are supported if:

- It is a named event variable or a one-dimensional event array declared in the module scope of a behavioral module (declaration in all others scopes are not supported).
- The only operations supported in DUT for the named event are:
  - Triggering a named event via the -> operator.
  - Triggering a named event via the ->> operator without any delay or event control.
  - Waiting for a named event via the @ event control operator.

Triggering a Named Event

- A named event trigger (`->`) is supported in DUT if the named event is declared in a behavioral module in the DUT.
- Non-blocking event trigger (`->>`) without delay or event control is supported in DUT as a blocking event trigger (`->`), and the same restrictions apply.
- Triggering a named event declared in DUT from testbench procedures is supported without restrictions. This applies for all (`->`) and (`->>`) trigger operations.

Note: It is important that the behavior of the named event should be “glitch-free”. This means that a named event should not be triggered more than once (via `->` or `->>`) in a single time step. Behavior is unspecified in DUT for multiple zero-delay triggers.

© 2021 All Rights Reserved.

Example

```
module top;  // TB
dut d(clk);
 ...
always begin
   ...
   -> d.e; // event trigger supported in TB for DUT named event
   ->> d.ev[0]; // nonblocking event trigger supported in TB for DUT named event
   ->> @(clk) d.ev[1]; // nonblocking event trigger with @event supported in TB
   ->> #1 d.ev[2]; // nonblocking event trigger with #delay supported in TB
end
endmodule

module dut(input clk); //DUT
    event e; // event variable supported
    event ev[4]; // one-dimensional event array supported
    always begin
        ...
        ->e; // event trigger supported in DUT for locally declared named event
        ->top.d.ev[0]; // event trigger supported in DUT for DUT named event
                       // trigger via hierarchical ref
        ->> ev[1];     // nonblocking event trigger supported in DUT,
                       // treated as ->ev[1]
        ->> @clk ev[2]; // nonblocking event trigger with @event not supported
                       // in DUT
        ->> #1 ev[3]; // nonblocking event trigger with #delay not supported
                      // in DUT
    end
endmodule
```

Waiting for a Named Event

- Waiting for a named event is supported in DUT if the named event is declared in a behavioral module in the DUT.
- Waiting for a non-constant select of a named event array is not supported.
- Waiting for a named event declared in DUT from testbench procedures is supported.

```
module top; // TB
dut d(clk);
 ...
always @(top.d.e) begin // waiting on DUT named event supported in TB
   ...
end
endmodule

module dut(input clk); // DUT
event e;       // event variable supported
event ev[3];   // one-dimensional event array supported
always begin
    ...
    @ev[0];    // waiting on locally declared named event supported in DUT
    @top.d.e;  // waiting on DUT named event via hierarchical ref supported
               // in DUT
end
endmodule
```


© 2021 All Rights Reserved.

Control Flow

Following is the support provided for dynamic loops, fork-join statements, and disable statements:

- Loops with dynamic loop bounds are supported. Each iteration is executed using a behavioral evaluation.

```
always begin
   ...
   while (a > b) ... // dynamic loop bound supported
end
```

- Verilog `fork-join` statements are supported when there is no delay or disable inside the `fork-join`. Only the control option `join` is supported. Parallel statement containing `disable` statements to a scope external to that parallel statement is not supported. Parallel statement containing delay control is not supported. All variables declared inside a `fork-join` block would be treated as if these are declared with static lifetime.

```
begin:outer
  fork
    begin:inner
      if (...) disable outer; // disable scope outside parallel stmt not
                              // supported
      if (...) disable inner; // disable scope internal to parallel stmt
                              // supported
      ...
    end

  #2 ...;                     // delay control not supported
  join                        // join supported
                              // (join_none or join_any not supported)
  end
```

- Only disable statements of a parent named scope are supported. Further restrictions for fork-join parallel statements are listed in the above bullet point.

```
module dut;
   always @(posedge clk) begin
      ...
      begin:x
         ...
         if (...) disable x;  // disable parent scope supported
            ...
      end
   end
   always @(negedge clk) begin
      ...
      if (...) disable dut.x; // disable non-parent scope not supported
         ...
   end
endmodule
```


© 2021 All Rights Reserved.

Subroutine Calls

Following points highlight the way subroutine calls are handled and the restrictions associated with subroutine calls:

- Subroutine calls from an `always` procedure in a behavioral module shall be inlined into the `always` procedure. The inlined task or function and its locally declared variables would be treated as these are declared with automatic lifetime.
- The data type of the formals, the arguments, and the locally declared variables must be packed and have static width.
- If a function is specified with `hw_recursive` and other config setting(s), such as GFIFO, TB_IMPORT and so on, IXCOM issues the following error:

  ```
  <function name> specified with hw_recursive is in conflict with other
  config setting(s).
  ```

  In such cases, examine the configuration settings of the target function to eliminate the conflict.

Support for Recursive Subroutines

A recursive subroutine is a function or task that calls itself directly or indirectly (through a call chain). A function that calls itself directly is called a *self-recursive function*, and a function that calls itself indirectly is called an *indirect recursive function*.

In the following example, function multiply is a *self-recursive function*, and functions foo and bar are *indirect recursive functions*.

```
function int multiply(int aa, int bb);
    int cc;
    if (bb == 1)
        cc = aa;
    else begin
        cc = multiply(aa, bb - 1) + aa;
    end
    multiply = cc;
endfunction

function int foo(int aa, int bb);
    int cc;
    if (bb == 1)
        cc = aa;
    else begin
        cc = bar(aa, bb - 1) + aa;
    end
    foo = cc;
endfunction

function int bar(int aa, int bb);
    bar = foo(aa, bb);
endfunction
```


© 2021 All Rights Reserved.

IXCOM does not support self-recursive subroutines in DUT by default. You need to specify directives in one of the following three ways to enable support for self-recursive subroutines in DUT:

- Specify the name and path of the function using the `+hw_recursive` option of the `ixcom` command:

  ```
  ixcom ... +hw_recursive+<module name>.<function name>
  ```

- Specify the name of the function in an `initial` statement in the module of the function:

  ```
  $ixc_ctrl("hw_recursive", "<function name>");
  ```

- Specify the function in the config side file in the following format and specify the config file using the `+tfconfig+<file_name>` option of the `ixcom` command:

  ```
  <module_name> <function name> hw_recursive
  ```

In addition, to control code explosion and reduce compilation time, you can specify a global limit on the iterations of recursive call inline by using the `+hw_recurs_limit+<N>` option of the `ixcom` command, where `<N>` is the limit:

```
ixcom ... +hw_recurs_limit+<N>
```

Set `<N>` to `0`, if you do not want to impose any limits on the level of inline. If the option is not specified, the default limit is `8`.

All the hardware non-recursive calls are inlined before the hardware self-recursive function calls are processed. Thus, the limit only applies to the inline of self-recursive function calls. Similar to loop-unrolling, inline of self-recursive function calls is performed iteratively. Starting from a process containing a self-recursive function call, the call is inlined with its function definition, and if there is any dead code, it is eliminated. If there is still a self-recursive call in the inlined code, another level of inline is performed. This iteration continues until all self-recursive function calls are inlined. Thus, the limit applies to the total iterations of inline; not on a specific self-recursive function call. In the last example, it requires at least 3 iterations to fully inline the self-recursive call `multiply(A, 3)`. Thus, you need to set the limit to `3` or greater, or to `0` indicating no limit by using the `+hw_recurs_limit` option.

Self-recursive calls are handled differently by IXCOM and HDL-Compiler. Suppose, the multiply function from the previous example is used in DUT as shown below:

```
module DUT(
   input clk,
   input reset,
       input    [3:0] A,
       input    [3:0] B,
      output   [6:0] product );
function int multiply(int aa, int bb);
      int cc;
      if (bb == 1)
          cc = aa;
      else begin
          cc = multiply(aa, bb - 1) + aa;
```


© 2021 All Rights Reserved.

```
      end
      multiply = cc;
endfunction
int C;
       assign product = C;
       // Reset
      always @(posedge reset) begin
           C <= 0;
      end
// Multiplication
      always @(posedge clk) begin
                   C = multiply(A, 3);
                   $display("@%t multiply(%d, %d) = %d", $time, A, 3, C);
       end
endmodule
```

In the above example, the multiply function is a self-recursive function call. But, because of the arguments to the function, the call will always terminate at the third time of recursion. HDL-ICE Compiler attempts to inline self-recursive calls. If the recursive call can be fully inlined within a small number of iterations at compile time, the self-recursive call is supported. Else, HDL-ICE Compiler generates an error.

IXCOM supports a self-recursive call if it is:

- A self-recursive function or task (no indirect recursive function or task)
- Verilog only (no VHDL)
- An automatic function (no static function)
- Local call (no XMR)
- Not in a package or compilation unit

If a self-recursive subroutine is to be executed in hardware, it needs to be specified at the compile time, and the RTL subset checking will skip checking on self-recursive subroutine call. Then, based on the nature of various IXCOM transformations, a transformation (for example, multi-event transformation, bounded queue support, hardware random, `+gfifoDisp`, and so on) might invoke the inline routine to inline those self-recursive calls. Also, for a behavioral module, the inline routine is invoked to inline self-recursive calls in its procedural blocks. So, the subroutine inline in IXCOM is performed on an on-demand basis, and the self-recursive calls that are not handled by IXCOM are passed down to HDL-ICE Compiler to handle.

When inline self-recursive calls in IXCOM are invoked, the compilation can be successful if all recursive calls are fully inlined at compilation time and there are no more recursive calls. Else, the compilation fails and a self-recursive call cannot be fully inlined at compilation time, which could be due to one of the following reasons:

© 2021 All Rights Reserved.

- The recursive call cannot be fully inlined within a configurable limit of iterations at compile time. In such cases, IXCOM issues an error.
- The recursive call can never be fully inlined at compile time, which might be because the passed arguments are not constants and, thereby, restrict the terminating condition of recursion from being determined at compile time. In such cases, IXCOM issues an error before the limit is reached.
- If a function is specified with `hw_recursive` and other config setting(s), such as GFIFO, TB_IMPORT and so on, IXCOM issues the following error:

  ```
  <function name> specified with hw_recursive is in conflict with other
  config setting(s).
  ```

  In such cases, you should examine the config settings of the target function to eliminate the conflict.

Other Restrictions

Following are other miscellaneous restrictions on a behavioral module:

- The enumerated data type is not supported if declared in a block name space in an `always` procedure, or in a block name space in a subroutine reachable from an `always` procedure through subroutine calling.

  ```
  always begin
      ...
      t1(...);
      begin:x   // block name space in always procedure
         typedef enum {r, g, b} color;  // enum declaration not supported here
         color c;
         ...
      end
  end
  task t1(...);  // task reachable by always procedure (reachable task)
  begin
      ...
      begin:y   // block name space in reachable task
         typedef enum {r, g, y} color;  // enum declaration not supported here
         color c;
         ...
      end
  end
  endtask
  ```

- Named blocks and variables declared in an `always` procedure or in a subroutine reachable from an `always` procedure in a behavioral module, are not supported if there are any hierarchical references to them.

  ```
  module dut;
    always begin
       ...
  ```


© 2021 All Rights Reserved.

```
      t1(...);
      begin:x   // named block in always procedure
        reg a;
        ...
      end
  end
  task t1(...);  // task reachable by always procedure (reachable task)
  begin
      ...
      begin:y   // named block in reachable task
        reg b;
        ...
      end
  end
  endtask
  always begin
      ...
      if (dut.x.a) disable dut.x; // hierarchical ref to x or x.a not supported
      if (dut.t1.y.b) disable dut.t1.y;  // hierarchical ref to y or y.b not
                                         // supported
      ...
  end
endmodule
```

- Labeled statements declared in an `always` procedure or in a subroutine reachable from an `always` procedure in a behavioral module, are not supported if there are any hierarchical references to them.

```
module dut;
always begin
    ...
    x:begin  // labeled statement in always procedure
      t1(...);
      ...
    end
end
task t1(...);  // task reachable by always procedure (reachable task)
begin
   y:begin  // labeled statement in reachable task
     ...
   end
end
endtask
always begin
   ...
   if (...) disable dut.x; // hierarchical ref to x not supported
   if (...) disable dut.t1.y;  // hierarchical ref to y not supported
    ...
end
```

- Synopsys `translate_on` / `translate_off` statements in a behavioral module:
  - `translate_off` statements are not supported in an `always` procedure and subroutines reachable from an `always` procedure through subroutine calling when:

© 2021 All Rights Reserved.

- The `translate_off` statement is time-consuming.

  Or,

- The `translate_off` statement affects the control flow of `translate_on` region, for example, a `translate_off` jump statement such as a `continue` statement of a `translate_on` loop statement.

- Cadence recommends that the `translate_on/off` statement should be used for the entire `always` procedure.

```
always @(posedge clk) begin
   while (a > b) begin
    //synopsys translate_off
    if (...) @(posedge clk) ... // time-consuming translate-off not
                                // supported
    if (...) continue;          // jump stmt affecting control flow not
                                // supported
    //synopsys translate_on
    ...
    end
end
```

Scheduling Behavioral Processes

Every change in signal value in the design is considered an update event. The `always` procedures are sensitive to update events of specific signals. This sensitivity is defined, for example, explicitly with event control statements like `@` or implicitly with the `wait` statements. When an update event occurs, all the `always` procedures that are currently sensitive to the signal of the update event are triggered to be executed. In simulation, the triggered `always` procedures are executed in an arbitrary order and read the current (non-sampled) signal values. In emulation, the triggered RTL `always` procedures are executed concurrently. When triggered by a synchronous edge event, such as `posedge clk`, the `always` procedure reads the sampled signals values. When triggered by an asynchronous non-edge event, such as `X` or an asynchronous edge event such as `posedge reset`, the `always` procedure reads the current non-sampled signal values. For the triggered behavioral processes, it might be necessary to delay execution to a subsequent Behavioral Evaluation (BEVAL) and/or spread the execution over multiple BEVALs. When the execution is delayed, the update values from blocking assignments are visible. The update values from the non-blocking assignments are also visible unless behavioral compiler performs additional transformations.

Behavioral process scheduling can delay the execution of a behavioral process for two reasons:

- **When the signal of a triggering update event is a delayed event**: You can specify an event to be a delayed event with the IXCOM `delayed_event` directive. Else, the compiler infers an event as a delayed event for a particular behavioral process if it is used as an asynchronous event signal.

© 2021 All Rights Reserved.

- **When you specify serialized groups using behavioral compiler serialization variables (marked with the IXCOM `bcs` directive) in which only one behavioral process from each group can be executed at a time**: The behavioral processes that cannot be executed concurrently with a particular behavioral process are referred to as its ***serialized behavioral processes***. Behavioral processes without any assignment to serialization variables have no serialized behavioral processes. When multiple behavioral processes in the same serialized group are ready to be executed at the same time, the one with the higher execution priority (typically based on the lexical program order) execute first. Delaying execution of a behavioral process is necessary when among its serialized behavioral processes, there is one already executing or one with a higher priority ready to be executed. In this case, execution is delayed until all its active and higher priority serialized behavioral processes become idle.

There are three basic states of a behavioral process:

- **Idle**: This is the state when the behavioral process is waiting for update event(s).
- **Delayed**: This is the state when the behavioral process is ready to be executed but is delayed due to a delayed event or conflict with its serialized behavioral processes.
- **Active**: This is the state when the behavioral process has started execution but requires more BEVALs to finish.

Behavioral evaluation is extended unless all behavioral processes are in the **Idle** state. A behavioral process is considered to be ready for execution if it is in the **Delayed** State, or if the behavioral process is triggered, but not by a delayed event.

Conditions for transitions from current state to next state are described below.

If the current state of the behavioral process is Idle:

© 2021 All Rights Reserved.

- If there is no triggering update event **[1]**, the behavioral process stays in the **Idle** state.
- Else, if it is triggered by a **delayed event [2]**, it transitions to the **Delayed** state.
- Else, if none of its serialized behavioral processes are **Active** or ready with higher priority **[3]**:
  - The behavioral process is executed for one BEVAL.
  - If the behavioral process requires more BEVALs to finish **[4]**, it transitions to the **Active** state.
  - Else, it transitions to the **Idle** state.
- Else, it transitions to the **Delayed** state.

If the current state of the behavioral process is Delayed:

- If none of its serialized behavioral processes are **Active** or ready with higher priority **[3]**:
  - The behavioral process is executed for one BEVAL.
  - If the behavioral process requires more BEVALs to finish **[4]**, it transitions to the **Active** state.
  - Else, it transitions to the **Idle** state.
- Else, it stays in the **Delayed** state.

If the current state of the behavioral process is Active:

- The behavioral process is executed for one BEVAL.
  - If the behavioral process requires more BEVALs to finish **[4]**, it transitions to the **Active** state.
  - Else, it transitions to the **Idle** state.

Detection of Behavioral Process Events

By default, behavioral compiler treats all asynchronous events as delayed events, and determines if a particular edge event requires delay based on its references inside the behavioral process. If there is a read of the edge-event signal in the behavioral process, it is identified as a delayed event. You can also specify all events of a signal to be immediate or delayed by marking the signal with the IXCOM `immediate_event` and `delayed_event` directives, respectively. Consider the following example of a behavioral process:

```
initial $ixc_ctrl("delayed_event", start);
initial $ixc_ctrl("immediate_event", z);
```


© 2021 All Rights Reserved.

```
always @(posedge clk, // immediate edge event: posedge of clk
         posedge rst) // delayed edge event due to read: posedge of rst
if (rst) begin
       x = a;
end
else begin
     @(posedge start); // delayed edge event due to directive: start
          x = b;
          @(y); // delayed non-edge event: y
          x = y;
          @(z); // immediate non-edge event due to directive: z
          x = b;
end
```

If the above behavioral process does not have any serialized behavioral processes, execution can only be delayed due to a delayed event. When this behavioral process is triggered by the immediate events, `@(posedge clk)` or `@(z)`, execution would be immediate and the behavioral process would read sampled signal values. When this behavioral process is triggered by any of its delayed events, execution would be delayed for one BEVAL and the behavioral process would read signal values at the time of execution.

The IXCOM `delayed_event` and `immediate_event` directives apply to all the events of the specified signal in the module. If the different behavioral processes need to have varying behavior, create separate equivalent signals to specify it differently. For example:

```
assign x_imm = x;
initial $ixc_ctrl("immediate_event", x_imm);
always @(x) // delayed non-edge event: x
       begin ... end
always @(x_imm) // immediate event due to directive: x_imm
       begin ... end
```

Identification of Behavioral Delayed Processes

When a signal is updated with a non-blocking assignment, its updated value is not visible to the `always` procedures that are triggered or executed at the same time. However, when execution of an `always` procedure is delayed, the updated values do become visible if the signal is referenced in the delayed `always` procedure. An `always` procedure for which execution might be delayed to behavioral evaluation is referred as ***Behavioral Delayed Process***.

A Behavioral Delayed Process can be:

- A behavioral process with delayed events.
- A behavioral process with serialized behavioral process for which execution might be delayed because of the need for serialization.
- A behavioral process that might take multiple BEVALs to finish, such as a behavioral process containing dynamic loops.

© 2021 All Rights Reserved.

- An `always` procedure that waits for an event signal, which is driven in Behavioral Delayed Process. Delayed update events lead to delayed triggering and execution. This can be a behavioral process or RTL always procedure.

An example module is shown below with different types of Behavioral Delayed Processes.

```
module test (input clk1, input clk2, input ack,
             input int n, input int a, input bit din);
       bit w; // BA-driven in BDP #1, use as event in BDP #3 and #4
       int x; // BA-driven in BDP #3, use a value in BDP #3
       int y; // BA-driven in BDP #2, use as value in BDP #2 and #4
       int z; // BA-driven in BDP #4
       int b; // NBA-driven in BDP #1, use as value in BDP #1 and assign #0
       int c; // NBA-driven in always #5, use as value in always #5 and assign #0
       int d; // NBA-driven in always #6, use as value in always #6 and assign #0
       int e; // NBA-driven in always #6, use as value in always #6
       reg [1:0] m; // Non-driven, use as value in BPD #3
       int i;  // loop index
       wire [31:0] f; // driven in assign #0, use as value in BDP #4
    // assign #0
    assign f = a + b + c + d;
    initial $ixc_ctrl("delayed_event", ack);
    // BDP #1: BP with delayed_event
    always @(posedge clk1) begin  // immediate edge event
      w = 0;
      @(ack); // delayed asynchronous event
      w = din;
      b <= b + 1;
    end
    // BDP #2: BP with multiple BEVALs due to dynamic loop
    always @(posedge clk1) begin // immediate edge event
      for ( i = 0; i < n; i = i + 1)   // dynamic loop
         if (a[i]) y = y + 1;
    end
    // BDP #3: BP with event signal w driven in BDP #1
    always @(posedge w) begin // immediate edge event
         @(posedge clk1); // immediate edge event
         x = x + m;
    end
    // BDP #4: RTL always procedure with event signal w driven in BDP #1
    always @(posedge w) begin
         z = f - y;
    end
    // always #5: BP only, not BDP
    always @(posedge clk1) begin // immediate edge event
         @(posedge clk2); // immediate edge event
         c <= c + 1;
    end
    // always #6: simple RTL always procedure, not BDP
    always @(posedge clk1) begin
         d <= d + 1;
         e <= e + 1;
    end
endmodule
```

To preserve the pre-nonblocking-assignment or sampled values of signals needed for Behavioral Delayed Processes, behavioral compiler transforms each behavioral module in two ways:

© 2021 All Rights Reserved.

- First, for signals that are locally driven in the module, behavioral compiler transforms the module to delay non-blocking assignment updates until all Behavioral Delayed Processes are **Idle**, waiting for update events.
- Second, for non-driven signals, behavioral compiler determines the signals that should be treated as driven by non-blocking assignments and explicitly replaces their uses with sampled copies in the module.

Behavioral Delayed Process Signals

A signal is considered a Behavioral Delayed Process signal if it is either directly used in a Behavioral Delayed Process or if it is a fan-in signal to a Behavioral Delayed Process signal. This includes the signals used as event signals as well as the ones used as read values in a Behavioral Delayed Process. In the example module, following are the Behavioral Delayed Process signals:

- Signals used directly in Behavioral Delayed Processes:

```
BDP #1
    use as event: clk1, ack
    use as value: din, b
BDP #2
    use as event: clk1
    use as value: i, n, a, y
BDP #3
    use as event: w, clk1
    use as read: x, m
BDP #4
    use as event: w
    use as read: f, y
```

- Fan-in signals to BDP signals:

```
assign #0 to f
    use as read (as f): a, b, c, d
```

Behavioral compiler might handle a Behavioral Delayed Process signal differently based on its signal type, datatype, driver type, and usage in Behavioral Delayed Processes in the module. Behavioral compiler considers the following categories:

- Signal Types:
  - Port signal: signal from module ports
  - Local signal: signal declared in module
  - XMR signal: signal referenced via hierarchical path
- Datatypes:
  - Packed datatype

© 2021 All Rights Reserved.

    - Without named components, such as `reg`, bit vector, or packed array
    - With named components, such as enumerated type or packed struct or union
  - Unpacked datatype, such as unpacked array or unpacked struct or union
- Driver Types:
  - Nonblocking-assignment-driven: Has any local nonblocking-assignment driver(s)
  - Blocking-assignment-driven: Has only local blocking-assignment driver(s) and no local nonblocking-assignment driver
  - Non-driven: Has no local driver
- Behavioral Delayed Process Usage
  - Use only as event signal
  - Use only as read value
  - Use as both event signal and read value

Delaying Local Nonblocking-Assignment-Signal Updates

To preserve the pre-nonblocking-assignment values of the Behavioral Delayed Process signals that are locally driven, behavioral compiler transforms all of the `always` procedures that contain the nonblocking-assignment drivers of Behavioral Delayed Process signals so that the actual update can be delayed until all Behavioral Delayed Processes are Idle. Only the nonblocking assignments in behavioral processes and the synchronous or clocked `always` procedures are candidates for such delayed-driver transformation. All the asynchronous or combinational drivers are considered as blocking updates even if non-blocking assignments are used. For example, a nonblocking assignment in an `always @(*)` procedure as shown below would not be delayed or transformed by behavioral compiler.

```
always @(*) begin
    if (p) q <= 0;
    else q <= r;
end
```

At the time of the nonblocking assignment, the information of the assignment is captured; the actual signal update is not done. Behavioral compiler transforms each nonblocking assignment to the local driving value with enable. When there are no more active Behavioral Delayed Processes, the driving value and enable are used to update the corresponding Behavioral Delayed Process signal. This is similar to multiple driver transformation, except that the time of updating the signal is controlled or delayed by the scheduling of the behavioral process.

© 2021 All Rights Reserved.

Restrictions Related to Non-Blocking-Signal Transformations

- Following are the restrictions in the way in which behavioral compiler performs delayed nonblocking-assignment transformations for the `always` procedures:
  - Behavioral processes are supported.
  - For asynchronous `always` procedures, nonblocking assignments are treated as blocking assignments:

    ```
    always_comb, always_latch
    always @(*), always @(a, …)
    ```

  - Synchronous RTL `always` procedures are supported provided these conform to the basic RTL synthesizable template in which the clocked and asynchronous regions can be clearly identified.
  - For `always` procedures for SFIFO, nonblocking assignments are treated as blocking assignments.
- Following are the restrictions in the way in which behavioral compiler supports the SystemVerilog datatype of nonblocking-assignment signals:
  - The packed objects are supported as follows:
    - No restrictions for port and local signals
    - For XMR signals, packed datatype without named components are supported. So, there should be no enum, struct, or union.
  - Unpacked objects like the following are not transformed:
    - Verilog memory, including memory-mapped and bit-blasted
    - An object with unpacked dimension
    - An unpacked struct/union
  - For dynamic objects, assignments are not supported in the DUT

Sampling Non-Driven Signals

For non-driven signals, behavioral compiler determines the signals that should be treated as nonblocking-assignment-driven and must be replaced explicitly with sampled copies based mainly on their signal types and Behavioral Delayed Process usages in the module. By default, behavioral compiler only replaces use of the supported non-driven read-only port signals with sampled copies. You can use the `beh_var` and `rtl_var` IXCOM directives to override the default sampling behavior for the supported non-driven Behavioral Delayed Process signals.

© 2021 All Rights Reserved.

The `beh_var` directive is used to prevent sampling of non-driven Behavioral Delayed Process signals. For example, if you add the `beh_var` directive in the example module for the non-driven port signal `n`, the signal `n` would no longer be sampled and `BDP #2` would use the signal `n` directly.

The `rtl_var` directive is used to force sampled value to be used for non-driven Behavioral Delayed Process signals. For example, if you add the `rtl_var` directive in the example module for the non-driven local signal `m`, the use of m would be replaced by a sampled copy of the signal `m` in `BDP#3`.

Comparison With Previous Releases

In previous VXE and UXE releases, behavioral compiler only transformed behavioral processes and used only the explicit-replacement approach to preserve the pre-nonblocking-assignment values in behavioral processes. Behavioral compiler would determine the signals used in behavioral processes that would require sampled values and replace those signals with explicitly sampled copies inside the behavioral processes. When a signal driven by a nonblocking assignment is used directly in a behavioral process, it is feasible and relatively straightforward to replace the signal in the behavioral process. However, if the signal used in the behavioral process is driven by combinational logic, its value actually depends on all of its fan-in signals, which may or may not be driven by nonblocking assignments. Thus, behavioral compiler needs to determine if sampled value is more likely to generate the right behavior. By directly delaying the nonblocking-assignment signal update for locally driven signals, behavioral compiler provides more consistent and accurate referenced values, by default, for Behavioral Delayed Processes.

Support for External Module References (XMR) in Behavioral Compilation

The behavioral compiler supports XMR disambiguation with the following datatype restrictions.

- XMR Disambiguation

  IXCOM treats each XMR as an individual object, especially when it is used in different named scopes. For better XMR support, behavioral compiler attempts to identify the XMRs that are equivalent and treats them as the same signal. In this release, behavioral compiler adopts a simple conservative approach and considers two XMRs as equivalent only if these have identical hierarchical paths and refer to the same external object for all module instances. This XMR disambiguation facilitates the identification of delayed events and Behavioral Delayed Process signals, and honoring of the directives for all intended XMRs.

- Datatype Restrictions for XMRs

© 2021 All Rights Reserved.

Delayed nonblocking assignment transformation and explicit sampling is restricted for XMRs to signals of packed data type without named components, such as enumerated type, struct, and union. Delayed nonblocking assignment transformation and explicit sampling requires a local signal to be created with the same datatype as the XMR signal. Creating a local datatype that mirrors the external datatype with named component can result in name conflict.

Using $IXC_CTRL Directives to Guide Behavioral Compilation

Use compiler directives to guide the behavioral compilation when the default behavior is not desirable or optimal for a design.

Specify the compiler directives by using the following syntax:

```
$ixc_ctrl("directive_name"[, <directive_object>]*);
```

The compiler directives are used for:

- Specifying Properties of Signals
- Exporting Named Events from DUT to Testbench
- Specifying Loop Unrolling
- Specifying Loop Stepping
- Specifying Behavioral Clock Event

Specifying Properties of Signals

The following directives are used for specifying properties of signals to override default behavioral compilation of the behavioral module:

- `$ixc_ctrl(“beh_var”, <signal>);`

  Use the `beh_var` directive in an `initial` statement of a behavioral module to specify the signal to be driven outside the module by blocking assignment in an `always` procedure executed during BEVAL. This is to prevent sampled value to be substituted for the reads of the specified signal in the module. This is the default behavior for reading non-driven local signals and XMR Behavioral Delayed Process signals.

  This directive only applies to the signals that are not locally driven. It has no effect on locally driven signals and non-Behavioral Delayed Process signals. This directive applies to port, local, and XMR signals with packed datatype.

© 2021 All Rights Reserved.

An `always` procedure that is sensitive to the `beh_var` signal that you have specified is a Behavioral Delayed Process because its execution might be delayed due to BEVAL.

- `$ixc_ctrl("rtl_var", <signal>);`

  Use the `rtl_var` directive in an `initial` statement of a behavioral module to specify the signal to be driven outside the module by non-blocking assignment in an `always` procedure executed during RTL evaluation, i.e. not during BEVAL. This is to force sampled value to be created and substituted for the reads of the specified signal in the module. This is the default behavior for non-driven Behavioral Delayed Process port signals.

  This directive only applies to signals that are not locally driven and are used only as reads (not as event signals). It has no effect on locally driven signals, non-Behavioral-Delayed-Process signals, or event signals. This directive applies to port and local signals with packed datatype. It also applies to XMR signals with packed datatype and without named components.

- `$ixc_ctrl("rtl_clk", <signal>);`

  Use the `rtl_clk` directive in an `initial` statement of a behavioral module to specify the variables that should be considered as RTL clock. This would enable certain performance optimization during behavioral compilation.

- `$ixc_ctrl(“delayed_event”, <signal>);`

  Use the `delayed_event` directive in an `initial` statement of a behavioral module to force all of the events on the specified signal in the module to be *delayed events*. This means that when an `always` procedure is triggered by this signal, its execution would be delayed so that the current non-sampled values would be used in emulation. This directive applies to all signal types: port, local, and XMR signals. For `always` procedures, this is the default behavior for asynchronous events.

  An `always` procedure that is sensitive to an edge event of a delayed event signal that you have specified, even without non-RTL or behavioral constructs, would be transformed as a behavioral process in order to honor the `delayed_event` directive.

- `$ixc_ctrl(“immediate_event”, <signal>);`

  Use the `immediate_event` directive in an `initial` statement of a behavioral module to force all events on the specified signal in the module to be *immediate events*. This means that when an `always` procedure is triggered by this signal, its execution would not be forced to be delayed. This directive applies to all signal types: port, local, and XMR signals. For `always` procedures, this is the default behavior for synchronous-edge events, such as `@(posedge clk)` where `clk` is not read in the `always` procedure.

© 2021 All Rights Reserved.

Important

In each of the above directives, specify at least one or more variables.

Exporting Named Events from DUT to Testbench

The following directives are used for specifying testbench references of named events declared in the DUT when the testbench is not compiled together with the DUT in IXCOM.

- `$ixc_ctrl("export_named_event_sens", <named_event>);`

  Use the `"export_named_event_sens"` directive in an `initial` statement in the behavioral module where the named event is declared to export the event for testbench wait.

- `$ixc_ctrl("export_named_event_trigger", <named_event>);`

  Use the `"export_named_event_trigger"` directive in an `initial` statement in the behavioral module where the named event is declared to export the event for testbench trigger.

Specifying Loop Unrolling

The `"bc_unroll_loop"` directive is used for specifying that all loops or a subset of the loops in a behavioral module should be unrolled prior to any behavioral transformation. With this directive, loops that have static constant loop bound will be forced to be unrolled, even if the loop body includes time-consuming statements, such as event control (`@`). Named scopes declared inside unrolled loops would be renamed, when necessary. This is to avoid name conflicts between duplicate copies of the named block, one for each unrolled loop iteration.

To specify this directive for a particular behavioral module, include the following line in an `initial` statement in the module:

```
$ixc_ctrl("bc_unroll_loop");
```

For example:

```
module dut;
...
initial $ixc_ctrl("bc_unroll_loop");  // unroll all loops with static constant
                                      // loop bounds in this behavioral module
always begin
   ...
   for (i = 0; i < 10; i = i+1) begin  // loop will be unrolled regardless of
                                       // time-consuming @event
     begin:xx  // named block will be renamed, so duplicate copies of the named
               // block will have unique name
```


© 2021 All Rights Reserved.

```
        @(count) ...;
      end
   end
   for (j = 0; j < 10; j = j + 1) begin // loop will be unrolled
      ...
   end
end
endmodule
```

To specify this directive only for a particular loop in the module, add the following line right before the loop:

```
//ixc bc_unroll_loop
```

For example:

```
module dut;
   ...
   always begin
      ...
      //ixc bc_unroll_loop
      // preceded by bc_unroll_loop directive, force loop unrolling
      for (i = 0; i < 10; i = i+1) begin
          // named block will be renamed, so duplicate copies of the named block
             // will have unique names
          begin:xx
                @(count) ...;
          end
      end
      // not preceded by bc_unroll_loop directive, loop unrolling not forced
      for (j = 0; j < 10; j = j + 1) begin
           ...
      end
   end
endmodule
```

Specifying Loop Stepping

The `"bc_step_loop"` directive specifies that the loop following the directive should be executed one iteration per behavioral evaluation. This directive is used to prevent loop unrolling that would cause resource explosion.

To specify this directive for a particular loop in the module, add the following line right before the loop:

```
//ixc bc_step_loop
```

For example:

```
module dut;
   ...
   always begin
```


© 2021 All Rights Reserved.

```
        ...
        //ixc bc_step_loop
        // preceded by bc_step_loop directive, force loop stepping
        for (i = 0; i < 100; i = i+1) begin
           // each behavioral evaluation would execute only one iteration
           // of the loop
           // resource is shared between different iterations
           ...
        end
        // not preceded by bc_step_loop directive, loop unrolling is allowed
        for (j = 0; j < 10; j = j + 1) begin
           ...
        end
     end
  endmodule
```

Specifying Behavioral Clock Event

The "bclk" directive is used for specifying explicit behavioral clock event. Operations separated by the "bclk" directive in a behavioral process would be executed in separate behavioral evaluation. The directive can be placed in any location where a statement is legal.

For example:

```
for (i = 0; i < X; i = i+1) begin
      a = a + b[i];
      $display("a = ", a);
      // ixc bclk
end
```

In the above example, a behavioral clock event would be added at the end of each loop iteration where the "bclk" directive is placed.

The "bclk" directive is useful for explicitly breaking up long chains of sequential operations. It is also useful for guarding GFIFO functions inside a loop. By default, GFIFO display transform ignores any $display statements inside a loop. It can be enabled with the +enableDispInLoopLU option of the ixcom command provided one of the following conditions is true.

- The loop can be unrolled.
- The loop is guarded by an event block, which means that the loop iterations are separated by at least one user- or behavioral-clock event.

In the example, the loop bound X is not a constant. Thus, the loop cannot be unrolled and the added "bclk" directive is necessary for preserving the display inside the loop. Using the +enableDispInLoopBC option for GFIFO display enables GFIFO-display transformation inside the loop by automatically inserting the //ixc bclk statement before the $display statement.

© 2021 All Rights Reserved.

Transforming `initial` Blocks to `always` Processes

Initial blocks in DUT are executed in the simulator and, by default, are required to finish prior to the hardware-acceleration run in the emulator. If required, the user-enabled `initial` blocks in the DUT can be transformed to `always` processes to be executed in emulator after swap-in. For each DUT `initial` block enabled for this transformation, IXCOM performs qualification checks (See Qualification Checks for Transformation of the `initial` Blocks to `always` Processes) to ensure that the `initial` block after conversion to an `always` process, will neither become non-synthesizable nor impose a large overhead. Only qualified `initial` blocks are transformed.

The `initial` blocks are ignored during transformation if these are:

- Within a translate-off region
- Used only to specify directives

Enabling the Transformation of the `initial` Blocks to `always` Processes

To enable this transformation for **all the initial blocks in a module in the DUT**, use either of the following methods:

- Specify the module with the `+mapInitClk+<mod_name>` option of the `ixcom` command. `<mod_name>` refers to the name of the DUT module that should undergo the transformation. For example:

  ```
  ixcom test.sv +dut+dut +mapInitClk+dut ...
  ```

- Add the following `mapInitClk ixc_ctrl` directive in an `initial` block in the DUT module that should undergo the transformation.

  ```
  $ixc_ctrl("mapInitClk");
  ```

IXCOM compiler generates a warning message when an `initial` block enabled by the `+mapInitClk` option or the `mapInitClk ixc_ctrl` directive fails the qualification checks and would not be transformed.

To enable this transformation for **specific initial blocks in the DUT**, add the `mapInitClk ixc` directive inside the initial block that should undergo the transformation as follows:

```
initial begin
    // ixc mapInitClk
...
end
```

IXCOM compiler generates an error message when an `initial` block enabled by the explicit `ixc mapInitClk` directive fails the qualification checks and would not be transformed.

© 2021 All Rights Reserved.

The qualified `initial` block is converted to an `always` process waiting for the posedge of `InitClk` as follows:

Pre-Transformation `initial` Block:

```
initial begin
    state = 0;
    if (...) ...;
end
```

Post-Transformation `always` Process

```
always @(posedge InitClk) begin
    state = 0;
    if (...) ...;
end
```

`InitClk` is a hardware-only clock signal. By default, it triggers once at the first RTL clock after swap-in. If the execution of the transformed `initial` blocks needs to be started before the first RTL clock, use the `-init_clk` option of the `xc on` command to force `InitClk` to trigger before the first RTL clock after swap-in.

You can manually transform initial statement to use `initClock` if the code is beyond automatic-mapping scope. Instantiate the `IXCinitClock(clk)` module defined in the `${IXC_HOME}/etc/IXCclkgen.sv` to source the initial clock for manual conversion:

```
IXCinitClock ic (InitClk);
// initial
  always @(posedge InitClk)
  < initialization code >
```

Qualification Checks for Transformation of the `initial` Blocks to `always` Processes

An `initial` block is generally not qualified to be transformed to an always block if it contains any of the following:

- Declaration or reference to a non-synthesizable datatype
- Declaration or reference to signals of non-synthesizable datatype
- Reference to a memory
- Reference to an unpacked signal
- Non-synthesizable construct

  Exception: Behavioral compilation is enabled, and the construct is supported for behavioral compilation.
- Subroutine call via a hierarchical path.
- Subroutine call to a task or function via a hierarchical path

© 2021 All Rights Reserved.

- Subroutine call to a task or function that is not in a local module scope or imported package
- System task or function call.

  Exceptions: `$finish`, `$stop`, and `$display` family supported by `gfifoDisp`.

© 2021 All Rights Reserved.

Compiling Behavioral Interface Constructs

IXCOM maps the behavioral interface constructs in Verilog and SystemVerilog into the emulator and automatically generates connections for interface constructs crossing the testbench and hardware partitions. These interface constructs include Verilog and SystemVerilog constructs for:

- Connecting objects between different module instances
- Connecting objects with external (foreign language) components

The standard RTL interface constructs, such as module ports and Cross-Module Reference (XMR) wires are supported directly.

Interface compilation focuses on the behavioral interface constructs like:

- Verilog XMR function and task calls
- C and SystemC calls to exported DPI tasks and functions
- SystemVerilog calls to C and SystemC imported tasks and functions
- `#delay` scheduling and `$time` support in emulator (implicit interface to scheduler)

  The above list includes both explicit and implicit interface constructs. All task and function calls listed above, except the `#delay` and `$time` are explicit behavioral calls. For example, `#delay` is an implicit interface call to the simulation time scheduler that needs to consider delay events from all the module instances.

Enabling Interface Compilation in IXCOM

Interface compilation is enabled automatically, by default, if the `-ua` option is specified with the `ixcom` command or if the design includes behavioral modules, driven by the explicit interface constructs and compiler directives specified with the `ixcom` command.

However, to enable support for `#delay` scheduling and `$time` in the emulator, use one of the following options:

- Specify the `+iscDelay` option with the `ixcom` command
- Add the following `map_delays` directive in an `initial` statement in the module containing the delays:

  ```
  $ixc_ctrl("map_delays");
  ```

The implicit interface call to the global scheduler must be explicitly controlled by using the `+iscDelay` option because the fine-grain delay control (`#1`) can have significant impact on

© 2021 All Rights Reserved.

the emulator performance. In most cases, the fine-grain delay control is used to avoid race conditions between the testbench and DUT in the sequential software simulation engine. The parallel emulation engine, on the other hand, is designed to be race free and the fine-grain delay control is not necessary and should be ignored.

With interface compilation, all the DUT `#delay` events are scheduled within the hardware in `tbrun` and `autorun` modes. Synchronization with the host simulator only happens when a host simulator delay event expires or a host interface call event occurs in the DUT.

Rules for #delay Support during Interface Compilation

The `#delay` statements are mapped to hardware according to the following rules:

- Only the blocking `#delay` statements in the `+iscDelay` modules are mapped; all `#delay` values in the default RTL modules are ignored.
- Only blocking intra-assignment delays `#delay` control are considered; all non-blocking intra-assignment delays are ignored, even in the modules specified with the `+iscDelay` option.
- Blocking delays in `initial` statements that meet the forever-loop clock generation style are supported in the modules specified with the `+iscDelay` option. For more details, refer to Modeling Clocks in SA Mode on page 1107.
- Both constant and variables delay values are supported.
- The `#0` delays, from `#0` statements and `#d` statements with d=0 at run time, are ignored.
- When the `+iscDelay` option is used, the clock generation must be a part of the DUT to avoid errors.

Considerations for Using Interface Compiler for $time and $stime Functions

The `$time` and `$stime` functions are supported through interface compilation in the hardware with the following considerations:

- The `$time` and `$stime` functions in the `always` procedures or tasks/functions must be in the form of procedural assignment statements when the `+iscDelay` option is used.

  ```
  var = $time;
  ```

  The `$time` value is a 64-bit object and it consumes a lot of hardware resources to deliver them from the scheduler. Typical applications of `$time` and `$stime` functions in the hardware partition include adding timestamp in transaction data recording and time logging and does not need a 64-bit time stamp. To reduce resource requirements and to

© 2021 All Rights Reserved.

achieve better performance, the use of the assignments guide the compiler to reduce the bit needed to present the time in the application.

- The `$time`, `longint'($time)`, and `$stime` functions are supported as-is when specified as arguments to GFIFO function calls.
- The `$time` function used in `$display` and `$fdisplay` statements are supported as-is.
- All transformed `$time` values are scaled to the minimum time unit of the design unless used in `$display` or `$fdisplay` statements, or as GFIFO function call arguments

Interface Task Mapping Control

Tasks and functions defined in DUT are mapped to execute in hardware unless these are explicitly declared to be interface tasks or functions. If the IXCOM compiler detects that a DUT task or function is not declared as an interface task, but is called from the testbench partition, it generates an error message.

However, if the testbench is not compiled using IXCOM, such as a Specman, UVM, or C testbench, such a condition cannot be detected during IXCOM compilation, and the behavior of the interface task or function might not agree with the expected simulation behavior unless you specify proper mapping control.

For example, consider the following DUT interface:

```
interface uvm_mon_if(input clk, C);
  bit S;
    always@(posedge clk) if( C ) notify_uvm();
    function void notify_uvm(); S = ~S; endfunction
endinterface
```

Here, the `uvm_mon_if` interface can be assigned to a virtual interface of a UVM monitor that is waiting on the signal `S` to toggle when condition `C` is asserted.

By default, the `notify_uvm` function is mapped to hardware, and the signal `S` driven by the function is also mapped to hardware. A call to such a function does not propagate the event to the UVM monitor in software, and causes functional failure.

You must explicitly map the `notify_uvm` function to the software by using the `tb_import` mapping control so that the signal `S` is mapped to software as well. A call from the hardware notifies the UVM monitor to get the expected behavior.

© 2021 All Rights Reserved.

Specifying Interface Mapping Control Using Directives in Source Code

Interface tasks and functions are declared (and defined) in the DUT partition for connection to testbench. You can control the mapping of an interface task or function by using interface compiler directives:

```
$ixc_ctrl("mapping-ctrl", "<task | function_name>" [, "options"]*);
```

You can use the following mapping controls:

- `tb_import`: for functions and tasks called from DUT to run in software.
- `tb_export`: for functions and tasks called from the testbench to run in hardware.
- `gfifo`: for non-blocking functions called from DUT to run in software in parallel with hardware.
- `sfifo`: for non-blocking functions called from the testbench to run in hardware in parallel with the testbench.
- `gsf_is`: for streaming non-blocking functions called from DUT to run in software in parallel with hardware.
- `tb_task`: for tasks called from the testbench to run in software.
- `tb_init`: for functions and tasks called in the initial phase.

The control options enabled by the mapping controls are defined in the following sections along with the detailed behavior of each mapping control.

Specifying Interface Mapping Control Using a Side File

You can use the `+tfconfig+<side_file_name>` option of the `ixcom` command to specify the interface-compilation mapping using a side file, in which each line specifies a mapping control in the following format:

```
<module_name> <task_func_name> mapping_ctrl [mapping_options]*
```

The `<module_name>` can be a specific SystemVerilog module, interface, package name or the `*` character, which means for all modules with the given task and function name.

The `<task_func_name>` can either be specific task or function name or have a prefix`*` to match all tasks and functions with the given prefix. However, you cannot have both wildcard `module_name(*)` and wildcard `task_func_name` (`prefix*`) at the same time.

Note: The `mapping_ctrl` and `mapping_options` are identical to the ones used with the `$ixc_ctrl` mapping directive.

© 2021 All Rights Reserved.

Determining Hardware and Software Objects

IXCOM compiler uses the following rules for mapping DUT objects to hardware or software:

- If an object is referenced by an `always` statement, it is considered a hardware object.
- If an object is referenced by a hardware task or function, it is considered a hardware object.
- If an object is only referenced by a software task or function, it is considered a software object.
- If an object is in the translate-off region, it is considered a software object.

Note that a software task or function (except a GFIFO function) can also reference hardware objects out of its local scope with a cost. IXCOM translates these hardware references to MARG operations.

IXCOM can only optimize the data transfer through function- or task-arguments; but not through out-of-scope references. To achieve good performance, avoid out-of-scope reference from software to hardware.

No Interface Task or Function calls from Combo Logic

All interface tasks and functions (except `tb_init` type) require instrumentation on the caller process and the caller process must use explicit sensitivity to guard its execution. So, calling interface tasks or functions from the following `always` blocks is not allowed:

- ❑ `always@(*)`
- ❑ `always_comb`
- ❑ `always_latch`

Adding instrumentation to implicit-sensitivity `always` blocks might cause Xcelium simulator to hang at time 0. IXCOM generates an error when it detects `always` blocks containing any constructs that require instrumentation.

For best performance and predictable results, limit the interface task and function calls from clocked `always` blocks as much as possible.

Argument values transferred between the caller and the callee are always 2-state even when the corresponding arguments have 4-state data types. As a result, `'bx` and `'bz` values cannot be transferred through interface task or function call arguments. This limitation applies to both hardware and software runs.

© 2021 All Rights Reserved.

Using tb_import Tasks and Functions

The `tb_import` tasks and functions are those that are called from the DUT to run in software.

`tb_import` is the default mapping for DPI import functions and tasks, and does not require explicit mapping control with default mapping options.

The `tb_import` SystemVerilog functions must be local to the module scope and require explicit mapping control; `tb_import` mapping control on XMR tasks and functions is not allowed.

IXCOM supports DPI tasks and functions in blocking and non-blocking execution semantics. An import DPI task (or function) can be marked as `tb_import`. Static and synthesizable unpacked structures and unpacked arrays are supported in `tb_import` tasks and functions.

In import DPI tasks (or functions), unpacked datatypes are not allowed in parameters. If, however, an import DPI task (or function) is marked as `tb_import`, memory parameters are allowed.

Compiler mapping of `tb_import` tasks and functions:

- Blocking: a call to `tb_import` task or function suspends the caller thread unless `tb_tbcall` option is used.
- Synchronous: hardware execution is suspended and control is transferred to software.
- A SystemVerilog `tb_import` task or function runs in software even if it is defined in DUT partition.

Optionally, you can use:

- `tb_pio` or `tb_mem`: use PIO or memory for passing arguments. By default, IXCOM uses `tb_mem`. Use the `+piodpi[+<mod>]*` option of the `ixcom` command to specify that `tb_pio` should be used globally or on a per-module basis.
- `tb_tbcall`: non-blocking execution without waiting for return.

Restrictions:

- A `tb_import` SystemVerilog task or function can reference DUT memory objects outside its local scope with cost.
- A `tb_import` SystemVerilog task or function should not reference DUT scalar (bit-vector) signals outside its local scope. DUT scalar objects should be transferred through task or function arguments.

© 2021 All Rights Reserved.

- A `tb_import` SystemVerilog task or function should only be used either as a wrapper to call functions and tasks in the testbench partition, or store transaction data in a local buffer like a SystemVerilog mailbox for the testbench to pick up through a virtual interface.
- Unpacked unions are not supported in parameters.

Using tb_export Tasks

`tb_export` tasks are those called from the testbench to run in hardware.

The `tb_export` task is the default mapping for export DPI tasks, and does not require explicit mapping control with default mapping options.

The `tb_export` SystemVerilog tasks must be local to the module scope and require explicit mapping control; `tb_export` mapping control on XMR tasks is not allowed.

Static and synthesizable unpacked structures and unpacked arrays are supported in `tb_export` tasks.

Compiler mapping of `tb_export` tasks:

- Blocking: a call to `tb_export` task halts the caller thread. The task is scheduled to execute in hardware after all the testbench threads are suspended at the current simulation time.
- Synchronous: The testbench can only call `tb_export` tasks when the hardware is suspended by host synchronization events.
- A `tb_export` task runs in hardware and can reference hardware objects out of its local task scope.

Optionally, you can use `tb_pio` or `tb_mem` and use PIO or memory for passing arguments. By default, IXCOM compiler uses `tb_mem`.

The `tb_marg` option allows any two-dimensional packed array arguments to be mapped to a memory.

Restrictions:

- A `tb_export` task body must meet the requirements listed for the IXCOM-extended RTL and behavioral subsets for mapping to run in hardware. Refer to Supported Behavioral Constructs for these requirements.
- The maximum total bit width of input and inout arguments of a `tb_export` task is 64K.

© 2021 All Rights Reserved.

- The maximum total bit width of output and inout arguments of a `tb_export` task is 64K.
- With the `tb_marg` option, the two-dimensional array arguments are excluded when calculating the total bit width of arguments.
- Unpacked unions are not supported in parameters.

Using tb_export Functions

`tb_export` functions are called from the testbench to run in hardware.

The `tb_export` function is the default mapping for exporting DPI functions and it does not require explicit mapping control.

The `tb_export` SystemVerilog functions must be local to the module scope and require explicit mapping control; `tb_export` mapping control on XMR functions is not allowed.

Compiler mapping of `tb_export` function:

- Blocking: a call to the `tb_export` function executes the function in hardware immediately. The caller waits for the round-trip latency of the SA channel.
- Synchronous and asynchronous: a `tb_export` function call can be initiated in any hardware execution state and does not suspend the hardware execution if the hardware is active.
- A `tb_export` function runs in hardware and can reference hardware objects out of its local function scope.

Restrictions:

- The body of a `tb_export` function must be strict RTL; it cannot have `while` loops or any behavioral constructs.
- You cannot define `tb_export` functions in compilation scope or packages.
- Arguments of `tb_export` SystemVerilog functions can only have packed data types.
- Arguments of `tb_export` DPI functions can have packed data types and static and synthesizable unpacked structs and unpacked array. Xcelium tool support for DPI tasks or functions with unpacked structure or array argument is incomplete. Thus, IXCOM assumes there is no testbench SystemVerilog caller to `tb_export` DPI functions with unpacked arguments. Specify the `+sfifoKeepSvUnpackedCaller option` with the `ixcom` command to override the default assumption. However, the Xcelium tool might generate an error in such situations.

© 2021 All Rights Reserved.

- Trigger of local-named event trigger is not allowed in `tb_export` functions. For `tb_export` functions in behavioral module, this restriction can be relaxed by using the `+allowEventTriggerInExportFunc` option of the `ixcom` command.

Using GFIFO Functions to Call Functions from DUT to Run on Software Side

GFIFO functions are non-blocking functions called from DUT to run in software in parallel with hardware.

GFIFO functions require explicit mapping control and can apply to both import DPI-C and SystemVerilog void functions that are declared in local module scope. For SystemVerilog functions, input argument types of scalar or packed bit-vector are supported. For DPI-C functions, input argument types of scalar, packed bit-vector, or static and synthesizable unpacked structures and arrays are supported.

Characteristics of GFIFO functions:

- Non-blocking: A call to a GFIFO function does not suspend the caller process, or the hardware execution. The call becomes a packet stored in a local buffer of each call instance. Then, it is collected into the GFIFO.
- Asynchronous: Call packets are transferred asynchronously from the GFIFO to the run-time tools by continuous polling.

  Note: High memory-transfer counts in profile are normal and expected when GFIFO is active.
- Parallel: At run time, the GFIFO packets are executed in parallel while the DUT clocks advance in time.
- GFIFO functions can preserve only the execution order of the call events. It cannot control the exact execution (simulation) time of a GFIFO function.
- A GFIFO function is a circular buffer and can run continuously at ICE speed if the GFIFO execution speed in software runs faster than the call-packet-generation speed in hardware.
- When a host-synchronization event, such as a `tb_import` call or testbench `#delay` event occurs, at run time, the data is flushed and all buffered GFIFO packets are executed before the synchronization event is executed.

Options of the GFIFO Function

- `tb_time`: Specifies the time stamp of the call events.

© 2021 All Rights Reserved.

- Use `marg_gfifo_time()` defined in `marg_user.h` to get the DUT time of the call event if `tb_time` option is used.
- Use `marg_time_precision()` defined in `marg_user.h` to get the time unit of `marg_gfifo_time()`.

Note: If the `tb_time` attribute is not specified for a GFIFO function, the time value returned by `marg_gfifo_time()` is the latest hardware time. This value is always greater than or equal to the hardware time at which the GFIFO is called. It is not deterministic and can vary from run to run even if the GFIFO is called at the same hardware time in every run.

- `varargs=bitw` : It is a variable-size argument transfer for formal arguments `width >= bitw`. Here, `bitw` must be `>= 64`.

  You can declare a formal argument of an import DPI function to a large width so that it can fit all possible actual arguments. However, the actual arguments can be much smaller than the declared formal width. For example, you can use the following DPI function to send different types of messages up to 4K bit:

```
import DPI-C void function send_msg (bit[3:0] m_type, bit[4096-1:0] message);
bit [7:0] a_short_msg;
always@(posedge BFM_clk)
   if(ready) send_msg(`SHORT, a_short_msg);
```

  In this example, the actual argument (`a_short_msg`) has only eight bits of data. With this option, only the actual eight bits (instead of 4K bits) of data is packed into GFIFO to speed up data transfer.

Use the `+gfifo_lbsize+<local_buffer_size>` option of the `ixcom` command for global configuration. For example, `+gfifo_lbsize+1` sets the buffer size to `1`.

For each GFIFO call instance, a local buffer of size is created to buffer call packets before collection into the GFIFO. GFIFO can only collect one call packet per FCLK. When multiple call packets are generated at the same time, local buffer can hold the call packets and keep the RTL clock running. When local buffers are full, behavioral cycles will kick in to suspend RTL clocks to prevent new packet generation. The default local buffer size is `4` and you can adjust it at compile time using the `+gfifo_lbsize` option. Using a deeper local buffer will cost more emulator gates, but will tolerate traffic congestion for better performance. Choose the local buffer size based on the GFIFO traffic pattern to trade-off between performance and gate count. If the gate count is limited, or the design has large number of GFIFO call instances, set the local buffer size to `1`. If the gate-count utilization is low, but traffic congestion causes high behavioral cycles, set the local buffer size to 8.

Xcelium tool support for DPI function with argument type of unpacked structure or unpacked array is incomplete. Thus, IXCOM assumes there is no testbench SystemVerilog caller to GFIFO functions with unpacked arguments and removes the import DPI declaration from the

© 2021 All Rights Reserved.

generated code. To avoid this removal, specify the `+gfifoKeepUnpackedDecl` option with the `ixcom` command. However, the Xcelium tool might generate an error in such situations.

Following are the limitations of GFIFO:

- A GFIFO function runs in software asynchronously with the hardware, and cannot have any reference to the hardware objects out of its function local scope.
- A GFIFO function can only call `tb_export` functions (no task allowed) or SFIFO functions to send transactions back to hardware asynchronously.
- If GFIFO SystemVerilog functions are only used for DUT event monitoring and logging, these do not trigger testbench events to generate input traffic. In such cases, the synchronous `tbrun` mode of execution can work well because testbench agents do not generate input traffic until a host synchronization event.
- If a GFIFO SystemVerilog function is used to notify a testbench agent to generate input traffic, it will trigger testbench-notification events. In the synchronous `tbrun` mode of execution, these notification events will be blocked until a host synchronization event. As a result, input traffic is not generated and might cause the system to hang. To avoid this problem:
  - Use the asynchronous `nbrun` mode of execution to execute testbench events as these are generated and to keep the input traffic flowing from testbench to DUT.
  - Limit GFIFO SystemVerilog functions to store transaction data in a software buffer (such as a SystemVerilog mailbox or queue) and use explicit host synchronization events from DUT to notify SVTB consumers to process transaction data from the software buffer.
  - Use `xc` run-time commands as follows:

    ```
    xmsim> xc ecmset maxSyncCycle <n>
    ```

    The above command sets the maximum FCLKs between `tbsync` events to be <*n*>. At run time, if no `tbsync` events are detected to occur after <*n*> FCLKs, a `tbsync` event is generated so that testbench events can be processed.

    ```
    xmsim> xc ecmset maxGfifoSyncCycle <num>
    ```

    The above command sets the maximum FCLKs between GFIFO events and `tbsync` events to be <*num*>. At run time, if no `tbsync` events are detected to occur <*n*> FCLKs after the last GFIFO call, a `tbsync` event is generated so that testbench events can be processed.
- GFIFO functions cannot be defined in or called from legacy Verilog modules.
- GFIFO functions with escaped name are not supported.
- The maximum total bit width of arguments of a GFIFO function is 32K.

© 2021 All Rights Reserved.

- The maximum total bit width of arguments from all GFIFO function call instances is 20M/ <*local_buffer_size*>.
- The maximum number of GFIFO call instances is 1,000,000.
- GFIFO function calls using default argument are not supported.
- GFIFO function calls cannot have a `$system` function call as an argument, with the exception of `$time` and `longint'($time)` function calls.

  Time arguments are not supported with the `varargs` optimization specification. Time arguments used for GFIFO function with `varargs` enabled would result in compile-time errors.
- GFIFO function calls using hierarchical reference via interface port are not supported.
- GFIFO function calls inside a loop that is not guarded by event block are not supported.
- GFIFO functions do not support unpacked parameters, with the exception of static and synthesizable unpacked structures and unpacked arrays for GFIFO DPI functions.

Using GFIFO for $display Family of Functions (+gfifoDisp)

One main application of GFIFO is streaming and logging monitor events. You can either define custom import DPI monitor functions or use `$display` family functions for logging DUT events.

Use the `+gfifoDisp[+<module_name>]*` option of the `ixcom` command to automatically map the following functions using GFIFO:

- ❑ `$display`, `$displayh`, `$displayo`, `$displayb`
- ❑ `$fdisplay`, `$fdisplayh`, `$fdisplayo`, `$fdisplayb`
- ❑ `$fwrite`, `$fwriteh`, `$fwriteo`, `$fwriteb`
- ❑ `$write`, `$writeh`, `$writeo`, `$writeb`

This `+gfifoDisp[+<module_name>]*` option maps the `$fopen`, `$fclose`, `$fflush` functions to `tb_import` if GFIFO-based display is enabled for modules containing these functions.

The `+gfifoDisp` option preserves the order of `$display` statements.

Note: The `+bufdisp` option is deprecated. The non-buffered `+iscdisp` option is still supported; but will be deprecated in a future release.

IXCOM compiler issues following warning when `+bufdisp` option is used:

© 2021 All Rights Reserved.

```
Warning! in file test.sv at line 1 in module top [BUFDISP_WARNING]
   +bufdisp option has been deprecated, please use option +gfifoDisp instead.
   +gfifoDisp has better performance and will also preserve the order of $display.
```

Using +staticFd to Reduce Gate Count with +gfifoDisp

In general, the file descriptor used by `$fdisplay*` functions can change dynamically in DUT. In order to support the most general use mode, the integer-file descriptor is stored in hardware and send to software at every call. However, in most cases, the file descriptor is created by `$fopen` in the `initial` statement and stay as a constant to the end of simulation. In this case, the file descriptor can stay in software. Use the `+staticFd[+<module_name>]*` option of the `ixcom` command to keep file descriptors in software, globally (or for the given set of modules) to reduce the hardware-resource requirement and improve I/O performance. If you have a large number of `$fdisplay*` statements, the gate-count saving can be significant. If the design has 10,000 `$fdisplay`, using the `+staticFd` option can save 3.2MG with a minimum local buffer size (`+gfifo_lbsize+1`).

Using GFIFO and SFIFO in TBA Applications

GFIFO and SFIFO enable streaming TBA transactions at ICE speed. However, the ICE speed can only be achieved with parallel testbench deployed together with GFIFO and SFIFO.

The following sections list some basic methods to control GFIFO and SFIFO execution.

Using GFIFO Flush Control

A host-synchronization event (`tbsync` event) flushes GFIFO and transfers execution control to testbench. For example, you can call a `tb_import` task or function (a host synchronization event) to flush GFIFO. However, a `tb_import` call creates a PI-PO hardware resource overhead as well as a run-time host processing overhead.

IXCOM provides a special `tbsync` module to generate `tbsync` events without any PIO and run-time overhead.

```
module IXCtbsync #(WIDTH=1) (input [WIDTH-1 0] trigger);
```

This module is defined in the `${IXCOM_INSTALL_HOME}/share/vxe/etc/ixcom/IXCclkgen.sv`. You can instantiate an `IXCtbsync` module and use the value-change events of the trigger signal to create a `tbsync` event and transfer control to the testbench side and flush GFIFO as a side effect.

For example:

```
bit tbs;
IXCtbsync tbsI(tbs);
```


© 2021 All Rights Reserved.

```
always@(posedge clk) begin
  gfifo_function_call(arg1, arg2, ..);
  tbs = ~tbs;
end
```

Using GFIFO for Synchronous TBA Applications

GFIFO is primarily used for asynchronous parallel TBA applications. It can also be used in synchronous sequential TBA application by adding `tbsync` control to flush GFIFO at every GFIFO function call. GFIFO mapping with flush control can improve performance over the `tb_import(tb_mem)` mapping when the design has many active transactors that can generate transactions at the same simulation time.

With `tb_import(tb_mem)` mapping, a call event creates a separate memory transfer overhead. As a result, the memory transfer overhead is proportional to the total number of call events.

On the other hand, mapping these functions as GFIFO functions with flush control gathers all call events at the same simulation time into a common GFIFO. As a result, the memory transfer overhead is fixed to one and is independent of the number of call events at a given time.

The more call events can happen at the same time, the better the performance gain with GFIFO mapping compared to `tb_import(tb_mem)` mapping. GFIFO mapping never has worse performance than `tb_import(tb_mem)` mapping.

GFIFO mapping can be up to 2X slower compared to `tb_import(tb_pio)` mapping when the number of PO bits is less than 8K. When the number of PO bits go over 8K, the GFIFO mapping has better performance than `tb_import(tb_pio)` mapping as well.

Using Proper Buffering with GFIFO and SFIFO Functions

When using GFIFO for TBA applications, a user-level buffer is needed to collect low-level clock-driven BFM data into a complete transaction data for testbench consumption. For GFIFO, this user buffer should be placed on the testbench side. The DUT side should not do any buffering; but simply call the GFIFO function as the data is sampled by the BFM clock.

When SFIFO is used for TBA applications, a hardware buffer such as a transaction FIFO or memory is needed on the DUT side.

- SFIFO can run 2X faster than the BFM clock. So, you must allocate deep enough transaction FIFOs with proper flow control to work with SFIFO transaction data streaming.

© 2021 All Rights Reserved.

- Simple flow control can be achieved by calling a GFIFO function from the BFM to notify the producer with transaction FIFO states to control transaction data generations.

Using SFIFO Functions to Call Functions from Testbench to Run on Hardware Side

SFIFO functions are non-blocking functions called from testbench to run in hardware in parallel with testbench.

Mapping an export DPI-C function to a SFIFO function requires explicit mapping control. An SFIFO DPI-C function can have input, output, and inout arguments. Output arguments from caller (C pointers) are collected into SFIFO together with values of input arguments. However, values of output arguments will not be valid until SFIFO is flushed. You need to prevent collision of output argument pointers for all buffered SFIFO calls so that return values will not collide to the same location.

For SFIFO DPI functions, argument types of scalar, packed bit-vector, or static and synthesizable unpacked structures and arrays are supported. Xcelium tool support for DPI functions with unpacked structure or array is incomplete. Thus, IXCOM assumes there is no testbench SystemVerilog caller to SFIFO DPI functions with unpacked arguments. Specify the `+sfifoKeepSvUnpackedCaller option` with the `ixcom` command to override the default assumption. However, the Xcelium tool might generate an error in such situations.

Mapping a SystemVerilog function declared in the local module scope to an SFIFO function requires explicit mapping control. The output arguments of the SystemVerilog function cannot be kept as a pointer. As a result, only input arguments are allowed for SFIFO SystemVerilog functions. Input argument types of scalar and packed bit-vector are supported.

Characteristics of SFIFO functions:

- Non-blocking: A call to an SFIFO function does not suspend the caller thread and hardware execution. The call becomes a packet stored in the SFIFO.
- SFIFO flush:
  - A `tb_export` function call (synchronous or asynchronous) flushes SFIFO before returning control to the caller.
  - SFIFO flushing does not suspend hardware execution.
  - You can also use a built-in C-function `SFIFO_flush()` to flush SFIFO.
  - If SFIFO call packets are generated while hardware is active, at run time, the SFIFO flushing is combined with GFIFO polling to minimize the latency from call to execution.

© 2021 All Rights Reserved.

  - If SFIFO call packets are generated while hardware is suspended by host-synchronizations events (such as `tb_import` calls or testbench `#delay` events), at run time, SFIFO is flushed right before control is transferred to hardware.

- Parallel: Execution of SFIFO in hardware is in parallel with both testbench and DUT.

SFIFO can only preserve the execution order of all call events but cannot control the exact execution time of an SFIFO function in hardware.

The main use model of SFIFO tasks is injection of input data into DUT. For this, limit the arguments of SFIFO tasks to inputs; SFIFO tasks with only input arguments do not suspend the calling thread. Output arguments are allowed. But, in such cases, the calling thread is be suspended until the SFIFO task finishes execution.

Restrictions Related to SFIFO Functions:

- An SFIFO task body must be RTL. The body of an SFIFO task can only have `wait` statements on the same edge of a clock signal, such as `@(posedge clk)`.
- SFIFO functions have the same pure RTL restrictions like `tb_export` functions.
- The execution speed of SFIFO is one packet per FCLK in 2X mode and two packets per FCLK in 1X mode. When using SFIFO to stream transactions to a BFM, the execution speed of SFIFO is faster than the processing speed of the BFM. You must use both data buffer and proper flow control to handle high speed streaming without losing transaction data.
- SFIFO functions are not supported in compilation-unit scope or package.
- SFIFO functions are not supported in legacy Verilog module.
- SFIFO functions with escaped name are not supported.
- The maximum total bit width of input and inout arguments of a SFIFO function is 32K.
- The maximum total bit width of output and inout arguments of a SFIFO function is 32K.
- A time-consuming SFIFO task should start with an event control like `@(posedge clk)`. This ensures that the software simulation has a proper clocking mechanism to avoid race conditions.
- SFIFO functions do not support unpacked arguments, with the exception of static and synthesizable unpacked structures and unpacked arrays for SFIFO DPI functions.

© 2021 All Rights Reserved.

Using Non-Blocking SFIFO Task Calls

The IXCOM compiler supports the mapping of DUT functions as SFIFO functions to minimize the overhead of interaction between the testbench and DUT through non-blocking and asynchronous communication. However, many Testbench-DUT interactions are time-consuming. So, it is best to model these as DUT tasks called from testbench.

Consider the following example of DUT interface with a SystemVerilog task:

```
interface uvm_driver_if(input clk);
    int BUSDATA0;
    int BUSDATA1;
    task void drive_data(input int data0, input int data2);
      @(posedge clk);
        BUSDATA0 <= data0;
      @(posedge clk);
        BUSDATA1 <= data1;
    endtask
endinterface
```

This `uvm_driver_if` interface can be assigned to a virtual interface of a UVM driver that injects test data by calling the `drive_data` task.

If you map the `drive_data` task as a `tb_export` task, the body of the task is executed inside the hardware, in a synchronous and blocking manner.

You can mark such interface tasks defined in the DUT partition for connection from the testbench as SFIFO tasks. SFIFO tasks are interface tasks that are called from the testbench, and run on the hardware side, without suspending hardware execution.

Two types of tasks in DUT can be marked as SFIFO tasks:

- Export `DPI-C` task

  By default, an export `DPI-C` task is mapped to a `tb_export` task if you have not specified any explicit mapping control.

  Note: You can map an export `DPI-C` task to an SFIFO task by using an explicit mapping control.

- SystemVerilog task

  Requires explicit mapping control to be mapped to an SFIFO task.

You can specify explicit mapping control to mark `export "DPI-C"` or SystemVerilog tasks in DUT as SFIFO tasks. In the absence of an explicit mapping control, the `export "DPI-C"` tasks are mapped to a `tb_export` task, by default.

© 2021 All Rights Reserved.

Use the `buflen=<number>` option as the `[mapping_option]` to specify the size of the hardware buffer for storing call packets for SFIFO task calls. Here, `<number>` is a positive integer.

By default, different SFIFO tasks are independently scheduled for execution in hardware. Therefore, do not assume an order for execution of different SFIFO tasks. However, if you need to preserve a specific order of execution of different tasks, assign the tasks to a common sequence group by specifying `sgroup=<name>` with the `$ixc_ctrl` task. Here, `<name>` is an alphanumeric string that defines the sequence group. For example, two tasks, `task1` and `task2`, can be assigned to the same sequence group, `G0`, as shown below:

```
$ixc_ctrl("sfifo", "task1", "sgroup=G0");
$ixc_ctrl("sfifo", "task2", "sgroup=G0");
```

In this example, the order of executions of `task1` and `task2` will match the order of their invocations. An SFIFO task can be assigned to only one sequence group. Also, sequence groups can only be specified for SFIFO tasks within the same module.

Characteristics of SFIFO Tasks:

- Non-blocking: A call to an SFIFO task does not suspend hardware execution if the hardware is active. SFIFO task calls can be initiated in any of the hardware-execution states. Calls to the same SFIFO task instances are executed in order in hardware. However, by default, different task instances are independently scheduled for execution in hardware.
- SFIFO can only preserve the execution order of all SFIFO call events but cannot control the exact execution time of an SFIFO task in hardware.
- In TBRUN mode at run time, the completion of an SFIFO task call is not visible to the testbench until the next host-synchronization event, such as a `tb_import` call, a testbench `#delay` event, or a GFIFO timeout event occurs. Without a host-synchronization event, the calling thread hangs forever.
- In NBRUN mode at run time, completion of a SFIFO task call is immediately visible at the next IXCOM scheduling polling event, and does not require a host-synchronization event to wake up its calling thread.
- If the body of an SFIFO task does not contain time-consuming constructs, it is treated as an SFIFO function, and follows the execution semantics of an SFIFO function.

Example of Non-blocking SFIFO Task Calls

In the following example, the input data is driven from testbench into BFM. Task calls are blocking in nature. So, for calling the SFIFO tasks, threads need to be created by using statements like `fork`.

© 2021 All Rights Reserved.

```
module tb;
  dut duttop();
  int done = 0;
  initial begin
    for(int i = 1; i <= 10; i++) begin
        fork
          begin: thread
            duttop.bfm.foo(in[i], out[i]);
            done++;
          end
        join_none
    end
  end
  initial begin
    // wait until all call threads finish
    wait(done == 10);
    $finish;
  end

  // Actual completion of SFIFO task calls in testbench partition, where
  // execution has finished inside hardware, will be only visible in testbench
  // at next host-synchronization event.
  //

  // Below is an example that results in a host-synchronization event
  // every 1000ns.
  always #1000 $display("Sync");
endmodule

module dut;
  reg clk;
  clkgen clkgen_inst(clk);
  BFM bfm_inst(clk);
endmodule

module BFM(input clk);
  int xbus, ybus;
  /* ASYNC task, buffer len, clk */
  task automatic foo(input int x, output int y);
      xbus <= x;
      @(posedge clk);
      @(posedge clk);
      y <= ybus;

  endtask
initial $ixc_ctrl("sfifo", "foo", "buflen=4");
endmodule
```

Using SyncOnEmpty (`soe`) Option for Non-blocking SFIFO Task Calls

In environments where the testbench and DUT execute asynchronously, like in the NBRUN mode, it is important to minimize the idle time spent in waiting. For example, rather than having the testbench send input data to an SFIFO task at random points, it is advantageous to send data exactly at the point when DUT needs more data to operate.

This section describes the SyncOnEmpty or `soe` option in SFIFO tasks. The option triggers a host-synchronization event whenever the DUT has finished execution of all the task calls pending inside the DUT buffer, and the internal buffer that stores the pending SFIFO task calls

© 2021 All Rights Reserved.

becomes empty. This allows synchronization to occur at the moment when DUT is ready for more computation. Upon synchronization, the testbench can send the data, minimizing the idle time of the hardware BFM. Alternatively, you can use the `+sfifotask_soe` option of the `ixcom` command to enable SyncOnEmpty for all SFIFO tasks.

In addition, this option forces a hardware-software synchronization in the TBRUN mode when the buffer becomes empty. When the testbench thread blocks waiting for the SFIFO task to finish in TBRUN mode, simulation can hang unless a host-synchronization event, such as a `tb_import` call or a testbench `#`-delay event wakes up the testbench. Using SyncOnEmpty sends the task results to the testbench when all SFIFO task calls have finished execution, ensuring that the testbench thread wakes up and proceeds with the computation.

Note that there is no limit on the number of calling threads to a SFIFO task. If multiple threads call the same task, these are executed in sequence by queuing the hardware buffer. When the hardware buffer is full, the caller first suspends on full buffer. Then, it again suspends, and waits for return from hardware. In TBRUN mode, this can cause heavy synchronization overhead. To reduce synchronization overhead, set the buffer length to be the maximum number of expected calling threads for a task.

Typically, a task has only a single calling thread. So, do not use the `soe` option unnecessarily because that would waste memory resources.

In the Example of Non-blocking SFIFO Task Calls, to use the `soe` option, modify the following line:

```
initial $ixc_ctrl("sfifo", "foo", "buflen=4");
```

Replace the above line with the following line:

```
initial $ixc_ctrl("sfifo", "foo", "buflen=4", "soe");
```

Support for Dynamic-Array Arguments in tb_export and SFIFO Tasks

Consider the following DUT interface with a SystemVerilog task. To map the SystemVerilog task to hardware, mark it as a `tb_export` or an SFIFO task.

```
interface uvm_driver_if(input clk);
    int BUSDATA[3];
    task drive_data(input int data []);
       for (int i = 0; i < data.size(); i++) begin
            BUSDATA[i] <= data[i];
            @(posedge clk);
       end
    endtask
    initial $ixc_ctrl("sfifo","drive_data");
endinterface
```

Unpacked or dynamic array arguments are not supported in SFIFO tasks. So, the `data` argument, in the `drive_data` task should be remodeled as a packed array argument.

© 2021 All Rights Reserved.

However, such remodeling is error-prone and incorrect modeling can easily cause performance problems.

To use unpacked and/or dynamic array arguments in SFIFO tasks, use the mapping options to direct the interface compiler to automatically insert instrumentation that efficiently transfers such arguments.

Interface Mapping Control for Dynamic-Array Arguments in tb_export and SFIFO tasks

Define the `tb_export` or SFIFO tasks in the DUT partition to connect from the testbench. Use the interface compiler directive to control the mapping of such a task:

```
$ixc_ctrl("tb_export", "task_name" [, mapping_option]*);
$ixc_ctrl("sfifo", "task_name" [, mapping_option]*);
```

You can also use a side file to specify interface compiler mapping with the `+tfconfig+<side_file_name>` option of the `ixcom` command. Inside the side file, each line should specify the mapping control in the following format:

```
<module_name> <task_name> tb_export [mapping_option]*
<module_name> <task_name> sfifo [mapping_option]*
```

Here, `<module_name>` can be a specific SystemVerilog module, interface, package name or the `*` character. The `*` character refers to all of the modules with the given task and function name.

The `<task_name>` can either be a specific task name or prefix `*` to match all tasks with the given prefix. However, you cannot have both wildcard `<module_name>` (*) and wildcard `<task_name>` (prefix `*`) at the same time.

To use unpacked, packed, and/or dynamic-array arguments, provide the information about such arguments in [`mapping_option`].

- `"<arg>.size <= <maxsize>"`: Given a dynamic array argument, `<arg>`, this option specifies the maximum size of the dynamic dimension.
- `"varg <= <default_maxsize>"`: Specify the default maximum size of the dynamic dimension of dynamic arrays. In a dynamic-array argument, for which `"<arg>.size <= <maxsize>"` option is not provided, `<default_maxsize>` is its maximum size.
- `"<arg>.size = <sizearg>"`: Given a bounded, unpacked array argument, `<arg>`, you can specify another argument, `<sizearg>`, which will specify the actual size of the data transferred in each task call.

  For example, in the `putdata` task, the actual size of the data transferred for a task call is given in the `datasz` argument. Let `datasz` be 64 in a particular task call. Then, even

© 2021 All Rights Reserved.

if the data argument has a static bound of `1024`, the interface compiler can only transfer 64 elements of data, reducing the traffic between the hardware and software.:

```
task putdata(int datasz, int data[1024]);
     for (int i = 0; i < datasz; i++) begin
          BUSDATA <= data[i];
          @(posedge clk);
     end
endtask
initial $ixc_ctrl("sfifo", "putdata","data.size = datasz")
```

- To map large two-dimensional packed arrays shown in the example below to memories, use the `tb_marg` mapping option of `tb_export tasks`:

```
task drive_data(bit [4095:0][31:0] data_0, data_1);
     for (int i = 0; i < 4096; i++) begin
          data_0 = data_0[i];
          data_1 = data_1[i];
     end
endtask
```

Specify the mapping option as follows:

```
$ixc_ctrl("tb_export", "task_name" [, mapping_option]*);
```

For example:

```
$ixc_ctrl("tb_export", "drive_data","tb_marg");
```

You can also use a side file to specify interface compiler mapping with the `+tfconfig+<side_file_name>` option of the `ixcom` command. In the side file, each line specifies a mapping control in the following format:

```
<module_name> <task_name> tb_export [<mapping_option>]*
```

Restrictions in Support for Dynamic Array Arguments in tb_export and SFIFO tasks

Following are the restrictions to be kept in mind when using unpacked or dynamic arrays:

- Unpacked or dynamic arrays are allowed only in INPUT arguments.
- In unpacked or dynamic arrays, the packed dimensions must have one of the following datatypes:
  - byte
  - shortint
  - int
  - longint
  - bit or logic vector

© 2021 All Rights Reserved.

- Only one unpacked dimension is allowed in unpacked or dynamic array arguments.
- Given a dynamic array argument in an SFIFO task, the task body can only use the `size()` method on the argument.
- Unpacked datatypes are not supported for arguments in the SFIFO tasks.

Calling GFIFO Functions from SFIFO Functions

GFIFO functions do not support return values; these have only input arguments. So a GFIFO needs to be able to call an SFIFO function to send return values back to hardware. SFIFOs, called from C functions, on the other hand, support return values or output arguments. Therefore, an SFIFO, typically, does not need to call GFIFO to communicate results back to software. Note that an SFIFO function called in SystemVerilog cannot have output arguments. Such SFIFO functions cannot send return values back to software. Therefore, such SFIFO functions that are written in SystemVerilog are allowed to call GFIFOs. This behavior is enabled by default. To disable SFIFOs calling GFIFO functions, use the `+no_scg` option at compile time. Then, a compile-time error will be generated if an SFIFO calls a GFIFO.

Note that only a restricted set of GFIFO calls, such as `$display`, `$fdisplay`, and other monitor-type GFIFO calls can be made from SFIFO functions. This is so because an SFIFO function can only call GFIFO functions that do not further call any SFIFO functions. Such GFIFO functions are called *terminal GFIFO functions*. An SFIFO function that calls the terminal GFIFO function might be called from SystemVerilog code or C code in the testbench. Similarly, the called terminal GFIFO function can be defined as SystemVerilog code or C code in the testbench. The SFIFO function that calls the terminal GFIFO function can itself be called from the testbench or a non-terminal (normal) GFIFO function.

After a call to `SFIFO_flush()` from your C testbench, all outstanding SFIFO functions are executed and their outputs, if any, are made available. The `SFIFO_flush()` also guarantees that any terminal GFIFO functions that are executed by these outstanding SFIFO functions will also be executed. For example, if you print something after returning from `SFIFO_flush()`, all outstanding terminal GFIFO `$display` statements will be printed before your print statement.

Keep the following points in mind while calling GFIFO functions from SFIFOs:

- Calling GFIFOs from SFIFOs is supported only for the `SV_channel` (the default channel); it is not supported for OTB-channels.
- When, a terminal GFIFO function tries to call an SFIFO function, a run-time error is generated.
- The limit is 32K bits of arguments per GFIFO or SFIFO.

© 2021 All Rights Reserved.

Running GFIFO-SFIFO Pairs in Parallel

This section describes the method to improve the performance of GFIFO-SFIFO by declaring multiple pairs of GFIFOs-SFIFOs, and running them in parallel.

A single GFIFO-SFIFO pair does not use up all the I/O bandwidth of an SA channel. This means that a single SA channel can support multiple pairs of GFIFO-SFIFOs operating simultaneously.

The IXCOM run-time software improves the GFIFO-SFIFO performance by creating multiple threads within a single testbench process to access multiple GFIFO-SFIFO pairs in parallel. All the GFIFO-SFIFO pairs are accessed through the solitary SA channel connected to the host that is running the testbench process.

Each GFIFO-SFIFO pair is called a GFIFO-SFIFO ***channel***. A GFIFO-SFIFO channel should not be confused with an SA Channel. An SA channel is the physical connection between the host and Z1. A GFIFO-SFIFO channel is a logical channel that you declare. In this section, the word channel refers to a GFIFO-SFIFO channel, or a GFIFO-SFIFO pair.

One GFIFO-SFIFO channel, which is the default channel named `SV_channel` is always available. Decide the names of each of the non-default GFIFO-SFIFO channels, and create a channel-configuration file that specifies the module instance(s) that will be serviced a particular GFIFO-SFIFO channel. Only a module instance that contains a GFIFO call or an SFIFO definition needs to be serviced by a GFIFO-SFIFO channel. If the mapping for a module instance containing a GFIFO call or an SFIFO definition is not specified, it is serviced by the default channel, that is `SV_channel`. To keep the channel-configuration file concise, you can map an entire instance tree to a particular GFIFO-SFIFO channel.

Important

The mapping ONLY applies to the DPI GFIFO functions (GFIFO functions written in C/C++), and DPI SFIFO functions (SFIFO functions called from C/C++).

SV GFIFO functions (GFIFO functions written in SystemVerilog), and SV SFIFO functions (SFIFO functions called from SystemVerilog) require a SystemVerilog simulator (xmsim) and always run in the default `SV_channel`, regardless of the contents of the mapping file. The `SV_channel` can service DPI GFIFO and DPI SFIFO functions as well.

Specifying the Channel-Configuration File

The format of the channel configuration file is a channel line followed by one or more instance lines. This pattern can be repeated as many times as necessary. Lines beginning with a `#` are treated as comments.

© 2021 All Rights Reserved.

A channel line defines a channel, and has the form:

```
channel <channel_name> [<channel_type>]
```

The optional <*channel_type*> specification can be `LL` or `HB`. `LL` indicates a low-latency channel, and `HB` indicates a high-bandwidth channel. If the channel-type specification is missing, the default channel-type specification is used, which is `HB` for the `SV_channel` and `LL` for the non-default channels. To change the channel-type specification to low latency for all channels, use the `+gsfifo_low_latency` option at compile time. Use the `+gsfifo_high_bandwidth` option to set the default channel type to `HB` for all channels.

Use a low-latency channel when a small round-trip latency is required between software and hardware. You can create a round-trip path by making an SFIFO call to transfer the data from software to hardware. The execution of the SFIFO function changes one or more signals. This triggers some hardware computation. Then, it triggers a GFIFO call that returns the result of the computation from hardware to software. To minimize the latency, call `SFIFO_flush()` after invoking the SFIFO function involved in the low-latency path so that it is flushed immediately.

To achieve higher bandwidth, the high-bandwidth channel needs access to special-communication resources in four Z1 domains. A low-latency channel needs access to special communication resources in only a single Z1 domain. To ensure that all channels can run in parallel, ideally:

```
Total number of domains in a design >= num_low_latency_channels +
4*num_high_bandwidth_channels
```

Low-latency channels require less domains. So, these can also be employed if the use of high-bandwidth channels will prevent all channels from running in parallel.

In previous VXE releases, the GFIFO-SFIFO channel type was always high-bandwidth channel. VXE 18.1 allows you to select between high bandwidth and low latency channel types.

Each channel line can be followed by one or more instance lines, which associate one or more module instances with the preceding channel. An instance line has the following format:

```
    <instance_specification> [GSFIFO|GFIFO|SFIFO][$fopen]
```

An <*instance_specification*> can be a <*hierarchical_instance_name*> or `::`<*module_name*>. If the <*instance_specification*> is `::`<*module_name*>, it represents all of the instances of the named module.

The `GSFIFO` keyword following an <*instance_specification*> indicates that in the instance(s) represented by the <*instance_specification*>, both DPI GFIFO calls and DPI SFIFO definitions will be serviced by the channel specified in the preceding channel line.

© 2021 All Rights Reserved.

The `GFIFO` keyword following an <*instance_specification*> indicates that in the instance(s) represented by the <*instance_specification*>, the DPI GFIFO calls will be serviced by the channel specified in the preceding channel line. The DPI SFIFO definitions in these instance(s) will be serviced by the default `SV_channel`.

The `SFIFO` keyword following a <*instance_specification*> indicates that in the instance(s) represented by the <*instance_specification*>, the DPI SFIFO definitions will be serviced by the channel specified in the preceding channel line. The DPI GFIFO calls in these instance(s) will be serviced by the default `SV_channel`.

The `$fopen` keyword is described in Using $fdisplay in OTB Context section.

To match multiple instances in an <*instance_specification*>, use the `*` wildcard in the <*hierarchical_instance_name*> or `::`<*module_name*>. The `*` wildcard character can only be used at the end of the name. For example, `top.sub.a.*` implies that all instances under `top.sub.a` will be handled by the channel specified in the preceding channel line. On the other hand, the <*hierarchical_instance_name*> `top.sub.a*`, without the `.` between `a` and `*`, would match multiple instance trees like `top.sub.a1` and `top.sub.a2`, and so on, in addition to `top.sub.a` and everything below it.

Following is an example of a channel-configuration file:

```
# This is a comment.
  channel channel_B
    top.* GSFIFO
  channel channel_A
   top.s1 GSFIFO
   top.s2 SFIFO
# Any DPI GFIFO call or DPI SFIFO definition in top.s1 is serviced by channel_A
# Any DPI SFIFO definition in top.s2 is serviced by channel_A, any DPI GFIFO
  call in top.s2 is serviced by the default channel.
# Any other DPI GFIFO call or DPI SFIFO definition under top is serviced by
  channel_B
# End of channel configuration file.
```

If multiple entries in the channel-configuration file match the module instance, the last-matching entry takes effect. You can sequence the entries in this file according to your requirements and place the higher-priority entries later in the file. You can repeat a channel line in the configuration file, provided the channel lines with the same channel name do not have conflicting channel-type specifications.

Consider a scenario in which `top.a.b.c` needs to be assigned to channel A, everything else under `top.a` needs to be assigned to channel B, and everything else under top needs to be assigned back to channel A. The following channel-configuration file with duplicate entries for channel A accomplishes the required mapping.

```
channel A
top.* GSFIFO
channel B
top.a.* GSFIFO
```


© 2021 All Rights Reserved.

```
channel A
top.a.b.c GSFIFO
```

Specify the channel-configuration file to IXCOM by using the `+mc_gsfifo_config+<file_name>` option. If the channel-configuration information is specified in multiple files, you can specify the `+mc_gsfifo_config+` option multiple times.

After compilation, IXCOM prints out the channel-configuration information in `xc_work/mc_gsfifo_config_info.log`. Examine this file to see if the instances were mapped to the channels as expected.

Concepts Related to Running GFIFO-SFIFO Pairs in Parallel

Each GFIFO-SFIFO pair is called a GFIFO-SFIFO channel. Each GFIFO-SFIFO channel comprises separate hardware and software portions. The hardware portion consists of the virtual logic and memories added by IXCOM to your DUT for servicing GFIFO and SFIFO calls. The software portion is called a context.

Channels Required for Running GFIFO-SFIFO Pairs in Parallel

The default GFIFO-SFIFO channel is named SV_channel. This default channel is capable of running both SV and DPI-C GFIFOs and SFIFOs. Previous UXE and VXE releases supported only a single GFIFO-SFIFO channel; In VXE 18.1, this channel is the default channel (SV_channel). On the testbench-software side, this channel is serviced by Xcelium and IXCOM run-time software, collectively known as SV-context. On the DUT-hardware side, the module instances containing GFIFO calls and SFIFO definitions that are mapped to the default channel (SV_channel) are called SV-instances.

The non-default GFIFO-SFIFO channels are called Other Testbench Channels or OTB-channels. OTB-channels can run only DPI-C GFIFOs and SFIFOs. Each OTB-channel has a unique name according to your specifications in the channel-configuration file. OTB-channels have additional restrictions as compared to the `SV_channel`. These restrictions are described in the relevant sections of this chapter, such as the Additional Restrictions Related to Running GFIFO-SFIFO Pairs in Parallel section. On the testbench-software side, each OTB-channel is serviced by an OTB-context which is a part of the IXCOM run-time software. Xcelium is not a part of the OTB-context. On the DUT-hardware side, the module instances containing GFIFO calls and SFIFO definitions that are mapped to OTB-channels are called OTB-instances.

Contexts Used for Running GFIFO-SFIFO Pairs in Parallel

The SV-context and OTB-contexts are automatically created and managed by IXCOM. The SV-context and OTB-contexts run independently and in parallel. The parallelism between the

© 2021 All Rights Reserved.

contexts is achieved according to your specification of the instances that will be serviced by each channel in the channel-configuration file, and the channels run in parallel.

Each OTB-context services the GFIFO and SFIFO functions mapped to its OTB-channel by constantly monitoring the OTB-channel for GFIFO call packets made by its OTB-instances (hardware). The OTB-context executes these GFIFO call packets by invoking the C/C++ GFIFO functions that you have defined. These C/C++ GFIFO functions can make SFIFO calls to send data or transactions back to the hardware, through the OTB-channel, if necessary.

The `SV_context` services the `SV_channel` similar to the way each OTB-context services its OTB-channels. In addition, `SV_context` executes the parts of the testbench that are not serviced by an OTB-channel. This includes all SystemVerilog testbench code, all blocking interface-tasks and functions (`tb_import` and `tb_export`), and all the parts that execute in Xcelium.

The SV-context and OTB-contexts run independently and in parallel within the same process. So, you must ensure that all the GFIFO functions that you write are thread-safe. For high performance, ideally, contexts should not share data. Two contexts should not simultaneously write to the same variable. Nor should one context read a variable while another context is simultaneously writing to it. If this cannot be avoided, then, do not place pthread-locking, busy-waiting, and synchronization calls directly in your GFIFO function because this can interfere with IXCOM's context management and cause a deadlock. Provide thread-safe access to shared variables outside your C/C++ GFIFO functions. One method to accomplish this is to create a separate pthread and a circular-command buffer, have the GFIFO function append a command for the pthread in the command buffer and return. The pthread needs to monitor the command buffer for the arriving commands, and take appropriate action for the arriving commands in a thread-safe manner. The Call Packet Transportation, Execution, and Synchronization for GFIFO-SFIFO Pairs section describes additional features that you can use to coordinate between contexts, further reducing the need for using pthread locking and synchronization primitives.

The ordering of GFIFO-SFIFO calls is not guaranteed across different contexts because of the parallel nature of the SV-context and OTB-contexts. The ordering is guaranteed within each context.

OTB-contexts have no notion of testbench time. A GFIFO function running in an OTB-context can only query the time at which it was invoked in hardware by calling `marg_gfifo_time` or `vpi_get_time` (provided the GFIFO function was declared with the `tb_time` attribute). However, it runs independently of the SV_context, outside the simulation time management infrastructure (time wheel) of Xcelium. So, OTB-contexts always run in `nbrun` mode. The `run`, `tbrun`, and `nbrun` commands apply only to the SV-context. OTB-contexts should simply drive untimed transactions into the hardware and monitor the hardware.

© 2021 All Rights Reserved.

Only the SV-context has direct access (via C handles) to the SystemVerilog objects in the design. The GFIFO functions executed in the SV-context also have access to all the `vpi_`, `svGet`, `svPut`, and `marg_` functions. OTB-contexts cannot directly access SystemVerilog objects, and do not have access to all the `vpi_`, `svGet`, `svPut`, and `marg_` functions. These restrictions are described in the Supported svGet and svPut Functions in OTB-Contexts, Supported vpi_ Functions in OTB-Contexts, and Supported marg_ Functions in OTB-Contexts sections.

Using $fdisplay in OTB Context

You can specify file-output tasks like `$fdisplay`, `$fwrite` etc. to be executed as GFIFO functions by mapping them to GFIFO (`+gfifoDisp` or `$ixc_ctrl("gfifo", $fdisplay")).`

Some changes are required when `$fdisplay` and `$fwrite` calls in a design are mapped to an OTB-channel. Xcelium creates the file descriptors for these tasks in the `SV_context` using `$fopen` and `$fclose` calls in the SystemVerilog code, and these file descriptors can only be used in the `SV_context`. For an OTB-context to service these calls, (while running in parallel with the `SV_context`), the IXCOM run-time software needs to manage the file descriptors by itself.

There are two methods of doing this. In both methods, for better performance, ensure that different GFIFO-SFIFO channels write to different files so that the file-write operations can be done in parallel.

A file descriptor managed by IXCOM should only be used by `$fdisplay` or `$fwrite` calls in OTB-instances. Conversely, a file descriptor managed by Xcelium should only be used by `$fdisplay` or `$fwrite` calls in SV-instances. File descriptors managed by IXCOM have the additional restriction that they cannot be `OR`'d like the `mcd` file descriptors in Verilog, and cannot be used in `vpi_mcd*` routines.

The first method is more flexible and requires you to replace calls to `$fopen` in OTB instances with the DPI-C import function `otb_fopen`, and replace calls to `$fclose` in OTB instances with the DPI-C GFIFO function `otb_fclose`. These functions are declared in `otb_user.h` and allow the IXCOM run-time software (not Xcelium) to manage these file descriptors. Calls to `otb_fopen` must be done prior to swap-in to establish a file descriptor. After swap-in, if the file becomes too big, you can close the original file and open a new one with the same descriptor by calling the DPI-C GFIFO function `otb_freopen` from the OTB context. Unlike `otb_fopen`, `otb_freopen` reuses an existing file descriptor. So, it does not returns a value and can be a GFIFO function. Both `otb_fclose` and `otb_freopen` can be declared as DPI-C GFIFO functions and called directly from the hardware in an OTB instance, or these can be called from an existing GFIFO function in an OTB instance.

© 2021 All Rights Reserved.

The other method to have file-output tasks like `$fdisplay` and `$fwrite` work in OTB channels is for IXCOM to override all calls to `$fopen` and `$fclose` that open or close files written by `$fdisplay` or `$fwrite` calls made from OTB-instances. This allows the IXCOM run-time software (not Xcelium) to manage these file descriptors. In this method also, the calls to `$fopen` to establish a file descriptor must be made before swapping in.

You need to indicate the calls to `$fopen` or `$fclose` need to be overridden by IXCOM. These are calls that open or close files that are written to by `$fdisplay` or `$fwrite` calls made from OTB-instances. The file descriptors returned by the overridden calls to `$fopen` should only be used in `$fdisplay`, `$fwrite`, or `$fclose` calls.

Specify the information to override calls to `$fopen` or `$fclose` in the channel-configuration file by providing the `$fopen` keyword in an instance line.

Here is an example of a channel-configuration file with a `$fopen` keyword:

```
# This is a comment.
  channel channel_A
   top.s1 GSFIFO $fopen
# Any DPI GFIFO call or DPI SFIFO definition in top.s1 is serviced by channel_A
# Any call to $fopen/$fclose in top.s1 will be overridden by IXCOM so that file
  writes can be safely performed in the parallel GFIFO-SFIFO environment.
# End of channel configuration file.
```

If an SV-instance makes calls to `$fopen` or `$fclose`, but the file descriptors for these calls are actually used in `$fdisplay` or `$fwrite` calls in OTB-instances, then the `$fopen` keyword can appear by itself in an instance line, such as `top $fopen`. This line ensures that only the `$fopen` or `$fclose` calls in `top` are overridden by IXCOM; any GFIFO calls or SFIFO definitions in `top` continue to be mapped to the SV-channel.

Compiling C/C++ TB Code

The header file `otb_user.h` contains declarations of additional utility routines (prefixed with `otb_`), described in the Using Additional Utility Functions in Running GFIFO-SFIFO Pairs in Parallel section. If you are using any of these routines, include this file, and recompile your C/C++ code.

When using Xcelium as the software simulator, always include the file `otb_user.h` and recompile your C/C++ GFIFO functions, particularly any code that contains calls to SystemVerilog DPI or VPI routines like `svGetScope`, `svSetScope`, `vpi_get_time` etc.

This recompilation is required because Xcelium is not thread-safe. IXCOM needs to redefine these functions so that these perform correctly in a parallel environment. IXCOM redefines these in Xcelium by defining its own versions of these functions, and by putting a `#define` in the `otb_user.h` file that maps the original function to the IXCOM version of the function. For example:

© 2021 All Rights Reserved.

```
#define svGetScope otb_sv_get_scope
```

Xcelium however, provides IXCOM the capability to override these functions natively. So, you do not need to recompile your C/C++ code when using Xcelium, unless you are using some `otb_` routines. Recompilation is optional if you are using Xcelium and not using any `otb_` routines. However, to maintain performance, it is recommended that you still add `#include otb_user.h` in your code and recompile it.

Ensure that the GFIFO functions written in C/C++ are thread-safe, preferably accessing data only within their own contexts.

Using Additional Utility Functions in Running GFIFO-SFIFO Pairs in Parallel

This section describes the additional utility functions (prefixed with `otb_`) available to OTB-contexts, as well as the SV-context. It also specifies the SUBSET of `svGet`, `svPut`, `vpi_`, and `marg_` utility functions available to OTB-contexts.

A log file is automatically created for each GFIFO-SFIFO channel. The name of the log file is `<channel_name>.log`. For the default channel, the name of the log file is `SV_channel.log`.

Getting the File Pointer to the Log File for Running GFIFO-SFIFO Pairs in Parallel

You can get the file pointer to the log file for the current channel by calling the following function that allows you to direct output from different channels to different log files in parallel. In the software mode, the `otb_get_log_file` file always returns the file pointer of `SV_channel.log` because in the software mode, all GFIFO and SFIFO calls are handled by the `SV_channel`:

```
extern FILE *otb_get_log_file(void);
```

Note that this function is provided to ease the migration and bring-up from single-channel to multi-channel GSFIFO. In this system of logging, the number and contents of the log files will change if you modify the number, name, or mapping of the channels in the channel-configuration file. You can create your own log files for the important hierarchies in the design and manage these yourself. That way, the log files do not change even if the channel-configuration file changes.

Printing the Statistics About the Time Taken by Various GFIFO Or SFIFO Functions

The `+xcprof` option has been enhanced to report performance statistics for each OTB-channel in the `xcprof.out.<channel_name>` file. The statistics for the default channel are printed in the file `xcprof.out` as before. However, you can also call the function `otb_print_stats` from a GFIFO function to print the performance statistics for the current

© 2021 All Rights Reserved.

channel to a file of your choice. If `fp` is `NULL`, the output is sent to the channel-log file (`<channel_name>.log`). If `resetStats` is true, all statistics counters are reset. While using `otb_print_stats` along with the `+xcprof` option, be aware that the `+xcprof` option prints and resets the statistics counters every time the design is swapped out of hardware.

```
extern void otb_print_stats(FILE *fp, int resetStats);
```

Getting the Name of the Channel Servicing the GFIFO Call

You can call the `otb_get_channel_name` function from GFIFO functions to obtain the name of the channel servicing the GFIFO call. This might be useful for debugging or logging etc. In the software mode, `otb_get_channel_name` always returns `SV_channel` because in this mode, all the GFIFO and SFIFO calls are handled by the `SV_channel`.

```
extern char *otb_get_channel_name(void);
```

Determining if the GFIFO Function is Running SV-Context

You can call the `otb_is_sv_context` function from a GFIFO function to determine if it is running in the SV-context.

```
extern int otb_is_sv_context(void);
```

Determining if the GFIFO Function is Running OTB-Context

You can call the `otb_is_otb_context` function from a GFIFO function to determine if it is running in an OTB-context, that is not the SV-context.

```
extern int otb_is_otb_context(void);
```

When you call `otb_is_otb_context()` from GFIFO functions, it is the same as `! otb_is_sv_context()`. However, if you call these functions from a thread that you created, this relationship will not hold. If IXCOM is used, both the `otb_is_sv_context` and `otb_is_otb_context` function will return 0 as expected, when called from your thread. In the Xcelium-only mode, however, the `otb_is_sv_context` function always returns `1`, and the `otb_is_otb_context` function always returns `0`.

Controlling the Frequency for IXCOM to Check for GFIFO Functions in an OTB-Context

The `otb_set_polling_interval` function can be called from a GFIFO function running in an OTB-context. It instructs the OTB-context to poll for the arrival of GFIFO calls from hardware once at the interval specified in the argument, instead of polling continuously. If the polling and execution of GFIFO functions finishes sooner than the interval, then the OTB-context sleeps for the rest of the time so that some other thread or process gets the CPU core. The minimum polling time is `4us` to `5us`. Therefore, the argument should be more than 5. The supported range of values for the argument is `1` through `10,000,000`.

```
void otb_set_polling_interval(int intervalInMicroSecs);
```


© 2021 All Rights Reserved.

This function slows down the GFIFO execution. So, it should NOT be used if enough CPU cores are available to run all of the threads.

Printing a printf-Style Message

You can use the `otb_printf` function to print a `printf`-style message to the channel log file, as well as the screen. Printing to the screen is slow. So, call this function sparingly.

```
extern void otb_printf(char *format, ...);
```

Printing a Fatal Error Message

You can call the `otb_fatalf` function from a GFIFO function to print a fatal error message to the channel-log file and the screen, and gracefully terminate the emulation. Do not call `exit()` or `pthread_exit()` directly from the GFIFO functions because this might cause the emulation to hang or finish abruptly. For errors, call the `otb_fatalf` function. For a normal emulation termination, call `$finish` from the DUT, or from the SV-context in the testbench.

Organize a call to `$finish` from the DUT as follows:

After completion, each SV-context or OTB-context sets its completion status to `1` in the DUT by calling an `SFIFO` function. The DUT `AND`s all of the completion bits and calls `$finish` when all the bits are `1`.

```
extern void otb_fatalf(char *format, ...);
```

Managing File Descriptors for $fdisplay GFIFO Calls in OTB Contexts

Use the DPI-C import function `otb_fopen` before swapping in to establish a file descriptor for `$fdisplay` GFIFO calls in an OTB instance.

```
extern int otb_fopen(const char *fileName, const char *mode);
```

Declare `otb_fclose` as a GFIFO function and call it directly from an OTB instance (or call it from an existing GFIFO function in an OTB instance) to close a file, which was opened using `otb_fopen`.

```
extern void otb_fclose(int fd);
```

Declare `otb_freopen` as a GFIFO function and call it directly from an OTB instance (or call it from an existing GFIFO function in an OTB instance) to close the file corresponding to the given file descriptor and open a new file with the same file descriptor. Use it to close a file when it becomes too large and open a new file in its place.

```
extern void otb_freopen(int fd, const char *fileName, const char *mode);
```

Creating your own pthreads

© 2021 All Rights Reserved.

Typically, you do not need to create your own pthreads because IXCOM automatically creates a pthread for each OTB-channel and runs the GFIFO functions assigned to a channel in the pthread for that channel. If, however, you do need to create your own pthread(s), do not call `pthread_create()` directly. Call `otb_user_pthread_create()` instead. This allows IXCOM to direct Linux signals to the appropriate thread.

```
extern int otb_user_pthread_create(pthread_t *thread,
                                   const pthread_attr_t *attr,
                                   void *(*start_routine) (void *),
                                   void *arg);
```

The extern declarations for all of the utility functions described above are in the `otb_user.h` file.

`otb_user.h` also redefines `pthread_create` to `otb_user_pthread_create` as follows:

```
#define pthread_create otb_user_pthread_create
```

Therefore, if you include `otb_user.h` in your source code files that call `pthread_create`, you do not need to manually replace `pthread_create` with `otb_user_pthread_create`.

Supported svGet and svPut Functions in OTB-Contexts

The only `svGet` and `svPut` functions that you can call from a GFIFO function serviced by an OTB-context are: `svGetScope`, `svSetScope`, `svGetNameFromScope`, `svGetScopeFromName`, and `svGetUserData`.

`svPutUserData` is not allowed because it cannot be overridden in a thread-safe way. However, you can call `svPutUserData` during the initialization or test-configuration phase, before swapping-in to the hardware, to configure the design. So, you can make calls to `svPutUserData` from the SV-context, before the OTB-contexts are activated. After swapping into hardware, the GFIFO functions that are serviced by OTB-contexts can call `svGetUserData` to get these configured values. Do not make any calls to `svPutUserData` from ANY context after the OTB-contexts are activated by swapping-in to the hardware.

Supported vpi_ Functions in OTB-Contexts

The only `vpi_` functions that you can call from a GFIFO function serviced by an OTB-context are `vpi_get_time` and `vpi_printf`. An additional restriction is that the first argument (`obj`) of `vpi_get_time` (`vpiHandle obj`, `p_vpi_time timep`) should be `NULL` when called from an OTB-context. Limit the usage of `vpi_printf` because printing to the screen can become a bottleneck.

© 2021 All Rights Reserved.

Supported marg_ Functions in OTB-Contexts

The only `marg_` functions that you can call from a GFIFO function serviced by an OTB-context are: `marg_gfifo_time`, `marg_otb_vmem_to_c`, and `marg_otb_c_to_vmem`. You can use `marg_gfifo_time` to obtain the hardware time when the GFIFO function being executed was invoked. Use the `marg_otb_` functions to asynchronously transfer data to and from memories in the DUT. The hardware might be running while these functions are reading or writing DUT memories. Also, the previously invoked SFIFO functions might not have completed execution when you call the `marg_otb_` functions. If these SFIFO functions also read or write the memory being accessed by the `marg_otb_` function, then you can make a call to `SFIFO_flush()` before calling the `marg_otb_` function to ensure that all the previously executed SFIFO functions have finished execution prior to the call to the `marg_otb` function.

Call Packet Transportation, Execution, and Synchronization for GFIFO-SFIFO Pairs

Each channel transfers the GFIFO and SFIFO call packets belonging to its portion of the design hierarchy in parallel with other channels. Within each GFIFO-SFIFO channel, the call packets are guaranteed to be executed in the order in which these are generated. However, the order of execution of the GFIFO or SFIFO call packets across different channels is not guaranteed.

Each OTB-context only executes the GFIFO call packets from its OTB-channel, which means that your C/C++ code is initiated by a GFIFO call from its OTB-instances on the hardware side. During the initialization or configuration of a design, you can initiate the communication from the software side by making a call from the SV-context to the OTB-instances on the hardware side. To facilitate this, the SV-context is allowed to make calls to the SFIFO functions defined in an OTB-instance through the corresponding OTB-context and OTB-channel.

The SV-context can call the SFIFO functions defined in both SV-instances and OTB-instances. So, the behavior of `SFIFO_flush()` has been modified for the SV-context to flush the pending SFIFO calls in the channel (SV-channel or OTB-channel) to which the last SFIFO function was directed.

An extra step is required to transfer the call from the SV-context to the corresponding OTB-context. So, make calls from SV-context to the SFIFO functions defined in OTB-instances sparingly, preferably only for initialization.

Call the `SFIFO_send()` function to force the transfer of SFIFO call data from the software to the hardware. If pending SFIFO calls with outputs are present, `SFIFO_send()` behaves just like `SFIFO_flush()`. Otherwise, `SFIFO_send()` returns immediately after initiating the transfer; It does not wait for all of the outstanding SFIFO calls to finish in the hardware before

© 2021 All Rights Reserved.

returning. A latency-sensitive testbench can use `SFIFO_send()` to initiate the transfer of call data for outstanding SFIFO calls at suitable times. Otherwise, IXCOM decides the time to transfer the SFIFO call data, which could be when enough SFIFO calls have been issued, when no SFIFO call has been made for some time, or when `SFIFO_flush()` is called.

Only the pending GFIFO or SFIFO functions in the default channel (`SV_channel`) are flushed on a `tbsync`. The OTB-channels run independently of `SV_channel`, and of each other. These are not explicitly flushed. OTB channels are oblivious of `tbsyncs`, and each OTB-channel continuously processes any pending GFIFO or SFIFO functions in that channel.

If a strict timing-gap control for arriving input transactions is required, the primitive `IXCholdClk` can be used to conditionally hold design clocks until a transaction arrives through the execution of an SFIFO call. The `IXCholdClk` primitive takes a boolean condition as input, and is defined in the <*install_dir*>/tools.lnx64/IXCclkgen.sv file.

Two GFIFO and SFIFO function attributes, namely `global` and `svcontext` facilitate communication between OTB-instances and the SV-context. Specify these attributes in a `$ixc_ctrl` statement or the `+tfconfig` file. For example:

```
initial $ixc_ctrl("sfifo", "mySfifo", "global");
initial $ixc_ctrl("gfifo", "myGfifo", "svcontext", "tb_time");
```

You can apply the `global` attribute to both GFIFO and SFIFO functions. Then, that particular GFIFO or SFIFO function is serviced by the SV-context through the default `SV-channel` even if the module instance containing the GFIFO call or SFIFO definition is an OTB-instance.

You can use GFIFO and SFIFO functions with the `global` attribute to easily pass data between OTB-instances in the hardware and the SV-context in the software. UVM or SystemVerilog testbenches are constrained to run in the SV-context. So, the `global` attribute can be used to allow UVM or SystemVerilog code to interact with OTB-instances in the hardware. For example, an SFIFO function with the global attribute can be called from the UVM or the SystemVerilog code in the SV-context to initialize or configure the OTB-instance(s).

SystemVerilog GFIFO and SystemVerilog SFIFO functions (GFIFO functions written in SystemVerilog and SFIFO functions that can be called from SystemVerilog) require a SystemVerilog simulator (Xcelium) and always run in the default `SV-channel`. These functions behave as if these are imbued with an implicit `global` attribute.

The `svcontext` attribute applies only to the GFIFO functions. Whenever a GFIFO function with this attribute is invoked, its call packet is sent to its OTB-channel. The OTB-context does not execute such a packet. But, it forwards it to the SV-context for execution. The `svcontext` attribute can be used by an OTB-context to notify the SV-context that it has completed a task that is of interest to the SV-context. For example, the `svcontext` attribute can be used to pass a large chunk of data, which was buffered in an OTB-context to the SV-context. An OTB

© 2021 All Rights Reserved.

instance can transfer data from hardware to an OTB context by invoking GFIFO function calls (without the `svcontext` attribute) through an OTB-channel. These GFIFO function calls append the data received in their arguments to a buffer in the OTB-context. Once all the calls to append data have been made, the OTB-instance calls a GFIFO function with the `svcontext` attribute. By the time the SV-context executes this function, all the GFIFO functions invoked by the OTB-context prior to the `svcontext` function would have executed, and all the data would have been appended to the software buffer. So, the SV-context can safely read this software buffer in the C code for the GFIFO function with the `svcontext` attribute. Note that the `global` attribute would not work in this case because the GFIFO call packet with the `global` attribute, instead of the `svcontext` attribute, would go through the SV-channel, not the OTB-channel, and would not be guaranteed to be executed after the previous (non-`svcontext`) GFIFO calls that append the data to the software buffer.

The following table summarizes the location of the GFIFO and SFIFO declarations, the mapping in the channel-configuration file, and the new attributes and how all of these affect the context, channel, and instance used to execute GFIFO/SFIFO functions.

#	Type	Decl Loc / Attribute	Caller Loc	Channel Used	Callee Loc
The following Rows # 1 and 2 show how to create a round-trip path between an SV-instance and the SV-context, through the default SV-channel.					
1	GFIFO	SV-instance / none	SV-instance	SV-channel	SV-context
2	SFIFO	SV-instance / none	SV-context	SV-channel	SV-instance
The following Rows # 3 and 4 show how to create a round-trip path between an OTB-instance and the OTB-context, through the OTB-channel.					
3	GFIFO	OTB-instance / none	OTB-instance	OTB-channel	OTB-context
4	SFIFO	OTB-instance / none	OTB-context or SV-context	OTB-channel	OTB-instance
The following Rows # 5, 6, and 7 show how to create paths between OTB-instances and SV-contexts by using either the default channel or the OTB-channel. Note that the above R0w #4 also describes the creation of such paths.					
5	GFIFO	OTB-instance / `global`	OTB-instance	SV-channel	SV-context
6	SFIFO	OTB-instance / `global`	SV-context	SV-channel	OTB-instance
7	GFIFO	OTB-instance / `svcontext`	OTB-instance	OTB-channel	SV-context

If required, you can use OOMRs in hardware to communicate or coordinate between two OTB-contexts. The first OTB-context makes an SFIFO call, which is serviced by its

© 2021 All Rights Reserved.

OTB-instance. This changes a signal in the hardware that is being monitored as an OOMR by the second OTB-instance. The second OTB-instance can, then, issue a GFIFO call, which will be serviced by the second OTB-context. The argument(s) of SFIFO and GFIFO contain the data that needs to be communicated between the two OTB-contexts.

Calling SFIFO functions from User pthreads

You can create your own pthreads, and make SFIFO calls from these pthreads. Before calling an SFIFO function, the pthread must set the scope properly by calling `svSetScope(scope);`. The SFIFO call is serviced by the channel that you configured to handle that SFIFO function. A user thread starts off in the `NULL` (illegal) scope. The scope handle passed to `svSetScope` can be obtained by calling `svGetScopeFromName`, or by obtaining the scope handle in the SV-context or OTB-context and passing it to the user thread. If you use `svGetScopeFromName`, use it only once to obtain the handle, which should be saved, and reused on future calls to SFIFOs in the same scope.

In the previous VXE versions, you needed to call `pthread_svSetScope` (instead of `svSetScope`) from the user thread to set the scope. This is no longer required, but is still available for backward compatibility.

User threads can also call `svGetScope`, `svGetNameFromScope`, `svGetUserData`, `SFIFO_flush`, and `SFIFO_send`. GFIFO functions are always executed in the SV-context or OTB-context, never in a user pthread.

A user thread cannot call any other `vpi_`, `svGet`, `svPut`, `marg_`, `tf_`, or `otb_` function.

Note that GFIFO functions are always executed in either the SV-context or OTB-context, never in a user thread.

Calling SFIFO functions from user threads is permitted. However, this approach has the following disadvantages.

- You cannot use the same code for simulation and emulation. This is because Xcelium does not allow a user thread to call an export DPI-C function.
- You need to do explicit pthread programming.
- User threads compete with IXCOM threads for CPU cores, and might slow down IXCOM's threads created for OTB-context or SV-context.

If you must use user threads, you can use the same code for simulation and emulation by doing the following:

- Do not directly call SFIFO functions from your thread.

© 2021 All Rights Reserved.

- Instead, append SFIFO call data to a buffer from your thread.
- Periodically call a GFIFO function from hardware, which reads the buffer and invokes the SFIFO call in an OTB-context or SV-context.

Additional Restrictions Related to Running GFIFO-SFIFO Pairs in Parallel

In addition to the restrictions mentioned in previous sections, running GFIFO-SFIFO pairs in parallel has the following restrictions:

- `HMPI`, `+gfifo_async_off`, `reset`, `save-restart` are not supported.
- A blocking `tb_export` function cannot be called from an OTB-context. Only non-blocking `tb_export` functions, that is `tb_export` functions with `sfifo` attribute can be called from an OTB-context.
- An OTB-context can only call an SFIFO function that is mapped to be serviced by its own OTB-channel. Inter-channel calls are not allowed from an OTB-context. However, the `SV-context` can call SFIFO functions serviced by OTB-channels.

Improving Performance of Parallel GFIFO-SFIFO by Using Multiple Processes

This section describes the ways to improve the performance of GFIFO-SFIFO by distributing the GFIFO-SFIFO processing workload of a single large design over multiple processes running on multiple hosts. Here, each host is connected to the emulator that is running that single large design.

Multiple processes are used in the following ways to improve performance:

- Each process is multi-threaded.
- Each thread in each process makes GFIFO-SFIFO calls to access the portion of the design for which it is responsible.
- Each process, typically, runs on a different host.

As the design size increases, the GFIFO-SFIFO processing workload is likely to increase as well. To scale GFIFO-SFIFO performance, you can spread out the increased GFIFO-SFIFO processing workload over more processes on more hosts. Each additional host provides more CPU cores, more memory, more bandwidth to the emulator, and more IO bandwidth to run your GFIFO-SFIFO functions.

Keep in mind that all processes running on the same host share the SA channel on the host. If you run each process on a different host, and no other emulation jobs are running on these hosts, then, each process gets a dedicated SA channel to the emulator.

© 2021 All Rights Reserved.

The multi-process solution provides a C/C++ interface that allows a user-defined process that can directly communicate with the emulator. You can also use this C/C++ interface to build an adaptor that uses a third-party simulator running the testbench to interact with the DUT running on the emulator.

Note: Going forward, this solution will replace HMPI. In HMPI, a user process interacts with the emulator by communicating with the *xmsim* process, which then, interacts with the emulator. This two-step communication is slow and not scalable because the Xcelium process tends to become the bottleneck. HMPI was required because in the previous VXE versions, only one process, which is the Xcelium process could interact with a particular design running in hardware.

Note that the multi-threaded multiple processes solution does NOT replace the multi-threaded single process solution, which is described in the Running GFIFO-SFIFO Pairs in Parallel section. To enable the multi-threaded single process solution, you provide a compile-time channel-configuration file, but no run-time channel configuration file. On the other hand, to enable the multi-threaded multiple processes solution you need to specify both - a compile-time configuration file and a run-time configuration file. If neither configuration file is specified, the single threaded-single process solution is used.

Using Multi-Process GFIFO-SFIFO

The process running xmsim is known as the SV process, and is named `SV_process`. To compile the `SV_process`, specify the additional option -`+mc_gsfifo_config+<compile_config_file>` with the `ixcom` command. This option is described in the Running GFIFO-SFIFO Pairs in Parallel section.

Create additional processes called OTB-processes (Other TestBench processes) by providing the `main` routine, the GFIFO code, and other initialization or support code in C/C++. Compile and link this code with libraries provided by IXCOM.

The IXCOM libraries linked in your OTB-processes enable you to interact with the design running in the emulator as follows:

1. Connect to the emulator.
2. Execute your GFIFO functions in the OTB-context of the OTB-process to which you mapped these.
3. Call SFIFO functions from your GFIFO functions.
4. Disconnect from the emulator.

You can also call utility functions, prefixed with `otb_` and a subset of `svGet`, `svPut`, `vpi_`, and `marg_` functions, from the GFIFO functions running in the OTB-process. These functions

© 2021 All Rights Reserved.

include all the functions allowed in OTB-contexts as described in the Running GFIFO-SFIFO Pairs in Parallel section, and additional `otb_` functions required to connect to and disconnect from the emulator.

Run-Time Configuration File

At run time, start the `SV_process` by using the `+mc_gsfifo_rt_config+<run_time_config_file>` option of the `xrun` command. Specify the mapping in the `<run_time_config_file>` to indicate which process will handle which GFIFO-SFIFO channel(s). Also, start the OTB-processes at the same time, ideally on different hosts. OTB-processes are C/C++ processes. So, these are launched like Linux processes.

To create the run-time configuration file, first decide the number of required processes. Then, name each process and decide the mapping between each process and the corresponding GFIFO-SFIFO channel(s) that each process will handle. These are the GFIFO-SFIFO channels that you declared in the compile-time configuration file specified with the `+mc_gsfifo_config+<config_file>` option.

Consider the following example in which five GFIFO-SFIFO channels are specified in the compile-time configuration file:

```
# Begin compile config file
channel SV_channel LL
# The above line has been added to change the default channel type of
# SV_channel from HB to LL.
# No instance lines are required after the channel line above,
# because IXCOM automatically maps unmapped instances containing
# DPI-GFIFO/SFIFO to SV_channel.
channel channel_A
top.s1 GSFIFO
channel channel_B
top.s2 GSFIFO
channel channel_C
top.s3 GSFIFO
channel channel_D
top.s4 GSFIFO
channel channel_E
top.s5 GSFIFO
# End compile config file
```

The channels can be mapped to three processes in the run-time configuration file as shown below:

```
# Begin run-time config file
# Lines starting with # are comments.
# The format of this file is:
# <process_name> : <channel_name> <channel_name> ...
# One line per process.

# The following line is not really required because any channel
# not mapped to any OTB-process is automatically mapped to SV_process.
```


© 2021 All Rights Reserved.

```
SV_process   : SV_channel channel_A
my_process_1 : channel_B channel_C
my_process_2 : channel_D channel_E
# Only the SV_process can handle SV_channel, if SV_channel is
# mapped in the config file, it should only be mapped to SV_process.
#my_process_1 handles channel_B and channel_C
#my_process_2 handles channel_D and channel_E
# End run-time config file
```

Create the simulation executable (xmsim) for the `SV_process` by using the `ixcom` command to compile the design.

For the run-time configuration file shown above, you need to create the executables for the two OTB-processes by compiling the C/C++ code and linking with libraries provided by IXCOM. For details, refer to the Compiling OTB-Processes section.

You can give any names to the executables. These do not have to be `my_process_1` and `my_process_2`. Multiple OTB-processes can share the same executable. You can call the C function `otb_init(char *<processName>)` from the OTB-process to specify the name of the process to IXCOM. The argument passed to `otb_init` will be `my_process_1` or `my_process_2`.

You can launch all processes at the same time on different hosts. Launching on the same host is permitted and might be useful for debugging, or when this feature is being used as a replacement for HMPI. However, it might result in sub-optimal performance if this feature is being used to improve GFIFO/SFIFO performance.

After being launched, all processes (SV-process and OTB-processes) initialize. The SV-process waits until all OTB-processes have initialized before proceeding further. The SV-process will periodically print out the names of the OTB-processes for which it is waiting (the ones that have not started and/or not initialized yet).

After initialization, OTB-contexts (in `SV_process` and in OTB-processes) wait for the SV process to swap into hardware before processing GFIFO calls in their respective channels.

IXCOM automatically calls the GFIFO functions in the proper OTB-context in the proper OTB-process. For example, GFIFO functions in `top.s4` that are assigned to `channel_D`, would be invoked in the `my_process_2` process in the OTB-context for `channel_D`. IXCOM would create an OTB-context for `channel_D` in the `my_process_2` process during initialization.

Note: Swap out is not supported with multi-process GFIFO-SFIFO. Once the design has been swapped into hardware, it must stay in hardware until the end of emulation.

OTB-processes need not be launched in the directory in which the design was compiled. If an OTB-process is launched in a directory different from the design compilation directory, set

© 2021 All Rights Reserved.

the OTB_DESIGN_DIR environment variable to point to the design compilation directory. The default value of OTB_DESIGN_DIR is the current directory specified as . (dot).

The SV-process and OTB-processes need to pass information to each other, such as communication of initialization state. This is done through files placed in a common directory. By default, this directory is OTB_DESIGN_DIR/tmp/OTB_WORK_DIR. You can override the default setting by setting the OTB_WORK_DIR environment variable.

Note: If the OTB_DESIGN_DIR or OTB_WORK_DIR environment variables are set to a relative path, the path is deemed relative to the current directory.

As user, it is your responsibility to ensure that an empty OTB_WORK_DIR exists before each run. IXCOM automatically empties this directory upon a successful run. However, it might continue to exist in case of a crash. So, it must be emptied before the next run. Execute the rm -rf <*OTB_WORK_DIR*>; mkdir <*OTB_WORK_DIR*> commands to create an empty directory before invoking the SV-process and OTB-processes.

If processes are started on different hosts, killing them all can be difficult in case of a hang, or some other malfunction. In such a case, delete the OTB_WORK_DIR. All processes periodically (every 2 seconds) check for the existence of this directory and kill themselves if it is not there. A new run should not be started until all processes involved in the previous run have terminated.

Following is an example of a simple OTB-process, its compilation and the API functions (otb_init, otb_start, otb_get_instances, otb_get_process_name, and otb_emulation_finished) that can be used to connect to the emulator from the OTB-process.

```
// Begin file : 'my_otb_process_code.c'
/* In this example ALL OTB-processes execute the same code,
 * they work on different instances in the design based on
 * the process name, which is passed in argv[1].
 */

#include <stdlib.h>
#include <stdio.h>
#include "svdpi.h"
#include "otb_user.h"

void initTB(void);

// This is the main routine for an OTB-process.

int main(int argc, char *argv[])
{
  if (argc < 2) {
    printf("ERROR: A process name is required in argv[1].\n");
    exit(1);
  }
  char *processName = argv[1];

  /* You must call otb_init first to let IXCOM know which process
   * this is, and to let IXCOM do its initializations.
```


© 2021 All Rights Reserved.

```
   */
  otb_init(processName);

  /* In between the call to otb_init and the call to otb_start,
   * you can call svGet/SetScope, svGetScopeFromName,
   * svGetNameFromScope, svGet/PutUserData here to associate your
   * data structures to scopes of your interest, as shown in
   * function initTB() below. Later, during GFIFO function execution
   * you can call svGetUserData to retrieve your scope-specific data
   * structures. Any other initialization/configuration that
   * requires calls to the svGet/Set functions listed above should
   * also be done here.
   */

  initTB(); // See definition of initTB below.

  /* You must call otb_start to create OTB-contexts for GSFIFO
   * channels handled by this process, and initiate the processing
   * of GSFIFO calls.
   */

  otb_start();

  /* The main thread can do whatever it wants here. Doing nothing
   * (just waiting for emulation to finish) is OK because GFIFO
   * calls handled by this process will be automatically invoked
   * by IXCOM in OTB-contexts created by IXCOM in this process.
   * These GFIFO calls can make SFIFO calls to send data back to
   * hardware. So, the main thread does not need to be involved in
   * sending transactions/data to or from the hardware. The main
   * thread is a user thread and does have the ability to make SFIFO
   * calls to the hardware, but this is discouraged as it is not
   * efficient. SFIFO calls should be made from GFIFO calls running
   * in OTB-contexts.
   * The main thread can periodically check if emulation has finished
   * using otb_emulation_finished() and exit if emulation is over.
   */

  while (! otb_emulation_finished()) sleep(1);

  return 0;

}

/* Some data that the user wants to associate with each
 * scope of interest.
 */

typedef struct {
  unsigned data[16];
} contextInfo;

void initTB(void)
{
  printf("Process %s is initializing.\n", otb_get_process_name());
  svScope *scopes;
  /* Call otb_get_instances() to get an array of scopes involved
   * with the current process.
   */

  int n = otb_get_instances(&scopes);
  printf("Num scopes = %d\n", n);
  int i;
  for (i = 0; i < n; i++) {
    svScope scp = scopes[i];
    printf("Initializing scope %s\n", svGetNameFromScope(scp));
    contextInfo *ctxInfo = (contextInfo *) calloc(1, sizeof(contextInfo));
```


© 2021 All Rights Reserved.

```
    if (ctxInfo == NULL) {
      printf("ERROR: Scope %s: No more mem.\n", svGetNameFromScope(scp));
      exit(1);
    }
    svPutUserData(scp, (void *) initTB, (void *) ctxInfo);
  }
  /* You can call "contextInfo *ctxInfo =
   * svGetUserData(svGetScope(), (void *) initTB);" * in your GFIFO
   * functions to retrieve the proper context for that scope.
   */
}
// End file : 'my_otb_process_code.c'
```

API Routines For OTB-Processes to Access Emulator

This section describes the API routines that OTB-processes can use to connect and disconnect to or from the emulator. These routines are listed in the above example of the OTB-process.

```
void otb_init(char *processName);
```

You must call `otb_init` from each OTB-process to let IXCOM know which process this is, and to enable IXCOM to do its initializations. Each OTB-process should call it only once, and no two OTB-processes should call `otb_init` with the same process name. `otb_init` waits for the SV-process to initialize. Then, it does its own initializations before retuning.

```
void otb_start(void);
```

After the call to `otb_init`, you must call `otb_start` to tell IXCOM to create OTB-contexts for GSFIFO channels handled by this process, and to initiate the processing of GSFIFO calls. `otb_start` waits until the SV-process has swapped into hardware before returning.

In between the call to `otb_init` and the call to `otb_start`, you can call `svGetScope` `svSetScope`, `svGetScopeFromName`, `svGetNameFromScope`, `svGetUserData`, `svPutUserData` to associate the data structures you have defined to the scopes of your interest. Later, during GFIFO function execution you can call `svGetUserData` to retrieve your scope-specific data structures. Any other user-initialization/configuration that requires calls to the `svGet` and `svSet` functions listed above should also be done here.

```
int otb_emulation_finished(void);
```

The user thread (that is thread running the main routine, or any other user-created thread) can periodically call `otb_emulation_finished` from an OTB-process to check if the emulation is over. Emulation cannot be terminated directly from an OTB-process. Emulation is only terminated from the SV-process, for example when `$finish` or `vpi_finish` is called from SV-process, or when the hardware call calls `$finish`. Typically, the OTB-process will do cleanup and exit when it detects that emulation is over. `otb_emulation_finished` should NOT be called from a GFIFO function because the emulation is obviously not over if a GFIFO function is running.

© 2021 All Rights Reserved.

```
void otb_finish(void);
```

From an OTB-process, the user thread that called `otb_init` and `otb_start` can call `otb_finish` to let IXCOM know that this OTB-process is done with the emulator. IXCOM shuts down the OTB-contexts and disconnects the calling process from the emulator before returning from this function. Emulation continues in the SV-process and other OTB-processes. `otb_finish` can be used as an alternative to calling `otb_emulation_finished` periodically, especially if the calling process can determine by itself whether it is done (for example after completing a pre-determined number of transactions). Once `otb_finish` returns, any subsequent calls to `otb_emulation_finished` return `1` because as far as the calling OTB-process is concerned, emulation is over. Specify the run-time option `+terminate_sv_proc_when_all_otb_procs_done` if you want to terminate emulation after all of the OTB processes have called `otb_finish`, regardless of what the `SV_process` is doing. When this option is specified, the xmsim process immediately calls `$finish` when it detects that all OTB processes have called `otb_finish`.

```
const char *otb_get_process_name(void);
```

`otb_get_process_name` can be called to retrieve the name of the current process. It should only be called after the call to `otb_init` has been made.

```
int otb_get_instances(svScope **scopes);
```

`otb_get_instances` can be called to get an array of scopes that contain GFIFO calls or SFIFO definitions that are handled by the channels that have been mapped to the current process (in the run-time configuration file). It can be used in between the call to `otb_init` and the call to `otb_start` to associate user data with scopes of interest using `svPutUserData`.

Compiling OTB-Processes

Use the following command to compile an OTB-process. It will search for IXCOM and Xcelium header files in all the appropriate directories and link the appropriate IXCOM libraries.

```
gcc -g -m64 -o my_otb my_otb_process_code.c my_gfifo_code.c \
    -I<vxe_install_dirIXCOM_INSTALL_HOME>/include \
    -I<vxe_install_dirIXCOM_INSTALL_HOME>/tools.lnx86/etc/ixcom -
I<ies_install_dir>/include \
    -L<vxe_install_dirIXCOM_INSTALL_HOME>/tools.lnx86/vxe/lib/64bit -ldart -
lotb -lotb_proc \
      <compile_dir>/xc_work/libixcg_otb_proc.so -ldl
```

Include the header file `otb_user.h` in your C code, for example, `my_otb_process_code.c` and `my_gfifo_code.c` before compiling the OTB-process. If you cannot do this for any reasons, for instance if your GFIFO code is available only as a pre-compiled shared library, follow the instructions in the <~~*ixcom_install_dir*~~*IXCOM_INSTALL_HOME*>/share/vxe/etc/ixcom/

© 2021 All Rights Reserved.

`otb_sv_wrapper.c` file to create an additional library called `libotb_sv_wrapper.so` and link this library with the OTB-process.

Launching the SV-Process and OTB-Processes

Use the following commands to launch an emulation that uses multiple-process GFIFO and SFIFO functions. The executable for both the OTB-processes is called `my_otb`, and it expects the process name in `argv[1]`. This process name is passed to `otb_init()` to specify the process name to IXCOM.

```
rm -rf tmp/OTB_WORK_DIR; mkdir tmp/OTB_WORK_DIR
xterm -e rsh host1 my_otb my_process_1 &
xterm -e rsh host2 my_otb my_process_2 &
xterm -e rsh host3 xrun -R -input cmd.i -sv_lib libtb.so
+mc_gsfifo_rt_config+mc_rt_config
```

You can also log into the three machines `host1`, `host2`, and `host3`, and invoke the appropriate commands on each of the three machines.

Reducing Compilation Time and Step Count when Using GFIFO

If there are too many GFIFO calls, the IXCOM compiler automatically employs a multi-level GFIFO scheme to reduce the step count and compilation time, and to support many more GFIFO calls as compared to the original single-level GFIFO scheme.

The number of GFIFO calls in each GFIFO-SFIFO channel is computed, and if it is greater than a certain threshold, that GFIFO-SFIFO channel is implemented using the multi-level GFIFO scheme. GFIFO-SFIFO channels are described in the Running GFIFO-SFIFO Pairs in Parallel section. The default value of this threshold depends on the hardware architecture, such as Protium or Palladium, and on the channel type, such as high bandwidth or low latency.

To change the threshold for enabling this mechanism, specify the `+gfifo_multi_level+<threshold>` option with the `ixcom` command. Here, `<threshold>` is a positive number. To ensure that this mechanism is always enabled, set `<threshold>` value to `1`. To ensure that the scheme is always disabled, set the `<threshold>` value to `1,000,001` or higher.

Following are the advantages of this mechanism:

- The maximum number of GFIFO calls supported in a design increases to `1,000,000`.
- Even when the design contains a large number of GFIFO calls, the step count is reduced and HDL-ICE Compiler requires shorter compilation times.

© 2021 All Rights Reserved.

Note: Other GFIFO limits remain the same. For example, the maximum total argument bit-width of a GFIFO call remains 32K bits. The hardware cost of a GFIFO call remains the same: approximately `10*<local_buffer_size>*<total_argument_size>` gates.

Merging GFIFO Functions for Optimizing Performance

The GFIFO infrastructure uses a 512-bit bus to transfer GFIFO call packets. It can transfer one call packet of size <= 512 bits per FCLK in 2X mode, and two call packets of size <= 512 bits per FCLK in 1X mode. If a call packet is < 512 bits, the 512-bit bus would not be fully utilized.

To improve GFIFO performance for designs containing modules with multiple GFIFO calls with small argument sizes (< 512 bits), that usually fire at the same time, use the `+gfifo_merge_calls` option of the `ixcom` command. This option enables a more efficient use of the GFIFO bus and it can lead to a potentially faster run-time for designs that show a high percentage of time spent in the `Gfifo lb full bwait` cycles category as listed in the IXCOM `xcprof.out` profile file.

For this optimization, IXCOM considers the candidates that can be merged at compile time and generates the merging logic at compile time. For instance, if a design has a module with many GFIFO calls, and each call has 64 bits of arguments, and more than one of these calls tend to fire at the same time, then IXCOM can merge these calls to ensure that the total argument size of the merged calls reaches 512 bits. In 2X mode, this packet of merged calls can be sent on the 512-bit bus together in a single FCLK, instead of taking as many FCLKs as the number of GFIFOs that were fired. The gains in 1X mode are similar (1/2 FCLK vs N/2 FCLKs). The merged calls are reported in the `xc_work/gfifo.log` file.

Note: Each call that is merged requires an extra 32 bits of control data that counts towards the 512-bit limit for merging calls at compile time. If a subset of GFIFOs in a group fires at a particular simulation time, the data for only those calls that fire is sent to the software. If the subset contains a single call, the extra 32 bits of control data for merging mentioned above is not sent to software.

Keep the following limitations in mind while using the `+gfifo_merge_calls` option:

- Only GFIFO calls in the same module are candidates for merging.
- For some designs, the merging logic might increase the step count, thereby negating the gains from fewer `Gfifo lb full bwait` cycles.

© 2021 All Rights Reserved.

Using GSFIFO Input Stream (GSF_IS) Functions to Enable DUT to Fetch Data from Testbench in a Streaming Manner

This section describes the GSFIFO INPUT STREAM (GSF_IS) functions for interface compilation of task and function calls cross the testbench and DUT boundaries. These functions are supported for asynchronous and streaming TBA applications.

In verification tasks, DUT performs computation on data obtained from the testbench. In many cases, such test data consists of a huge number of data items and/or data items are generated on the fly, such as data read from disk files or network sockets. Such data can be best modeled as a stream, that is a sequence of data items from external data sources.

One common method for fetching stream data is to have DUT pull data from testbench instead of having the testbench drive data into DUT. This helps increase verification throughput by minimizing the testbench-side activity.

Following is an example where DUT calls a function in a blocking manner to obtain data.

```
import "DPI-C" function int getData(output bit [31:0] data);
initial $ixc_ctrl("tb_import", "getData");
bit [31:0] data;
always @(posedge clk) begin
       getData(data);
       consumeData(data);
end
```

The above code is not optimal because it requires TBSYNC at every clock cycle in order to execute the import DPI function `getData`. Note that the `getData` function cannot be marked as GFIFO because it has an output argument.

This section describes a new class of interface functions GSFIFO Input Stream (`GSF_IS`) functions, which allows DUT to fetch the data from the testbench in a streaming manner, without TBSYNC overhead.

To use a `GSF_IS` function, define a function that returns an integer value `retval` so that `retval[0] = 1` if the output arguments are valid data items; else it is equal to `0`. If the end of the stream has been reached that is no further function calls will fetch valid data, `retval[1] = 1`; else it is equal to `0`.

Here, `retval[31:2] = 0`, which means that:

- `retval = 0` when there is no data but end of stream has not been reached.
- `retval = 1` when the data is valid but end of stream has not been reached.
- `retval = 2` no data and end of stream has been reached.
- `retval = 3` when the data is valid and it is the last data item, which means that the end of stream has been reached.

© 2021 All Rights Reserved.

Inside DUT, specify the actions depending on the return value of the function call as shown in the following example.

```
import "DPI-C" function int getData(output [31:0] data);
initial $ixc_ctrl("gsf_is", "getData");
bit [31:0] data;
int retval;
  always @(posedge clk) begin
       forever begin
          retval = getData(data);
          if (retval != 0) break;
          @(posedge clk);
       end
       if (retval[0]) // valid data
       consumeData(data);
       if (retval[1]) // end of stream reached
       $finish;
  end
```

Interface tasks and functions and Mapping Control for GSF_IS Functions

For details about interface tasks or functions and possible mapping controls, refer to Interface Task Mapping Control section.

GSF_IS functions are called from DUT to run in software. To control the mapping of a GSF_IS function use the following interface compiler directive:

```
$ixc_ctrl("gsf_is", "<function_name>" [, <mapping_option>]*);
```

You can also use a side file to specify the interface-compiler mapping by using the `+tfconfig+<side_file_name>` option of the `ixcom` command. In the side file, each line specifies a mapping control in the following format:

```
<module_name> <function_name> gsf_is  [<mapping_option>]*
```

The `<module_name>` can be name of a specific SystemVerilog module, interface, or package. Alternatively, use the `*` (asterisk) character, which includes all the modules with the specified task and function name.

The `<function_name>` can be either a specific function name or include the prefix`*` (asterisk) character to match all the functions with the given prefix. However, you cannot have both wildcard module_name (*) and wildcard function_name (prefix*) at the same time.

A GSF_IS function is an interface function called from DUT but runs in software. Keep in mind that:

- GSF_IS function calls do not suspend hardware execution.
- GSF_IS function must have the integer return type. The return value of the function, `retval`, should have the following properties:

© 2021 All Rights Reserved.

- `retval[0] = 1` if the output arguments are valid. Else it is `0`.
- `retval[1] = 1` if the end of stream has been reached. Else it is `0`.
- `retval[31:2] = 0`

The IXCOM compiler maintains an interface buffer of data items and attempts to pre-fetch the data items when the number of data items in the buffer becomes less than the threshold value.

By default, a call to a GSF_IS function is nonblocking. For the hardware caller function, bit `0` of the return value is `1` if the output arguments have been successfully fetched. Else, bit `0` of the return value is `0`.

A GSF_IS function generates a sequence of data items until it reaches the end of the stream. For the hardware caller function, bit `1` of the return value is `1` if it has reached the end of the stream. Else, bit `1` of the return value is `0`.

Options of the GSF_IS Function

`depth=<number>`: Specifies the depth of the buffer, which is used to store the pre-fetched data. The depth value must be a power of `2`, larger than or equal to 4. By default, buffer depth is `32`.

refill=<number>: Specifies the threshold value so that pre-fetching occurs when the number of empty slots in the queue is greater than or equal to the threshold value. The default refill value is `ceil(depth/8)`.

Restrictions of the GSF_IS Functions

Following are the restrictions with respect to the GSF_IS functions:

- GSF_IS functions must have at least one OUTPUT argument. Optionally, the first argument can be an INPUT argument. Refer to the Locally-blocking on User Clock section for using the input argument). INOUT arguments are not allowed.
- GSF_IS function arguments can only have integral or bit vector types. Open arrays arguments are not allowed.
- GSF_IS SV functions are not supported. Only import DPI functions are supported.
- The output of GSF_IS functions should not depend on DUT state or DUT simulation time.
- GSF_IS functions cannot be defined in or called from legacy Verilog modules.
- GSF_IS functions with escaped name are not supported.

© 2021 All Rights Reserved.

- The maximum total bit width of arguments of a GSF_IS function is 32K.
- GSF_IS function call using hierarchical reference via interface port is not supported.
- GSF_IS function call inside a loop that is not guarded by event block is not supported.
- Unpacked datatypes are not supported for parameters in GSF_IS functions.

Examples of GSF_IS Functions

GSF_IS function calls are nonblocking. Therefore, whenever you call a GSF_IS function, you need to check the return value to determine the action to be taken next. Add a simple polling loop to emulate the blocking function call behavior, which means that control should proceed to the next statement only when either the data item is valid and/or the end of the stream has been reached.

Nonblocking Example of a GSF_IS Function

```
// dut.sv

import "DPI-C" function int getData(output bit [31:0] data);
initial $ixc_ctrl("gsf_is", "getData");
bit [31:0] data;
int retval;
always @(posedge clk) begin
    retval = getData(data);
    if (retval[0]) consumeData(data);
    if (retval[1]) $finish;
end

// dpi.c
#define VALID 0x1 // valid data
#define EOS 0x2 // end of stream
#define MAX_TRANSACTIONS 1000
static FILE *fp = NULL;
int num_transactions = 0;
int getData(svBitVecVal *data) {
    if (!fp) {
        if ((fp = fopen("datafile", "r")) == NULL) {
           return EOS;
        }
    }
    if (!feof(fp)) {
        if (fread(data, 4, 1, fp) != 1) // fread failed
            return 0;
        else if (++num_transactions == MAX_TRANSACTIONS)
           // reached end of stream; the last data item
           return VALID | EOS;
        else
          return VALID;
    }
    else
      return EOS;
}
```


© 2021 All Rights Reserved.

Globally-blocking on Behavioral Clock Example of GSF_IS Function

Even if the GSF_IS function calls are nonblocking by default, blocking behavior can be imposed. The following example shows how to freeze the entire design until the `getData` function call obtains the valid data (or end of stream is reached).

```
import "DPI-C" function int getData(output bit [31:0] data);
initial $ixc_ctrl("gsf_is", "getData");
bit [31:0] data;
int retval;
always @(posedge clk) begin
    forever begin
       retval = getData(data);
       if (retval != 0) break;
       // ixc bclk
    end
    // at this point, either data is valid and/or reached EOS
    if (retval[0]) consumeData(data);
    if (retval[1]) $finish;
end
```

The `forever` block can be used as a common idiom for calling the `getData` function to have blocking behavior without advancing time. The `//ixc bclk` pragma is replaced with `@(posedge behav_clk)` and when this statement is reached, the design clock is stopped and behavioral clock is used instead for computation.

The `getData` function is called at time `t`. So, use this global blocking behavior to execute the next statement "`if (retval[0] consumeData(data);` at time `t`.

Locally-blocking on User Clock

The following example shows how to freeze only the `always` block that contains the call to the `getData` function. The other part of the design proceeds while this `always` process blocks until either a data item is successfully fetched and/or the end of stream is reached.

```
import "DPI-C" function int getData(output bit [31:0] data);
initial $ixc_ctrl("gsf_is", "getData");
bit [31:0] data;
int retval;
always @(posedge clk) begin
    forever begin
        retval = getData(data);
        if (retval != 0) break;
        @(posedge clk);
    end
    // at this point, either data is valid and/or reached EOS
    if (retval[0]) consumeData(data);
    if (retval[1]) $finish;
end
```

Unlike the example in Globally-blocking on Behavioral Clock Example of GSF_IS Function, the `forever` block illustrated above waits for the result of `getData` using the design clock.

© 2021 All Rights Reserved.

This is useful if the exact time when the `getData` function call returns (with valid data) does not matter. Compared to Globally-blocking on Behavioral Clock Example of GSF_IS Function, this code is RTL and does not have behavioral-instrumentation overhead.

Using Stream Handles in GSF_IS Functions

The first argument of a GSF_IS function can be an INPUT argument. One restriction is that the actual value of this INPUT argument must be constant in all calls to the GSF_IS function.

Handles can be used as caller scope IDs when a GSF_IS function is called from multiple places within DUT. The handle can be used to either identify the caller of the function or as a file descriptor as shown in the following example.

```
// dut.sv
import "DPI-C" function int getHandle(strings s);
import "DPI-C" function int getData(input int handle,
                                    output bit [31:0] data);
initial $ixc_ctrl("gsf_is", "getData");
initial h = getHandle("test.data");
bit [31:0] data;
int retval;
always @(posedge clk) begin
    forever begin
        retval = getData(h, data);
        if (retval != 0) break;
        @(posedge clk);
    end
    if (retval[0]) consumeData(data);
    if (retval[1]) $finish;
end
```

```
// dpi.c
#define VALID            0x1 // valid data
#define EOS              0x2 // end of stream
#define MAX_TRANSACTIONS 1000
static FILE *fps[MAX_FILES];
static int nfiles = 0;
int getHandle(char *name) {
    FILE *fp;
    if ((fp = fopen(name, "r")) == NULL) return -1;
    fps[nfiles++] = fp;
    return nfiles-1;
}
static FILE *fp = NULL;
int num_transactions = 0;
int getData(int handle, svBitVecVal *data) {
    FILE *fp = fps[handle];
    if (handle >= MAX_FILES || !fp) return EOS;
    if (!feof(fp)) {
        if (fread(data, 4, 1, fp) != 1)
            return 0;
        else if (++num_transactions == MAX_TRANSACTIONS)
            return VALID | EOS;
        else
            return VALID;
    }
```


© 2021 All Rights Reserved.

```
    else
        return EOS;
}
```

DPI Task Mapping for Parallel C-Based Testbench

To support DPI tasks with blocking semantics, IXCOM coordinates with the software simulator Xcelium to regulate the flow of control and data between hardware and software.

The `tbrun` execution mode alternates the execution between hardware and software to preserve the simulation semantics as much as possible. The `tbsync` events, such as blocking DPI calls, trigger context-switching between the software and hardware execution.

The `nbrun` execution mode introduces parallel execution between the testbench and hardware. In the `nbrun` mode, the tbsync events stop the hardware execution and wait for the testbench to process output events. The `nbrun` mode works well with C-based testbenches that use GFIFO or SFIFO. The `nbrun` mode does not work well with C-based testbenches that call the blocking `tb_import` or `tb_export` tasks. Blocking task calls trigger `tbsync` events and limit parallel software-hardware execution.

The GFIFO and SFIFO functions provide non-blocking streaming parallel execution. But these are limited to one-way functions and do not cover parallel execution of DPI tasks.

In verification environments that use parallel testbenches, such as SV-threads in `nbrun` mode or pthreads in C calls with export DPI tasks, you need to use the `pt_import` and `pt_export` classes of interface tasks.

Note that `pt` stands for Parallel Testbench and in the `tb_import` and `tb_export` mapping `tb` stands for default TestBench.

- A `pt_export` task (or function) is an export DPI task (or function) defined in the DUT and called from SV-thread or pthread. Unlike `tb_export` tasks (and functions), `pt_export` tasks (and functions) can be safely called from pthreads asynchronously.
- A `pt_import` task is an import DPI task defined in C and called from the hardware. It is executed in an SV-thread in the `nbrun` mode or a pthread. It provides a convenient way to call a C task to run in parallel with the hardware.
- A `pt_import` function is an import DPI-C function that is defined in C and called from the hardware. A `pt_import` function call does not create a `pthread` but holds simulation time until the call returns.

To control the mapping of a `pt_export` and `pt_import` task (or function), use the following interface compilation directives:

```
$ixc_ctrl("pt_import", "taskfunc_name" ["async" | "sv_thread"]);
```


© 2021 All Rights Reserved.

```
$ixc_ctrl("pt_export", "taskfunc_name");
```

Note: For `pt_import` functions, the `async` and `sv_thread` options do not apply.

You can also use a side file to specify the interface compilation mapping with the `+tfconfig+<side_file_name>` option of the `ixcom` command.

In the side file, each line specifies a mapping control in the following format:

```
<module_name> <taskfunc_name> pt_import [async | sv_thread]
<module_name> <taskfunc_name> pt_export
```

The <*module_name*> can be a specific SystemVerilog module, interface, or package name, or the asterisk `*` character, which implies that the mapping control is specified for all modules with the given task and function name.

The< *taskfunc_name*> can either be a specific task or function name or have a prefix * to match all tasks and functions with the given prefix. However, you cannot have both wildcard <*module_name*>(`*`) and wildcard <*taskfunc_name*> (prefix `*`) at the same time.

An import DPI routine declared in a compilation-unit or regular package can be marked `pt_import` by using command-line arguments or a configuration file. However, tasks or functions in a compilation-unit scope or package cannot be marked `tb_export` or `pt_export`.

These guidelines are similar to the ones mentioned in Specifying Interface Mapping Control Using a Side File.

Using pt_export Tasks and Functions

A `pt_export` function/task is an export DPI-C function (or task), which is defined in DUT and called from a software thread like SV-thread, `pt_import` pthread, or user-created pthread.

Execution semantics for `pt_export` tasks (or functions):

- Blocking: A call to `pt_export` task or function suspends the caller pthread or SV-thread until the task or function finishes execution in hardware.
- A `pt_export` task or function runs in hardware and can reference hardware objects outside of its scope.
- Calling a `pt_export` task from SV-thread in the `tbrun` mode triggers a `tbSync` event on return and the hardware waits for the software side to complete processing the output events just like a `tb_export` task call.

© 2021 All Rights Reserved.

- Calling a `pt_export` task from SV-thread in the `nbrun` mode does not trigger a `tbSync` event on return. The hardware runs in parallel while the software side is processing the output events.
- Calling a `pt_export` task from a pthread does not trigger a `tbSync` event on return. It has the same behavior as if calling from a SV-thread in the `nbrun` mode.

Restrictions for pt_export Tasks or Functions

Keep the following points in mind with respect to the restrictions related to `pt_export`:

- The `pt_export` task and function calls are not buffered and can only handle one caller at a time.
- If two asynchronous pthreads call the same `pt_export` function instance at the same time, a run-time warning is given and their execution is serialized in random order.
- If two SV-threads or synchronous pthreads call the same `pt_export` task, a fatal error is generated because both the single-caller constraint and synchronized execution semantics cannot be satisfied.
- The body of the `pt_export` task or function must meet the requirements list for the IXCOM-extended RTL and behavioral subsets for mapping to run in hardware.
- `pt_export` tasks and functions are not supported in compilation-unit scope or package.
- Arguments of `pt_export` tasks and functions can only have packed datatypes or unpacked struct type.
- The `pt_export` functions can have output arguments and can return values of the following scalar data types: bit, logic, byte, shortint, int, longint, and enum. Other datatypes, such as chandle, integer, and so on are not supported.
- The `pt_export` functions cannot call import DPI functions. Neither `tb_import` nor `pt_import` functions are allowed.
- The `pt_export` tasks or functions do not work in software-simulation mode. These must be called after swap-in to hardware.
- The `pt_export` tasks and functions cannot be called by OTB processes. In Multi-process runs, all `pt_import` and `pt_export` tasks (or functions) are mapped to run in the SV process.

© 2021 All Rights Reserved.

Using pt_import Tasks

A `pt_import` task is an import DPI-C task that is defined in C and called from the hardware. Calling a `pt_import` task instance creates a server pthread to execute the task in parallel to all the SV-threads in the simulator.

Calling a `pt_import` task creates a SV-thread if the `sv_thread` option is used in the mapping definition.

A pthread `pt_import` task call does not trigger a `tbSync` event because it is not a part of the SV-thread system. However, such a task call can stop the hardware by using a different sync or async control mechanism.

In the `tbrun` mode, a SV-thread `pt_import` task call triggers a `tbSync` event to stop the hardware and waits for the software to complete the C processing.

In the `nbrun` mode, an SV-thread `pt_import` task call does not trigger a `tbSync` event and the hardware runs in parallel while the software is doing the C processing.

Execution semantics for `pt_import` tasks:

- Locally blocking: A call to the `pt_import` task suspends the calling DUT process — the `always` statement. This is applicable for both pthread and SV-thread `pt_import` tasks.
- A pthread `pt_import` task has two global execution modes
  - Synchronous mode: Emulator holds the simulation time until the call returns or a time-consuming `pt_export` task call.
  - Asynchronous mode: Emulator continues to advance the simulation time and runs the hardware parallel to the `pt_import` task in software.

Restrictions for pt_import Tasks

Keep the following points in mind with respect to the restrictions related to `pt_import` tasks:

- A `pt_import` task cannot call `tb_export` tasks, SFIFO tasks, or `tb_task` tasks or functions.
- Arguments of `pt_import` tasks can only have packed datatypes or unpacked struct. Open array arguments are not allowed in `pt_import` tasks or functions.
- `pt_import` tasks do not work in the software-simulation mode. These must be called after swap-in to hardware.

Server pthread Control for pt_import Tasks

© 2021 All Rights Reserved.

The main application of pthread `pt_import` task is to create parallel traffic-generator threads for driving transactions into DUT through the `pt_export` tasks and function calls. Such traffic-generator threads run to the end of emulation and, by default, the return of a `pt_import` task terminates the server pthread.

A minor application of pthread `pt_import` tasks is for BFMs to get data from a software device model, such as reading a software DRAM model. With the default global synchronization mode of execution, pthread `pt_import` mapping can achieve significant performance gain compared to the default `tb_import` mapping for the same import DPI-C task if you keep the server pthread for future call processing.

You can keep a server pthread on return by calling the following marg control function:

```
#include "marg_user.h"
marg_pt_import_keep_server(on)
```

`marg_pt_import_keep_server(1)` keeps the server pthread for the calling task instance on return.

`marg_pt_import_keep_server(0)` changes the calling task instance back to the default behavior of terminating the server pthread on return.

The performance gain comes from the dedicated server pthread and no `tbSync` processing overhead. However, you cannot replace all `tb_import` mapping by pthread `pt_import` mapping to gain performance because of the following reasons:

- The pthread `pt_import` mapping requires the import DPI-C task to be thread-safe and not conflict with the GFIFO processing in the main SV simulator thread.
- A CPU has limited cores to support server pthreads. When no core is available to run a server pthread, the Linux process slows down and might even hang.

You should only map timing-critical import DPI-C tasks to pthread `pt_import` to improve BFM performance. Timing critical tasks are called at very high frequency and must be pthread-safe. You can use the SA run-time command (`xrun +xcprof`) to help identify these tasks. Refer to Profiling Design Run-Time Performance on Hardware with XC Profiler section for details.

Controlling Synchronous and Asynchronous Execution Modes

By default, all pthread `pt_import` tasks are configured to run in global synchronous mode at initialization. In this mode, a call to the pthread `pt_import` task holds the simulation time (RTL clocks) in hardware. However, it does not trigger a `tbSync` event. It holds the simulation time until the following events have occurred:

- A time-consuming `pt_export` task is called from itself or other active `pt_import` task instances.

© 2021 All Rights Reserved.

- The call returns from software.

Releasing the hold on simulation time is necessary for time-consuming `pt_export` tasks to execute. The simulation time hold will resume at the cycle that all active time-consuming `pt_export` tasks are completed in hardware.

When multiple `pt_import` tasks are active, the emulator will hold the simulation time if one of these tasks requests to hold the simulation time. You can dynamically release the clock hold by calling the following marg control function:

```
#include "marg_user.h"
marg_pt_import_run_async(int on)
```

`marg_pt_import_run_async(1)` reconfigures the calling pthread `pt_import` task instance to run in the global asynchronous mode and immediately releases the simulation time hold to let the hardware run in parallel with the software asynchronously.

`marg_pt_import_run_async(0)` reconfigures the calling pthread `pt_import` task instance back to the global synchronous mode and immediately holds the simulation time in hardware.

Typical traffic generator pthread `pt_import` tasks need to perform initialization and reset processing in synchronized mode with accurate simulation time control. Once complete, these can switch to asynchronous mode by calling `marg_pt_import_run_async(1)` and let the hardware be free-running and asynchronously pump transactions to the hardware.

Just like an SV-thread `pt_import` task call in the `nbrun` mode, a pthread `pt_import` task call in the asynchronous mode does not finish in zero simulation time even if it is a non-time-consuming task in simulation. All `pt_import` task calls should be treated as if a time-consuming import DPI-C task and its execution time is non-deterministic per call. You must use behavioral logic to hold RTL clocks if deterministic round-trip latency is needed in BFM modeling.

You can use also the Xcelium run-time command option to initialize all the pthread `pt_import` tasks to start in the asynchronous mode:

```
xrun +pt_import_async_task
```

Alternatively, you can use the compile-time `pt_import` configuration option `async` to initialize a specific pthread `pt_import` task to start in the asynchronous mode.

A `pt_import` task initialized to `async` mode will enter `async` mode (releasing the hold on the simulation time in hardware) at the first `pt_export` task call or at the second call event if the first call event does not call a `pt_export` task. This delay allows for default initialization processing in a typical traffic generator. Call `marg_pt_import_run_async(1)` to start `async` execution prior to the default call or event.

© 2021 All Rights Reserved.

Using pt_import Functions

A `pt_import` function is an import DPI-C function that is defined in C and called from the hardware. A `pt_import` function call does not create a pthread but holds simulation time until the call returns.

A `pt_import` function call can be treated as `tb_import` function call that does not trigger a `tbSync` event but holds simulation time and and flushes the call packets in the SV-channel GFIFO.

Note: Only the GFIFO in the SV process are flushed. GFIFO in the OTB channels are not flushed.

Restrictions for pt_import Functions

Keep the following points in mind with respect to the restrictions related to `pt_import` functions:

- A `pt_import` function cannot call any export DPI-C function. It must be a simple C function.
- Arguments of `pt_import` functions can only have packed datatypes or unpacked struct. Open array arguments are not allowed in `pt_import` tasks and functions. String arguments are supported in `pt_import`, including all parameter directions: input, output, and inout.
- The `pt_import` functions can have output arguments and can return values of the following scalar data types: bit, logic, byte, shortint, int, longint, and enum. Other datatypes, such as chandle, integer, and so on are not supported.
- The `pt_import` functions do not work in the software simulation mode. These must be called after swap-in to the hardware.

Example: Pthread Traffic Generator (pt_import Task)

This example shows how to create a pthread traffic generator using a pthread `pt_import` task.

```
// dut.sv
module dut;
   export "DPI-C" task drive;
   task drive(input int data);
     while (!ready) @(posedge clk);
     bus <= data;
   endtask

   export "DPI-C" task resetT;
   task resetT();
     reset <= 1;
     repeat(4) @(posedge clk);
```


© 2021 All Rights Reserved.

```
    reset <= 0;
  endtask
  initial $ixc_ctrl("pt_export", "drive", "resetT");
  import "DPI-C" task run_drive_thread();
  always @(posedge clk) begin
    run_drive_thread();
    $finish;
  end
  initial $ixc_ctrl("pt_import", "run_drive_thread");
endmodule
// dpi.c
  #include "marg_user.h"
  extern int drive(int data);
  extern int resetT();
  void run_drive_thread() {
    FILE *fp = fopen("testfile", "r");
    int data;
    resetT();
    marg_pt_import_run_async(1);
    while (!feof(fp)) {
      fscanf(fp, "%d\n", &data)
      drive(data);
    }
  }
```

Example: Short-Latency Memory BFM model (pthread pt_import Task)

This example shows the use of pthread `pt_import` tasks, in the default synchronous mode, for replacement of the `tb_import` tasks for latency.

The task must be configured with `marg_pt_import_keep_sever(1)` to get the performance benefit.

```
// bfm.sv
module bfm(clk, cmd, addr, din, dout);
input clk;
input [7:0]     cmd;
input [9:0]     addr;
input [31:0]    din;
output[31:0]    dout;
bit[31:0]       dout;
always@(posedge clk) begin
   if(cmd!=0) begin
       memRW(cmd, addr, din, dout);
   end
end
import "DPI-C" task
   memRW(input  bit [7:0]  cmd, bit [9:0] addr, bit[31:0] din,
         output bit [31:0] dout);
initial begin
   $ixc_ctrl("pt_import", "memRW");
end
// dpi.c
#include "marg_user.h"
unsigned mem[1024];
void memRW(unsigned *cmd, unsigned *addr, unsigned *din, unsigned *dout){
   if(*cmd == 1){
```


© 2021 All Rights Reserved.

```
      mem[*addr] = *din;
   } elseif(*cmd == 2){
     *dout = mem[*addr];
   }
   marg_pt_import_keep_server(1);
}
```

Example: Memory BFM model (pt_import and GFIFO functions)

This example shows the use of `pt_import` function as a `tb_import` function replacement to flush GFIFO without paying `tbSync` overhead. No pthread is created in this example.

```
// bfm.c
module bfm(clk, cmd, addr, din, dout);
   input clk;
   input [7:0]      cmd;
   input [9:0]      addr;
   input [31:0]     din;
   output [31:0]    dout;
   bit [31:0]       dout;
   always@(posedge clk) begin
      if(cmd == 1)
         memW(addr, din);
      else if(cmd == 2)
        memR(addr, dout);
   end
   import "DPI-C" context
     function void memR(input bit [9:0] addr, output bit [31:0] dout);
   import "DPI-C" context
     function void memW(input bit [9:0] addr, input bit[31:0] din);
   initial begin
      $ixc_ctrl("pt_import", "memR");
      $ixc_ctrl("gfifo", "memW");
   end
endmodule
```

```
// dpi.c
unsigned mem[1024];
void memW(unsigned *addr, unsigned *din){
  mem[*addr] = *din;
}
void memR(unsigned *addr, unsigned *dout){
  *dout = mem[*addr];
}
```

Example: Controlling RTL Clocks in pthread Asynchronous Mode (pthread pt_import Task)

You can run the same BFM memory model in the pthread `pt_import` task example in asynchronous mode. Running memory transactions in parallel with the design can further improve the performance. However, you must use behavioral logic to control the RTL clocks to meet the latency requirement of the memory model:

- The `pt_import` task call is blocking but will not finish in 0 time in asynchronous execution mode. The BFM must hold RTL clocks while detecting the next transaction

© 2021 All Rights Reserved.

when the previous one has not completed. Without holding the clock, it will miss the new transaction.

- The number of clock cycles needed to finish a `pt_import` task call is not deterministic. Holding RTL clocks is necessary to meet the latency assumption of a transaction.

```
// bfm.sv
module bfm(clk, cmd, addr, din, dout);
    input clk;
   input [7:0]     cmd;
   input [9:0]     addr;
   input [31:0]    din;
   output[31:0]    dout;
   bit[31:0]       dout;
   bit             txOn;
   // tbxOn must use blocking assignment (=), not (<=)
   // so the behavioral clock hold clock logic can see its value
   // change in the same RTL cycle (simulation time).
   always@(posedge clk) begin
      if(cmd == 1 || cmd == 2) begin
          txOn = 1;
          memRW(cmd, addr, din, dout);
          txOn = 0;
      end
   end
   import "DPI-C" context
     function void memRW(input  bit [7:0]  cmd, bit [9:0] addr,
     bit[31:0] din, output bit [31:0] dout);
     initial begin
     $ixc_ctrl("pt_import", "memRW", “async”);
   end
   // wait for previous transaction to complete
   always@(negedge clk) begin
      if( cmd == 1 || cmd == 2) begin
          while(txOn) begin
             //ixc bclk
          end
      end
   end
   // max read latency is 3 clocks
   bit [7:0] w1;
   always@(negedge  clk) begin
      if( cmd == 2) begin
          w1 = 0;  do @(negedge clk) w1+=1; while(w1 < 3 && txOn);
          while(txOn) begin
            //ixc bclk
            w1+=1;
          end
      end
   end
endmodule
```

Using Parallel C-based Testbench to Your Existing Environment

For C-based testbench code that is not pthread-safe, you can apply `pt_export` and SV-thread `pt_import` tasks with the `nbrun` mode to speed up execution time via parallel software-hardware execution while keeping software-execution scheduled and sequential.

© 2021 All Rights Reserved.

For C-based testbench code that is pthread-safe, you can apply pthread `pt_import` tasks to create pthreads to further speed up execution time via parallel execution within the C-based testbench code.

You can use the global-mapping compilation options of the IXCOM compiler to first try the `nbrun` mode before selectively applying pthread `pt_import` mappings. These IXCOM options change the default mapping of DPI-C functions and tasks globally:

- `+pt_export_dpi_task`: changes the default mapping of export DPI tasks to `pt_export` (instead of `tb_export`)
- `+pt_export_dpi_func`: changes the default mapping of export DPI functions to `pt_export` (instead of `tb_export`)
- `+pt_export_dpi`: apply both the `+pt_export_dpi_task` and `+pt_export_dpi_func` options together.
- `+ptsv_import_dpi_task`: changes the default mapping of import DPI tasks to `pt_import` using SV-thread (instead of `tb_import`)
- `+ptsv_import_dpi_func`: changes the default mapping of import DPI functions to `pt_import`.
- `+ptsv_import_dpi`: apply both the `+ptsv_import_dpi_task` and `+ptsv_import_dpi_func` mapping options.

These compile-time mapping options do not force the `nbrun` option at run time. The `pt_import` and `pt_export` mapping options have the same blocking behavior as `tb_import` and `tb_export` in the `tbrun` mode and only have non-blocking parallel behavior in the `nbrun` mode. You can first validate the compile result in the `tbrun` mode before testing out the `nbrun` mode without needing to re-compile the design.

First, you can apply `+pt_export_dpi_task` mapping to speed up proactive testbenches, such as traffic generators, in the `nbrun` mode.

Second, you can apply `+ptsv_import_dpi_func` mapping to speed up function-based reactive testbenches, such as non-time-consuming virtual device models, in the `nbrun` mode. This option can reduce latency and overhead in software processing while keeping the same synchronous `tb_import` function behavior in both `tbrun` and `nbrun` modes.

Third, you can apply `+ptsv_import_dpi_task` mapping to speed up task-based reactive testbenches, such as time-consuming virtual device models, in the `nbrun` mode. However, this mapping works only if BFMs are not sensitive to round-trip clock delays. Without knowing the BFM-timing constraints, you can still apply the `+ptsv_import_dpi_task` mapping at compilation time and use the options of the `xrun` run-time command.

© 2021 All Rights Reserved.

`+ptsv_import_nbrun_sync` forces SV-thread `pt_import` tasks to behave like `tb_import` tasks in the `nbrun` mode without recompiling the design.

You must apply `+pt_export_dpi_func` when using pthread `pt_import` task mappings so that the pthread can call export functions. Using the `+pt_export_dpi_func` mapping does not have performance impact in the `nbrun` mode with only SV threads.

Using tb_task Tasks and Functions

The `tb_task` tasks and functions are called from testbench to run in software.

Mapping export DPI-C or SystemVerilog tasks and functions to `tb_task` requires explicit mapping control. Without explicit mapping control, DPI-C export tasks and functions will default to `tb_export` mapping into hardware.

Compiler mapping of `tb_task` tasks and functions:

- Synchronous: Testbench can only initiate a `tb_task` task call when hardware execution is on halt and testbench is active.
- A `tb_task` task runs in software and can reference hardware objects out of its task or function scope with cost.

Restrictions:

  - A `tb_task` body can have behavioral constructs that are beyond the hardware subset because it is executed in software.
  - A `tb_task` can also have non-RTL argument types (for example SystemVerilog unbound arrays and string) because callers are from testbench, not from DUT.

Using tb_init Tasks and Functions

The `tb_init` tasks are called in initial time only.

Mapping SystemVerilog tasks and functions to `tb_init` tasks and functions requires explicit mapping using `$ixc_ctrl`. The IXCOM compiler does not instrument such a task or function.

Use these tasks and functions for initialization only before hot swap.

© 2021 All Rights Reserved.

Using tb_inline Tasks and Functions for Execution in Hardware and Software

By default, IXCOM allows a DUT task or function to be mapped to execute in either the software partition or the hardware partition; but not in both partitions. Unless a DUT task or function is specified to be an interface task or function, it is mapped to execute in the hardware partition. It is not possible to specify that the software callers to a DUT task or function execute the task or function in the software partition and the hardware callers execute the task or function in the hardware partition. In order to achieve this behavior, you need to manually modify the design by taking the following steps:

1. Create a copy of the DUT task or function in the same module as the original DUT task or function and rename the copy to have a unique name.
2. Specify `tb_task` mapping control for the original DUT task or function. This maps the original task or function to be executed in the software partition, while the copy of the DUT task or function is mapped to be executed in the hardware partition.
3. Modify the hardware callers to the task or function in the design to call the renamed copy of the task or function.

Instead of taking the above steps, you can use the mapping control option `tb_inline` that enables the interface tasks and functions that are declared (and defined) in the DUT partition to be connected to the testbench, and, thereby, enable the task or function to be mapped to execute in both software and hardware partitions. The mapping control option `tb_inline` enables IXCOM to automatically clone a DUT task or function (TF) so that the original copy of the task or function is mapped to execute in the software partition and the cloned one is mapped to execute in the hardware partition. IXCOM also automatically updates the callers to the task or function so that each caller calls the copy of the task or function mapped to execute in the partition of the caller.

In this section, a *Software Caller* for a task or function is a statement mapped to the software partition that calls the task or function, and a *Hardware Caller* for a task or function is a statement mapped to the hardware partition that calls the task or function.

You can specify the `tb_inline` mapping control for a task or function in the following ways:

- Using the `$ixc_ctrl` pragma:

  ```
  $ixc_ctrl("tb_inline", "<TF_name>")
  ```

- Specify a configuration file with the following format by using `+tfconfig+<config_file_name>` option:

  ```
  <module_name> <Task or function name> tb_inline
  ```

Following are some examples for specifying the tb_inline mapping control.

```
// S1 is a caller of 't_b' and 't_b' is a callee of S1.
// The always process is a caller of 't_b' and the 't_b' is a callee
```


© 2021 All Rights Reserved.

```
// of the always process.
always@(posedge clk)
        t_b();        // statement S1
end

task t_a();
     t_b();           // statement S2
end

task t_b;             // 't_b' is a caller of 't_c'.'t_c' is a callee of 't_b'
     t_c();           // statement S3
endtask
```

In the above examples, `t_b` is called by S1 and S2 statements. which are within the `always` process and `t_a` task, respectively. Thus, `S1`, `S2`, the `always` process, and `t_a` are all *callers* of `t_b`, and `t_b` is a *callee* of `S1`, `S2`, the `always` process, and `t_a`. `S3` and `t_b` call `t_c`. So, both `S3` and `t_b` are callers of the `t_c` scope. Also, `t_c` is a callee of both `S3` and `t_b`.

Restrictions in Using tb_inline Tasks and Functions

A tb_inline task or function can be called from any software or hardware scope. However, keep the following considerations in mind when using the tb_inline mapping control:

- `tb_export` or `SFIFO` tasks and functions cannot be called from `tb_inline` tasks and functions.
- `tb_task` tasks and functions cannot be called from `tb_inline` tasks and functions.
- All `tb_inline` tasks and functions are cloned by IXCOM so that one version of the task or function executes in the hardware partition and another version of the task or function executes in the software partition. Thus, only those tasks and functions that are allowed to be called from the software partition as well as from the hardware partition can be called within the body of a `tb_inline` task or function.

  Therefore, a task or function marked as `tb_export` cannot be called from within a `tb_inline` task or function. For such designs, the version of the `tb_inline` task or function that is mapped by IXCOM to execute in the hardware, that is the cloned version will contain calls to the task or function marked as `tb_export`. So, because calls to a task or function marked as `tb_export` from a scope mapped to the hardware partition are not allowed, such usage of `tb_inline` mapping control is not supported. Similarly, calls totasks or functions marked as `tb_task` from `tb_inline` tasks or functionsa are not supported.

© 2021 All Rights Reserved.

The following table describes the output for possible mappings of tasks and functions that are called from within a `tb_inline` task or function. The callee task or function column lists all possible callee tasks and functions for the `tb_inline` task or function.

Callee Task or Function	Output	Supported
No marking - implies that it is mapped to hardware	Warning	Yes. But it is not recommended.
tb_import task or function or GFIFO function	No warning or error	Yes
tb_export task or function or SFIFO function	Error	No
tb_task	Error	No
tb_inline	No warning or error	Yes

As shown in the above table, a task or function, which is called by a `tb_inline` task or function must be either a `tb_inline` or a `tb_import` task or function or a GFIFO function to ensure that no warnings or errors are issued. If a `tb_inline` task or function calls a task or function that is not marked using any mapping control, IXCOM issues a warning.

Consider the following example:

```
test.sv:
module m1(input logic clk);
       task m1_sw();
          m2i.tbi();
       endtask
       task m1_hw();
          m2i.tbi();
       endtask

       initial begin
               $ixc_ctrl("tb_export", "m1_hw");
               $ixc_ctrl("tb_task",   "m1_sw");
       end

       m2 m2i();
endmodule

module m2;
       task tbi;
       endtask

       initial begin $ixc_ctrl("tb_inline", "tbi"); end
endmodule

module top;
       wire clk;
       m1 m(.clk(clk));
endmodule
```

The above example is compiled with the following command:

© 2021 All Rights Reserved.

```
ixcom -ua +dut+top test.sv
```

Fine-Tuning Behavioral Performance

This section describes the levels of behavioral evaluation reported in the working directory and the steps to identify the source of behavioral evaluations.

Reporting Behavioral Evaluation

The following levels of behavioral evaluation are reported in the working directory for any hardware run:

1. The total number of behavioral evaluations is reported in `xrun.log` and `tmp/xc_run.log` as `bevals` in the `HW execs` statistics. For example:

   ```
   --- HW execs 1000 evals 300 bevals 100 tbsyncs
   ```

2. Behavioral evaluations for individual module instance is reported in `tmp/xc_run.log` in decreasing order of the number of evaluations. For example:

   ```
   ======================================================
      Behavioral evals statistics:
   ------------------------------------------------------
         213 dut.trans2
         153 dut.trans3
         21 dut.trans1
   ======================================================
   ```

   Multiple module instances might be active during a single behavioral evaluation. So, the sum of these behavioral evaluations can be greater than the total number of behavioral evaluations reported in the `HW execs` statistics.

3. When the design is compiled with the `+bcDebug` option of the `ixcom` command, detailed breakdown of behavioral evaluations for individual behavioral events are provided in the statistics in `tmp/xc_run.log`. For example:

   ```
   ======================================================
      Behavioral evals statistics:
   ------------------------------------------------------
         213 dut.trans2
            106 _zyL86_pbcCapOut2
            213 _zyL74_pbcCapOut1
         153 dut.trans3
            153 _zyL99_pbcCapOut1
         21 dut.trans1
            10 _zyL86_pbcCapOut2
            21 _zyL74_pbcCapOut1
   ======================================================
   ```


© 2021 All Rights Reserved.

In the sample report shown above, the second level signals `_zyL86_pbcCapOut2`, `_zyL74_pbcCapOut1`, and so on, correspond to behavioral events on line 86 and 74, respectively.

If a behavioral cycle has entered an infinite loop, the hardware run will hang with no time advance. You can debug the infinite loop by checking the `tmp/xc_run.log` file for the "Behavior evals statistics" section. The 2nd column indicates whether or not a particular behavior processor has entered an infinite loop.

Identifying the Source of Behavioral Evaluations

The following are the steps to identify the source of behavioral evaluations:

1. From the `"Behavioral evals statistics"` in `tmp/xc_run.log`, identify the module instance that incurs large number of behavioral evaluations.
2. Identify the module corresponding to the module instance with large `evals`.
3. Look up the module in `xc_work/bc.log` to find a list of line numbers of behavioral processes. For example:

   ```
   Beh Procs found at lines 66, 90
   ```

   Make sure the listed `always` procedure is behavioral and cannot be rewritten as RTL to improve performance.
4. When necessary, compile with the `+bcDebug` option of the `ixcom` command to get detailed statistics for individual behavioral events for each module instance. This helps in isolating the source of behavioral evaluations.

© 2021 All Rights Reserved.

System Task Mapping in IXCOM flow

In the IXCOM mode, the behavior of system tasks can be emulated by issuing testbench callback to map the system tasks. Mapping of general system tasks or functions in IXCOM mode is described in the following sections:

- Using $ixc_ctrl Pragma within the Module
- Using ixcom Command-Line Option to Map System-Task Calls

Using $ixc_ctrl Pragma within the Module

To map a system task in a DUT module, write an `ixc_ctrl` pragma in an `initial` statement in that DUT module, as follows:

```
$ixc_ctrl("tb_import", "<system_task_function_name>"[, "tb_mem"]);
```

Here, "`tb_mem`" is an optional argument to denote that the input/output arguments are passed through the memory buffer.

Using ixcom Command-Line Option to Map System-Task Calls

Use the following option of the `ixcom` command to map specific system-task calls in specific modules:

```
+tb_import_systf+[<mod_name>.]<sys_tf_name>[:mem][+...]
```

The above syntax specifies the system task call `<sys_tf_name>` in DUT module `<mod_name>` to get mapped in blocking mode.

Note: The `<sys_tf_name>` does not have the `$` symbol as a prefix to its name. Without `<mod_name>`, all the system-task or -function calls of `<sys_tf_name>` in all DUT modules would be mapped. The optional argument of `:mem` denotes that the mapping uses the memory buffer to transfer input and output. Otherwise, primary I/O resources are used, by default.

For example, `+tb_import_systf+dut.display:mem` denotes mapping of `$display` function in DUT module `dut` using memory buffer to transfer the input arguments. This is equivalent to adding the following `initial` statement to module `dut`:

```
initial $ixc_ctrl("tb_import", "$display", "tb_mem");
```

No change needs to be made to the RTL source code, if the command-line option is used.

© 2021 All Rights Reserved.

Mapping of $display-Family Tasks

The `$display` task is an important system task call, which is widely used in RTL code. The IXCOM mode provides the following mapping methods to map the `$display`-family system tasks, including:

- `$display`
- `$displayb`
- `$displayh`
- `$displayo`
- `$write`
- `$writeb`
- `$writeh`
- `$writeo`
- `$fdisplay`
- `$fdisplayb`
- `$fdisplayh`
- `$fdisplayo`
- `$fwrite`
- `$fwriteb`
- `$fwriteh`
- `$fwriteo`
- `$fopen`

  Note: For using `$fopen` in hardware:

  - Either, add the following in the module:

    ```
    initial $ixc_ctrl("tb_import", "$fopen")
    ```

  - Or, include the `+tb_import_systf+fopen` option with the `ixcom` command, and disable RTL-subset checking on the specific module as follows:

    ```
    ixcom +no_rtlchk+<mod> +tb_import_systf+fopen
    ```

- `$fclose`

© 2021 All Rights Reserved.

- `$fflush`
- `$error`
- `$warning`
- `$info`
- `$fatal`

The following mapping methods are sorted by descending precedence and ascending performance or speed.

1. `$ixc_ctrl("tb_import", "<sys_task_name>"[, "tb_mem"])`

   Put a `tb_import` pragma in an `initial` statement in a DUT module, and map the specific `$display` task calls as `tb_import` task calls. The optional argument specifies using the memory buffer for argument transfer.

2. `+iscdisp[+[<dut_mod_name>[+...]]]`

   Use this command-line option to map all `$display`-family task calls in the specified DUT modules, in blocking mode and using primary I/O resources for argument transfer. Without the specification of <*dut_mod_name*>, all `$display`-family task calls in all DUT modules would get transformed.

3. `+gfifoDisp+<dut_mod_name>[+...]`

   Use this command-line option to map all `$display`-family task calls in a special fast non-blocking mode using memory buffers. The `+gfifoDisp` option preserves the order of `$display` statements.

The mapping method of higher precedence overrides the one of lower precedence. For example, `$display` is mapped in blocking mode if there is an `$ixc_ctrl("tb_import", "$display")` pragma in the module `dut`, even if the `+gfifoDisp+dut` option of the `ixcom` command is used.

Caution

Note that `$error`***,*** `$warning`***,*** `$info`***,*** `$fatal` ***are not supported with the*** `+gfifoDisp` ***option.***

Support for System Tasks at Elaboration Time

IXCOM supports `$fatal`, `$error`, `$warning`, and `$info` system tasks at elaboration in module and generate scopes as per SystemVerilog 2009 LRM Section 20.10 as follows:

© 2021 All Rights Reserved.

- `$fatal` aborts the elaboration.
- `$error` generates the approrpiate message and stops the elaboration.
- `$warning` and `$info` messages appear during IXCOM compilation, and the elaboration and compilation proceed normally.

Note: These elaboration system tasks (`$warning`, `$info`, `$error`, and `$fatal`) are handled by IXCOM, and not passed to ***xrun***. Note that IXCOM does not evaluate the messages as in `$display`. It only detects the presence of the system task and issues the fatal, error, warning, or note accordingly.

Using System Tasks and Procedures at Run time

Many IEEE-standard system tasks can be added to the Verilog code and, in most cases entered at the Command-Line Interface (CLI). In addition, many equivalent VHDL procedures can be added to the VHDL code.

- Other system tasks can be entered using the `$xc_cmd()` Verilog system task, which lets you execute a CLI command from your source code. Refer to the `$xc_cmd();` section in the *Commands and System Tasks for SA Mode* chapter of the *VXE Command Reference Manual*.
- If your design contains system tasks, you can get a syntax error when compiling with another simulator. To avoid this problem, surround such code with the `IXCOM_COMPILE` compiler directive. The `IXCOM_COMPILE` compiler directive is automatically recognized by the IXCOM compiler.

Supported Verilog System Tasks and VHDL Procedures

To use the following Verilog system tasks and VHDL procedures:

- Add the system task directly to the Verilog code
- Add the procedure directly to the VHDL code
- Enter the system task at the CLI (Command-Line Interface)

To use the VHDL procedures, you must add the following library clause to your code:

```
library axis;
use axis.axis.all;
```

Note: Instance paths to signals or Verilog primitives are specified using period delimiters, that is:

```
instance.instance.signal_or_primitive
```


© 2021 All Rights Reserved.

Executing Non-Synthesizable Code in the Emulator

The majority of typical testbench code involves the monitoring and manipulation of signals in the Device Under Test (DUT) to direct simulation or to detect error conditions occurring during simulation. This code is usually non-synthesizable and typically represents the largest performance bottleneck in simulation. There are several options to deal with this type of monitor code:

- Exclude the code completely from your simulation. This option provides the fastest performance, but sacrifices some debugging capabilities.
- Use the Testbench Callback technology to automatically execute the non-synthesizable code in the emulator hardware. Refer to Compiling DUTs with Non-Synthesizable Code by Using Testbench Callback Technology on page 768 for more detail.
- Partition the monitor code into a synthesizable error-condition check and a non-synthesizable code segment that executes when the error is detected. This technique is described later in this section.
- Set the monitor code to be active in software simulation mode only. The design is swapped to software simulation time at specific time intervals, or after specific error conditions are detected. This technique is described later in this section.

Activating Monitor Code Only in Software Simulation

There are two techniques for making monitor code active only in the software simulation mode:

- Marking Monitor Code to Simulate Only in Software Simulation
- Swapping to Software Simulation Mode at the Appropriate Time

Marking Monitor Code to Simulate Only in Software Simulation

IXCOM simulator lets you *mark* either a whole module in the DUT, or just a portion of it, to be active in software simulation mode only; in software acceleration mode the code is totally inactive and has no affect. After you swap to software simulation mode, the code becomes active again from the next simulation time-tick. This feature is especially useful for minimizing the impact of monitor code in testbenches. There are two ways to mark modules in this fashion:

- To *mark* an entire module to simulate while in software simulation mode only, specify the `+dutignore+<module_name>` compiler option. All instances of the module *and its sub-hierarchy* will not simulate at all while running in simulation acceleration mode.

© 2021 All Rights Reserved.

- To *mark* just a portion of a module to be simulated while in software simulation mode, use the `$axis_rccoff` system task and the `//ixc rtl_off` pragma. The `$axis_rccoff` task takes no arguments, returning a boolean `true` if the design is currently running in software simulation mode or `false` if running in co-emulation mode.

  Here is an example:

```
always @(posedge error_flag)
begin
    if ($axis_rccoff)
   // code executed only if design running in software simulation mode
end
endmodule
```

  Note: This system task is non-synthesizable and thus cannot be used in a DUT module.

Swapping to Software Simulation Mode at the Appropriate Time

To simulate the monitor code, you must initiate the swap from simulation acceleration mode to software simulation mode:

- You can control the swapping feature at the command line, using the commands described in the SA Run-Time and Debug Commands chapter of the *VXE Command Reference Manual*. If you want to break on a signal in simulation acceleration mode, that signal must be exported at compile time, as previously explained.

- *Verilog testbench*: Use the `$xc_cmd` system task to execute a CLI command directly from your testbench, for example:

```
module testbench ();
   ...
   always @(posedge sys_clk)
   begin
      if (packet_status==3'b0 && !packet_received)
      $xc_cmd("verbose 2");     // enable xc verbose messages
   end
endmodule
```

Note: The following commands are not supported through the `$xc_cmd` system task if you are using the xeDebug tool:
`on`, `off`, `stop`, `run`, `tbrun`, `autorun`, `nbrun`, `match`, `bmatch`

Monitoring Signals and Displaying Messages

The `$display` and `$monitor` system tasks, which can be used in the testbench to display message strings and signal values during simulation, are fully supported in the software simulator and produce expected results if used in non-accelerated modules.

© 2021 All Rights Reserved.

However, if used in a DUT module, by default, these tasks will not work as expected while running the design in simulation acceleration modes, because they are not synthesizable and are thus ignored by the emulator hardware. If you are trying to add a testbench module to the DUT so it can be accelerated, this can be a problem, for example:

```
module dut (..);
always @(posedge sys_clk)
begin
    if (start_mem_cycle && write_en && read_en)
    $display("Invalid memory cycle at time %t", $time);
end
```

In this example, the `$display` system task is ignored if running the design in simulation acceleration mode.

Any of the preceding techniques for speeding up monitor code can also be used to display these messages. If using the partitioning technique, you would modify the previous example, as follows:

```
module dut (..);
reg dummy_reg;
always @(posedge sys_clk)
begin
    if (start_mem_cycle && write_en && read_en)
        dummy_reg = 1'b1; //register toggles when error condition detected
    else
        dummy_reg = 1'b0;
end
```

Now add the following code to a testbench module:

```
always @(posedge testbench.dut.dummy_reg)
    $display("Invalid memory cycle at time %t", $time);
```

In the previous example, only the `$display` message itself is simulated in software simulation; actual error checking is accelerated by the emulator hardware.

Compiling DUTs with Non-Synthesizable Code by Using Testbench Callback Technology

The Testbench Callback technology lets you embed non-synthesizable code in a DUT or call a task containing non-synthesizable code from a DUT design unit. The software simulator automatically executes this code while running.

The unique Testbench Callback technology lets you enable other types of non-synthesizable code, as well; this code is then executed.

Testbench Callback technology gives you the ability to:

© 2021 All Rights Reserved.

- Improve the performance of designs with testbench code, such as bus protocol monitors and code that detects violations or fatal conditions.
- Display messages while running at hardware speeds.
- Execute non-synthesizable code, such as PLI C/C++ routines, Operating System calls, or other testbench code, when certain DUT conditions occur.
- Inject data into, and extract data from, your design at high speed, using MARG functions.
- Load new data files, load memories, or dump contents of memories, and so on, on demand.
- Control (stop, start, and so on) Palladium Z1 Engine.
- Report assertion violations.

There are two ways to use Testbench Callbacks in Verilog designs:

- Testbench Callback Automated Method: In the automated method of using Testbench Callbacks, blocks of non-synthesizable code in the DUT are enabled using a special pragma. No other intervention is required, but some restrictions apply.
- Testbench Callback Primitive (axis_tbcall): The Testbench Callback primitive (`axis_tbcall`) lets you directly call a user-written task containing non-synthesizable code when a particular condition occurs. This method provides additional flexibility over the automated method and gives you greater control.

Testbench Callback Automated Method

Use IXCOM compiler to mark a region of statements inside a Verilog module that has been marked for mapping to hardware (under DUT hierarchy by using the `+dut` flag) to be executed in software. These set of statements are enclosed within a `begin..end` block. This enables a very flexible way to execute a monitoring or error-reporting code, which otherwise would not be mapped to hardware. This feature is provided through the Testbench Callback automated method.

The automated method of using Testbench Callbacks lets you enable non-synthesizable code, such as `$display` statements embedded in a DUT module, typically located within the `//synopsys translate_off` blocks. To enable such code, add the following comment pragma to a `begin` statement in your `always` procedure:

```
// ixc tbcall_region
```

The code enclosed by the commented `begin` statement to the next `end` statement will not be checked by the Synthesis Policy Checker and will be automatically executed by the

© 2021 All Rights Reserved.

software simulator at the appropriate time. No user intervention is required. The comment pragma is ignored by all other tools.

This is illustrated by the following example.

```
module DFF_1(clk, D, Q,)
  input clk;
  input [3:0] D;
  output [3:0] Q;
  reg [3:0] Q;
    always @(posedge clk) begin
        Q <= D;
        begin //ixc tbcall_region
            $display("Q= %d", $time, Q);
        end
    end
endmodule
```

In this example, the block of code specified within the `tbcall_region` pragma is scheduled to execute after the non-blocking assignment in the simulator scheduling queue. This is done to simulate the behavior of the hardware. The `$display` statement should display the old value of signal, *`Q`*. However, in this example, the `tbcall_region` is executed after the non-blocking assignment, and the new value of signal `Q` is displayed in both the hardware and the software modes, when the `+dut` switch of the `ixcom` command are used. This happens when the signal value used, is assigned using the non-blocking assignment before the `ixc tbcall_region` pragma.

To see the *old values* of VHDL signals and of Verilog signals that are the target of non-blocking assignments within the same `always` procedure, use the `+tbcnba` switch. With the `+tbcnba` switch, IXCOM analyzes the dependency of the signals read in the `ixc tbcall_region` pragma and are assigned using the non-blocking assignment. IXCOM, then, generates the old values of such signals to be used in the `ixc tbcall_region` pragma. However, this applies to signals that are written and read within the same `always` procedure. If the signals are written in a different `always` procedure and read within the `ixc tbcall_region` pragma, which is within another `always` procedure, new values of these signals are used.

VHDL Example

In this example, at the rising edge of `clk`, `writeline` will display the new value of *`Q_tmp`*.

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_textio.all;
library axis;
use axis.axis.axis;
library std;
use std.textio.all;
entity DFF_1 is
port (
```


© 2021 All Rights Reserved.

```
D : in std_logic_vector(3 downto 0);
clk : in std_logic;
Q : out std_logic_vector(3 downto 0 )
);
end DFF_1;

architecture DFF of DFF_1 is
signal Q_tmp : std_logic_vector(3 downto 0 );
    begin
        a1: process(clk)
        begin
          if(clk'event and clk = '1') then
            Q_tmp <= D;
          endif;
        end process a1;
        Q <= Q_tmp;

        a2 : process (clk)
          variable l :line;
        begin
           if (clk'event and clk = '1') then
             -- ixc tbcall_region_on
             write(l, NOW);
             write(l, Q_tmp);
             writeline(output, l);
             -- ixc tbcall_region_off
           endif;
        end process a2;
end DFF;
```

Verilog Example

In this example, at the rising edge of `clk`, `$display` will display the new value of *`Q`*.

```
module DFF_1 (clk, D, Q);
input clk;
input [3:0] D;
output [3:0] Q;
reg [3:0] Q;
always @(posedge clk) begin
    Q <= D;
end

always @(posedge clk) begin //ixc tbcall_region
    $display("Q= %d", $time, Q);
end

endmodule
```

Restrictions and Usage Suggestions for Testbench Callback Automated Method

Please be aware of the following restrictions and usage suggestions for the automated method of using Testbench Callbacks:

- The pragma can be added only to `always` procedures, not tasks or functions. However, you can call a task or function from the pragma code body.

© 2021 All Rights Reserved.

- Additional restrictions apply to the use of delayed timing events and cross-module references within the pragma code body (Refer to Restrictions on Testbench Callback Primitive (axis_tbcall) on page 773).

- Remember that all code within the comment pragma block is executed by the software simulator at a slower speed. For best performance, minimize the code within these pragmas. For example, if you have a conditional statement that checks for a particular condition, put the `ixc tbcall_region` pragma after the condition-check code. That way the check is performed in the hardware at full speed and only switches to the software simulator if the condition actually occurs, for example:

```
always @(posedge clk)      // condition checked in hardware
begin
    if (AddrW == AddrR)    // condition checked in hardware
    begin       // ixc tbcall_region switches to software simulator
        $display(" Write through operation not allowed!\n");
        $display(" Write  Addr, Read Addr = %h", AddrW);
    end        // switches back to Palladium Z1 here
end
```

  For additional usage suggestions, refer to Usage for Testbench Callback Primitive (axis_tbcall) on page 775.

- When two `always` processes wake up at a clock edge, the correct behavior is to display the old value of the signal, that is, the value of the signal before it is updated.

  For example, in the following code, the behavior should be to display the old value of signal `a` before it is incremented by 1:

```
always @(posedge clk)
begin
     a <= a + 1;
end

always @(posedge clk)
begin // ixc tbcall_region
     $display(a);
end
```

  However, by default, when you use the `ixc tbcall_region` pragma, the updated value of the signal is displayed. In the above code, the value of signal `a` passed to the `$display` statement is read from the software side. By this time, the value of `a` is already updated, which is incorrect. To ensure that the generated code reads the value of the variables before these are updated, use the `+tbcnba` option of the `ixcom` command. The `+tbcnba` option enables reading of variables that can be assigned in a non-blocking way from outside of the process.

Testbench Callback Primitive (axis_tbcall)

IXCOM flow provides a special primitive, `axis_tbcall`, that lets you call a Verilog task while running in emulation modes. The task typically contains non-synthesizable code and is

© 2021 All Rights Reserved.

automatically executed in the software simulator. This mechanism enables you to perform operations not typically allowed in the DUT, such as testbench code, PLI C/C++ code, file I/O, display messages, and so on.

To call a testbench task from a Verilog module, call the built-in primitive `axis_tbcall` from within a DUT design unit, using the following syntax:

```
axis_tbcall(callback_signal, "callback_task");
```

The *callback_signal* is a 1-bit wire declared in the DUT module—ports cannot be used for this purpose. The *callback_task* is a local Verilog task with the testbench code to execute. To cause the *callback_task* to be called, create a transition from 0 to 1 on the *callback_signal*.

Restrictions on Testbench Callback Primitive (axis_tbcall)

The following restrictions apply when using the `axis_tbcall` primitive:

- The `callback_signal` should be triggered only by a sequential `always` procedure. Combinational blocks can have unexpected results like glitches. For example:

```
reg callback_trig, A, B;
always @(A or B)    // combinational: NOT recommended for Callback
begin
    if (A > B) callback_trig <= 1'b1;
    else callback_trig <= 1'b0;
end
axis_tbcall(callback_trig, "callback_task");
reg callback_trig, A, B;
always @(posedge clock)   // sequential: Recommended for Callback
begin
    if (A > B) callback_trig <= 1'b1;
    else callback_trig <= 1'b0;
end
```

- Statements that generate delayed simulation events such as `wait`, `@event`, and delay assignments
  (`A <= #1 B;`) can be executed only after swapping into software simulation mode (refer to Controlling Palladium Z1 from within Testbench Callback Tasks on page 776). You should not use `forever` statements since these will prevent you from swapping back to emulation modes.

- The *callback_task* can call another task only if that task is declared in a testbench or other module running in the software simulator.

- Only the following signals can be referenced by the *callback_task*:

  - Nets or registers declared in a DUT module can be referenced only if they are read. You can write to signals that represent edge-triggered flip-flop registers using the

© 2021 All Rights Reserved.

`$axis_rwrite` task (refer to Writing to DUT Signals Using Testbench Callback Tasks on page 776).

Note: Reading signals creates special Palladium Z1 I/Os, which impact performance.

- Remote signals using cross-module references can be referenced.

  Note: Reading signals in the DUT creates special Palladium Z1 I/Os, which impact performance.

- Variables declared locally in the task can be read or written without restriction. You can reference variables in other tasks if the remote tasks are located in the testbench or other modules running in the software simulator.

  Note: No Palladium Z1 I/Os are created.

- Memories mapped to the emulator memories can be read or written using MARG functions.

- If you read a DUT signal in an `axis_tbcall`, you will get the value at the end of current timestep, that is, after flip-flops have been updated, for example:

```
integer bcount;
    always @(posedge clk)
    begin
        bcount <= bcount + 1;        // bcount incremented to 1
        d_bcount <= bcount;          // change in d_bcount calls
                                     // Callback Routine
    end
    axis_tbcall(d_bcount, "my_cb");
    task my_cb; // called when d_bcount changes from 0 to 1
    begin
        $display("bcount:%b", bcount);
                                // displays updated value(1)
        $display("d_bcount:%b", d_bcount);
                            // displays updated flip-flop value
    end
    endtask
```

- Some mismatches between the software and hardware runs of design can be observed when sampled value functions are used in continuous assignments, before calling the `axis_tbcall` primitive.

Example of Testbench Callback Primitive (axis_tbcall)

This example shows how to use `axis_tbcall` to display text from the DUT during emulation modes.

```
‘timescale 1ns/100ps
```


© 2021 All Rights Reserved.

```
module dut;
  reg [63:0]  dL;
  reg [6:0]   dS;
  reg [1:0]  cnt;
  reg        enE;
  wire       enP;
  wire       clk;
  axis_clkgen #(1, 20) (clk);

  always@(posedge clk) begin
    dL <= dL + 64'h0001000100010001;
    dS <= dS + 1;
    cnt <= cnt + 1;
    if(cnt == 0) begin
       enE <= ~enE;
    end
  end

  initial begin
    dL = 0;
    dS = 0;
    enE = 0;
    cnt = 0;
    $axis_set_clkgen("clk", 0, 5, 10, 10);
  end

  ixc_pulse(enP, enE);

  axis_tbcall(enP, "t1");

  task t1;
    reg [32:0] t;
    begin
        t = $time;
        $display(t, "L: %h", dL);
        $display(t, "S: %h", dS);
        if(t > 1024)
           $finish;
    end
endtask

endmodule
```

Usage for Testbench Callback Primitive (axis_tbcall)

Here are some usage suggestions for the Testbench Callback primitive (`axis_tbcall`):

- As indicated earlier, the Testbench Callback task runs on the software simulator at a slower speed. For best performance, minimize both the amount of code in the task as well as the number of times the task is called. The condition that triggers the Testbench Callback task should be checked in Palladium Z1 and should occur infrequently for best performance.
- The most efficient method for transferring data into and out of the Testbench Callback task is by instantiating a Verilog memory in the DUT and writing the desired data to it using synthesizable code. You can then read or write the contents of the memory in the Testbench Callback task using MARG functions.

© 2021 All Rights Reserved.

Writing to DUT Signals Using Testbench Callback Tasks

Testbench Callback tasks can be used to write to Verilog registers *that are edge-triggered flip-flops* in the DUT using the `$axis_rwrite` system task, for example:

```
$axis_rwrite(tb.dut.reg, 1);
```

However, to avoid unexpected results, the registers and signals should not be written to in the same timestep in which the DUT itself writes them. For example, if a signal is updated on the negative-edge of a clock signal in the DUT, the testbench callback task should write to it on the positive-edge.

Controlling Palladium Z1 from within Testbench Callback Tasks

The following commands enable you to execute most CLI commands from within your Testbench Callback task. The following commands are most commonly used for controlling Palladium Z1:

- `xc off`

  This command first swaps the design at the end of the current timestep to software simulation mode from either emulation mode, then continues running in software simulation mode. This swap causes ignored DUT design units (those specified with the `+dutignore+<module_name>` option) and code within `translate_off` blocks to become active and begin executing. You must execute this command in the Testbench Callback before executing any `wait`, `@event`, or assignments with delays.

- `xc stop`

  This command swaps the design into software simulation mode and stops the running of the design at the end of the current timestep. You are also returned to the CLI.

- `xc on`

  This command swaps the design immediately to emulation mode from software simulation mode. The DUT is downloaded into Palladium Z1, which continues running in an emulation mode. You must specify this command at the bottom of your Testbench Callback if you used either the `xc stop` or `xc off` commands earlier.

- `xc autorun`

  This command swaps the design immediately to emulation mode from software simulation mode. The DUT is downloaded into Palladium Z1, which continues running in emulation mode. You must specify this command at the bottom of your Testbench Callback if you used either the `xc stop` or `xc off` commands earlier.

© 2021 All Rights Reserved.

Assertion-Style Callbacks

If your design uses many assertion-style checkers that produce messages only when an error condition occurs (infrequently), you can get better performance if you use the assertion-style version of the Testbench Callback primitive: pass a `#(1)` parameter to the regular primitive:

```
axis_tbcall #(1) (callback_signal, "callback_task");
```

Assertion-style primitives work the same way as regular primitives except they do not create Palladium Z1 I/Os, so you should not use them for displaying DUT signal values because they will be incorrect. To read these signals, first swap to software simulation mode using `xc off`.

```
axis_tbcall #(1) (assert_signal, "assert_task");
task assert_task;
   begin
      $display("design data %h", design_data);
   end
```

Forcing Signals from Hardware

When a hardware signal is forced from the hardware partition, the IXCOM compiler performs the required transformations using the `ixc_rforce` and `ixc_wforce` force primitives. The `ixc_rforce` is used to force register type of signals and `ixc_wforce` is used to force wire type of signals.

Keep the following considerations and limitations in mind while forcing the signal from hardware:

- Only `struct` and array of scalar types are supported. All combinations of array `struct` such that the basic element is a scalar are supported.
- Forcing signals from the testbench to the DUT is not supported.
- Force on signals that are synthesized as memory or are forced to be memory by using pragma is not supported.
- If a signal is forced in a process, it cannot be read in the same process. The signal can be read from other processes without any issues.
- A force statement can be replaced by the force primitive except if there are multiple forces on the same signal across the design. If a signal is forced from anywhere in the design, the other forces need to be disabled.
- Hardware force primitives only accept packed signal.
- Forcing a signal with scope that is local to a function is not supported.

© 2021 All Rights Reserved.

To optimize the force instrumentation in cases such as the following where the bind primitive is used for force, specify the `+frc2bind` option so that the force of this kind in the `initial` block is converted to a simple `bind` primitive.

```
initial
force signal_one = signal_two;
```

This has the same effect of binding `signal_one` to `signal_two`.

The synchronization mechanism is an elaborate setup and might consume a considerable amount of resources. If you know that there is only one force on a signal and do not wish to generate the synchronization signal, then specify the `+light_frc+<mod_name>` option. This skips the generation of the synchronization circuit.

This option might create problems if the signal is forced from multiple locations as a new force does not cause old force to be released in which case the behavior is undefined. However, if there is only one force on the signal in the whole design, this option can result in significant saving of the resources used in the emulator.

Compiling Designs in Stages using IXCOM Compiler

The IXCOM compilation process comprises the following compilation stages (or steps):

- Design analysis
- Design elaboration
- Generation of software and hardware models
- Compilation of the design for software simulation using Xcelium
- Compilation of the design for the Palladium Z1 hardware
  - Compilation using HDL-ICE Compiler
  - Compilation using the xeCompile Compiler

You can control the IXCOM compilation process to start from, stop at, or rerun from a specific compilation stage by using the `-target <stage>` option of the `ixcom` command. To control the compilation, use the various values for the `<stage>` argument of the `-target` option as described in the following sections.

The `<stage>` argument enables you to compile the design until a specified stage or just rerun the specified stage. You can specify the `<stage>` values using a minimum number of case-insensitive characters.

© 2021 All Rights Reserved.

Following are the values that you can use for the <*stage*> argument:

<*stage*> value	**Function**
swmodel	Compiles the software model for SA by using the Xcelium tools.
ixvcg	Creates IXCOM-generated HDL for HDL-ICE and Xcelium. Note that this neither option invokes HDL-ICE nor Xcelium compiler to compile the design.
lwdmodel	Invokes the created LWD snapshot database. It is equivalent to the following make command in which <*LWD_database_directory_path*> is xc_work/lwd_work: `make -f <LWD_database_directory_path>/lwd_make` Refer to Generating Light Weight Debug Database (LWD) from IXCOM for details about LWD snapshot.
hdlice	Synthesizes the hardware model for mapping to the hardware emulator by using the HDL-ICE Compiler.
postHDLICE	Updates the design hierarchy information. **Note:** This stage is required only if the use model requires external netlists to be imported.
xecompile	Maps the design to the Palladium Z1 hardware by using the xeCompile compiler.
postXECOMPILE	Updates the design partition information for the external netlist. **Note:** This stage is required only if external netlists need to be imported.
xecompile:designImport	Runs only the designImport step of xeCompile.
xecompile:skipImport	Runs xeCompile, skipping the designImport step.
xecompile:precompile	Runs compilation only until precompile.
xecompile:et3pass1	Runs precompile, if necessary, and starts PPC compileFind until et3pass1 is done.
xecompile:et3pass2	Skips the first part of the flow and starts from et3pass2.

© 2021 All Rights Reserved.

Elaboration and Non-Elaboration Contexts

This document uses two terms to define the compilation process:

- Elaboration context refers to the compilation use model in which you include the source file(s) or `-top` <*du*> <*options*> option with the `ixcom` command on the command line.
- Non-elaboration context refers to the compilation use model in which you invoke the `ixcom` command to rerun a compile stage on the pre-elaborated design by using the `-target` <*stage*> option.

Creation for IXCOM-generated HDL (-target ixvcg)

The `-target ixvcg` option analyzes Verilog and/or VHDL source files, transforms the design into RTL, and generates the modified source files for compiling the hardware model and software model.

For example, the following command analyzes source files and elaborates the design:

```
ixcom -target ixvcg -ua +dut+Dut tb.sv dut.sv
```

Similarly, the following command elaborates the design:

```
ixcom -target ixvcg -ua +dut+Dut -top Tb
```

In the elaboration context, this use model includes the following compilation stages:

- Elaboration
- Generation of software and hardware models

In a single-step IXCOM compilation flow, you must analyze the source and elaborate the design again to regenerate IXCOM-generated HDL. In a multi-step IXCOM compilation, involving mixed-language compilation, for example, you must elaborate the design to regenerate IXOM-generated HDL.

Running the `ixcom -target ixvcg` command alone, is not supported. To run the design on hardware, you must invoke the software model stage by using the `-target swmodel` option of the `ixcom` command to complete the Xcelium software compilation stage. Additionally, invoke the `ixcom -target hdlice` and `ixcom -target xecompile` commands separately to complete the hardware-compilation stages.

Software Compilation Stage (-target swmodel)

In the elaboration context, this use model includes the following compile stages:

© 2021 All Rights Reserved.

1. Elaboration
2. Generation of software and hardware models
3. Xcelium compilation and elaboration to generate simulation snapshot

In the non-elaboration context, only the Xcelium compilation, which is listed as step 3 above, is performed.

Note: To run the design on hardware, you must invoke the `ixcom -target hdlice` and `ixcom -target xecompile` commands separately.

Important

For compiling designs with the imported external netlist files (See Importing External Netlists in the SA Mode), keep in mind that all the hardware-compilation stages must be completed prior to using the `-target swmodel` option. The hardware-compilation stages include the additional stages as described in the Importing Netlists Using the XEL Scripts for ICE-Mode Compilation section.

Examples

- The following command analyzes source files, elaborates the design, and invokes Xcelium to generate the simulation snapshot:

```
ixcom -target swmodel -ua +dut+Dut tb.sv dut.sv
```

- The following command elaborates design and invokes Xcelium to generate the simulation snapshot:

```
ixcom -target swmodel -ua +dut+Dut -top Tb
```

- The following command invokes Xcelium to generate the simulation snapshot:

```
ixcom -target swmodel
```

Hardware Compilation Using HDL-ICE Compiler Stage (-target hdlice)

In the elaboration context, this stage includes the following steps:

1. Elaboration
2. Generation of software and hardware models
3. Xcelium compilation and elaboration to generate simulation snapshot
4. Compilation of the generated hardware model by using HDL-ICE Compiler

© 2021 All Rights Reserved.

5. Update of design information for the external netlist post the HDL-Compiler stage. Note that this is applicable only if external netlists need to be imported.

In the non-elaboration context, this use model compiles the design using HDL-ICE Compiler, which is listed as step 4 above. In this context, all other options specified with the `ixcom` command are ignored. If you need to specify any additional compilation options for the design, you must re-elaborate the design.

NOTE: To run the design on hardware, you must issue the `ixcom -target xecompile` command separately.

Examples

- The following command analyzes source files, elaborates the design, invokes Xcelium to generate the simulation snapshot, and invokes HDL-ICE Compiler to compile the hardware model:

  ```
  ixcom -target hdlice -ua +dut+Dut tb.sv dut.sv
  ```

- The following command elaborates the design, invokes Xcelium to generate the simulation snapshot, and invokes HDL-ICE Compiler to compile the hardware model:

  ```
  ixcom -target hdlice -ua +dut+Dut -top Tb
  ```

- The following command invokes HDL-ICE Compiler to compile the hardware model:

  ```
  ixcom -target hdlice
  ```

For compiling external netlists, the following command invokes HDL-ICE Compiler to separately update the design-hierarchy information:

```
ixcom -target postHDLICE
```

In the elaboration context, running the compilation for the `postHDLICE` stage has the same effect as the `hdlice` stage.

You do not need to invoke the `postHDLICE` stage if you have invoked the `hdlice` stage:

```
ixcom ... -target hdlice
```

Note: The compiler reports an error if you attempt to compile the design for hardware without the required hardware-compilation options like the `-ua` and `+dut` options of the `ixcom` command.

Hardware Compilation Stage Using xeCompile (-target xeCompile)

In the elaboration context, this option has no effect. In the non-elaboration context, this option invokes the xeCompile compiler to map the design to hardware, and all options specified with the `ixcom` command are ignored except for the following:

© 2021 All Rights Reserved.

```
-xeCompile [arg]
-xeCompileArgs [arg]
```

Note: The compiler saves the xeCompile-specific options that are specified with the `ixcom` command for elaborating the design, to the compiler database. These options are read and passed to the xeCompile stage in the above invocation.

You must re-elaborate the design if additional compile options have to be applied to the design.

For external-netlist compilation, this stage also includes the design partition information update.

Examples

Use the following command to rerun the `xeCompile` stage:

```
ixcom -target xecompile
```

For compiling external netlists, design-partition information can be updated separately by using the following command:

```
ixcom -target postXECOMPILE
```

The `postXECOMPILE` stage is not required if you have used the `xeCompile` stage with the `-target` option. In the elaboration context, using the `postXECOMPILE` option has the same effect as using the `xeCompile` option.

Use the following command to run only the `designImport` step of the xeCompile stage:

```
ixcom -target xecompile:designImport
```

Use the following command to run the xeCompile stage, skipping the `designImport` step at the beginning:

```
ixcom -target xecompile:skipImport
```

Note: If you try to run the `xeCompile` stage, and the `hdlice` compilation stage has not been completed successfully, compilation is aborted with an error message.

One of the ways to invoke xeCompile from IXCOM is by including customized scripts for the `xeCompile` options and settings and specify these in a file named `compilerOptions.qel`, which you pass to the `ixcom` command. You can edit parts of this file between invocations of the `ixcom -target xecompile:<xxx>` command without needing script customization.

This improves the efficiency of the iterations and eliminates the need for any redundant tasks, such as executing the `precompile` command or the `et3pass1` compilation.

© 2021 All Rights Reserved.

The `compileFind` command helps find the best compilation trade-off when the Parallel Partition Compiler (PPC) is in use. This allows a partial compilation to be done, stopping the flow at critical points to let you make some flow adjustments. Refer to the Methods to Optimize Design and Improve Performance When Using Parallel Partition Compiler (PPC) for details on using PPC.

Use the following targets with the `ixcom` command to separate the invocations of xeCompile and ensure that compilation pauses for you to perform the required tasks in between the invocations:

```
ixcom -target xecompile:precompile ...
```

The `xecompile:precompile` option ensures that compilation runs only until `precompile`. Then, IXCOM compilation stops and returns the control to you.

Ensure that the `compilerOptions.qel` file includes the required settings for a successful PPC compilation (including `partitionGroups`, if necessary). Essentially, include the following option for the PPC flow in the `compilerOptions.qel` file:

```
compilerOption -add { distributedMulticore ON }
```

Use the following option to ensure that IXCOM compilation continues until `precompile`, if necessary, and starts PPC `compileFind` until `et3pass1` is done.

```
ixcom -target xecompile:et3pass1 ...
```

Using the above target instructs xeCompile to invoke the `compileFind` command instead of the `compile` command. Therefore, you must include the `compileFind` option in the `compilerOptions.qel` file through a TCL variable named `compileFindOptions`.

Note: The `et3pass1` and `et3pass2` targets will not work if `compilerOptions.qel` contains the `precompile`, `compile`, or `compileFind` commands, or if `compileScript` is set in the `compilerOptions.qel` file. Allow IXCOM to invoke the right commands when these targets are used.

For example, the following command instructs `xeCompile` to run `compileFind -et3pass1 -lsf`:

```
set compileFindOptions "-lsf"
```

The following command instructs IXCOM to skip `et3pass1` and start from `et3pass2`.

```
ixcom -target xecompile:et3pass2 ...
```

The `xecompile:et3pass2` target instructs xeCompile to invoke `compileFind` instead of the `compile` command. Therefore, you must include the `compileFind` option in the `compilerOptions.qel` file through a TCL variable named `compileFindOptions`.

For example, the following command instructs xeCompile to run: `compileFind -et3pass2 -max_trials 2 -lsf` command:

© 2021 All Rights Reserved.

```
set compileFindOptions "-max_trials 2 -lsf"
```

Here, the compiler assumes that `et3pass1` was run before the `compileFind` command, and the compiler aborts if `et3pass1` has not been run.

These targets facilitate you to try different PPC resource assignment strategies.

For instance, you can try the following command sequence:

```
ixcom -target xecompile:precompile
ixcom -target xecompile:et3pass1
```

Modify PPC partitions resource assignment in `compilerOptions.qel` and use the following command:

```
ixcom -target xecompile:et3pass1
```

If the results are good, you can finish `compileFind` with the following command:

```
ixcom -target xecompile:et3pass2
```

Note that it is not necessary to execute the following command first because the `et3pass1` target is able to run `precompile` automatically if required:

```
ixcom -target xecompile:precompile
```

However, none of these targets run `designImport`. So, `designImport` must be run prior to run these targets.

You can also choose to run a few trials first and if the initial results are good, run `et3pass2` again with more trials:

```
ixcom -target xecompile:et3pass1
```

Request `-max_trials 3` in `compilerOptions.qel`:

```
ixcom -target xecompile:et3pass2
```

If the results are reasonable, increase `-max_trials` to `20` in `compilerOptions.qel`.

```
ixcom -target xecompile:et3pass2
```

Tip

Another way of saving compilation time, is to skip the `dbgPrepare` step when you only want to identify the best trade-off or the best step-count. For this, use the `-noDbgPrepare` option of the `compileFind` command.
Note, however, that unless `dbgPrepare` is run subsequently, the final database will not be functional and it would not be possible to download it to an emulator host. So, in this case, when you are happy with the performances of `compileFind`, manually run `dbgPrepare` from `xeCompile` or the Linux shell.

© 2021 All Rights Reserved.

Generating Pulses

The following three primitives are provided for generating pulses and clocks:

- ixc_pulse (for Testbench Callback, Synchronous Elements)
- ixc_pulse#(0) (for Combinational Logic and Latches)
- ixc_pulse#(1) (for Palladium Z1 eval or Simulation Timestep Clock)

ixc_pulse (for Testbench Callback, Synchronous Elements)

The `ixc_pulse` primitive generates a pulse when an input *signal* changes. Here is how to instantiate it:

```
ixc_pulse (clk, signal);
```

This pulse goes from 0 to 1 when the signal changes, then back to 0 at the end of the simulation timestep or Palladium Z1 evaluation cycle.

This feature is useful for creating pseudo-timed events, such as creating a pulse on any edge of a clock. It is also used in conjunction with the Testbench Callback technology when you need a 0 to 1 transition to occur on the Testbench Callback signal.

Note: The output *clk* can be used only to directly drive synchronous elements, such as flip-flops or synchronous memories. Use only the *posedge* of the pulse signal *pclk* in the design, and never use the pulse value in combinational logic or latches.

ixc_pulse#(0) (for Combinational Logic and Latches)

The `ixc_pulse#(0)` primitive also generates a pulse (*ps*) when an input *signal* changes value. Here is how to instantiate it:

```
ixc_pulse#(0) (ps, signal);
```

However, this primitive has different delta cycle semantics: it changes from 0 to 1 and holds the value 1 to the last delta cycle, End-of-Time (EOT), of the current simulation time. Then it goes back to 0 immediately after EOT.

You can use this pulse signal in combinational logic and latches to do special modeling.

For example, consider the following flip-flop model:

```
always @(posedge c) q <= d;
```


© 2021 All Rights Reserved.

To guarantee deterministic results in parallel simulation, Palladium Z1 forces all flip-flops to latch the old value of d into q and does not enable special flip-flop modeling that could capture the new value of d on the posedge of c. But `ixc_pulse#(0)` provides a way to achieve this function, as follows:

```
ixc_pulse #(0) (pc, c);
always@(pc or c or d) if(pc & c) q = d;
```

In this way, you can build an equivalent flip-flop model that will latch in the new value of d.

Important

Do not use the edge of the `ixc_pulse#(0)` signal because edge detection logic in Palladium Z1 is based on the EOT value of the previous eval. Since the EOT value of this new pulse will be 1, not 0 as in the regular `ixc_pulse` model, the edge detector will fail if the input signal changes in every simulation time step. To use the positive edge of the pulse, use the regular `ixc_pulse` primitive to generate the pulse. And never use the negative edge of the pulse signal because it will not work in Palladium Z1.

ixc_pulse#(1) (for Palladium Z1 eval or Simulation Timestep Clock)

The `ixc_pulse#(1)` primitive generates a Palladium Z1 eval clock or the simulation timestep clock.

The only way to use this primitive is as follows:

```
wire VCC = 1'b1;
ixc_pulse#(1) (evalClk, VCC);
```

Again, use only the positive edge of *evalClk* and never use its value in logic or latches.

Note: `ixc_pulse (evalClk)` is no longer supported.

© 2021 All Rights Reserved.

Pragma Handling in IXCOM

By default, the code within the IXCOM-supported pragmas is excluded from synthesis when compiling for hardware, that is, the Simulation Acceleration mode, but retained when compiling for software simulation mode. You can modify this behavior by using the following options with the ixcom or vlan and vhan commands for translate_on, translate_off, and Synopsys full_case and parallel_case pragmas:

- Handling translate_on and translate_off Pragmas in IXCOM: Use the following options for translate_on and translate_off pragmas:
  - +rtlIgnPragma[+<*mod*>]: Using this option implies that the translate_off and translate_on pragmas are ignored in the design or the specified design units and the code within the pragma blocks is executed in both the hardware and software partitions. If the optional <*mod*> parameter is not specified, it implies that this option is valid for all the modules or design units in the design, otherwise, it implies that it is applicable only to the specified modules or design units.This option needs to be specified with the vlan and ixcom commands to take effect in the Mixed-language designs. For more information, refer to description of +rtlIgnPragma in the *Commands and System Tasks for SA Mode* chapter of the *VXE Command Reference Manual*.

    Note: +rtlCommentPragma[+<*mod*>]: Using this option implies that the pragma itself is honored and the code within the translate_on and translate_off pragmas is treated as comment and ignored in both hardware and software partitions. This option needs to be specified with the vlan, vhan, or ixcom command. If the optional <*mod*> parameter is not specified, it implies that this option is valid for all the modules or design units in the design, otherwise, it implies that it is applicable only to the specified modules or design units. For more information, refer to the description of +rtlCommentPragma in the *Commands and System Tasks for SA Mode* chapter of the *VXE Command Reference Manual*. Using this option helps in ignoring the unnecessary code, such as syntactically incorrect code, without modifying the source, provided the code is inside pragma. If this option is provided with the ixcom command, the code inside the pragma must be syntactically correct for the file to be parsed.

  - When you specify the +rtlCommentPragma+<*mod*> +rtlIgnPragma options with the vlan or ixcom command, IXCOM ignores the translate_off and translate_on pragma on all the modules in the design and honors the translate_off and translate_on pragma only for the module specified with the +rtlCommentPragma+<*mod*> option. IXCOM, as a rule, gives more preference to +rtlIgnPragma+<*mod*> and +rtlComentPragma+<*mod*> options over the generic +rtlIgnPragma and +rtlCommentPragma options that apply to the entire design.

© 2021 All Rights Reserved.

- Handling `full_case` and `parallel_case` Pragmas in IXCOM: Use the following option for Synopsys `full_case` and `paralle_case` pragmas:
  - `+rtlHonorCasePragma`: Using this option implies that the Synopsys `full_case` and `parallel_case` pragmas within a Verilog `case` statement are honored during synthesis.This option needs to be specified with the `vlan` and `ixcom` command. For more information, refer to the description of `+rtlHonorCasePragma` in the *Commands and System Tasks for SA Mode* chapter of the *VXE Command Reference Manual*.
    - `full_case` pragma: In a Verilog `case` statement, the case item expression might not cover all the possible values that the case expression can assume. In such a case, the RTL compiler creates additional logic to handle the missing conditions. Specifying the `full_case` pragma enables RTL compiler to generate more efficient logic by eliminating the logic for the missing case conditions. This might result in synthesis/simulation mismatch if case expression assumes a value that does not match with any case item expression.
    - `parallel_case` pragma: In a Verilog `case` statement, the case item expression can be a dynamic expression. In such cases, LRM requires that the case item expression be evaluated sequentially as more than one case item expression might match with the specified case expression. The statements in the first matching case item expression needs to be executed. For RTL compiler, this implies synthesizing the priority logic. Using `parallel_case` pragma instructs the RTL compiler that the case item expression are all mutually exclusive and there will *only* be one match. RTL compiler can eliminate the priority logic and synthesize a more efficient logic. This, can, however result in synthesis/simulation mismatch if more than one case item expression matches the specified case expression.

Rules for Using translate_off and translate_on Pragma

The following rules are followed in IXCOM for determining whether or not the `translate_off` and `translate_on` pragma usage is suitable for hardware:

- An object written in the `translate_off` region should not be read inside in a non-translate_off region.
- The code inside the `translate_off` region should not be written to module ports.

The above usages result in a hardware-software mismatch, and modules containing such code are partitioned to the software side. Reading an object in the `translate_off` region is considered a legal usage for hardware. If the IXCOM compiler detects an assertion in the

© 2021 All Rights Reserved.

DUT that references a signal in a `translate_off` region in the design, the assertion is ignored with a warning.

Ignoring Pragmas in VHDL Designs

Use `-ignore_pragma` <*name*> option with the `vhan` command to ignore all the pragmas in the VHDL source prefixed with the specified <*name*>.

For example, `-ignore_pragma synopsys` will not honor pragmas that have `synopsys` as their prefix, such as, `synopsys translate_off/on`. The ignored pragmas are treated as comments.

Using //ixc Pragmas

The `//ixc` pragmas can be used for passing compiler directives to IXCOM as follows:

- To enable non-synthesizable code, such as `$display` statements embedded in a DUT module, typically located within the `//synopsys translate_off` blocks.

  ```
  // ixc tbcall_region
  ```

- To move the action blocks of assertions (in DUT) to hardware or software partition, instead of the default behavior, which is to run the pass action block on hardware and fail action block on software.

  ```
  // ixc hw_action_block
  // ixc sw_action_block
  ```

- To specify that all loops or a subset of the loops in a behavioral module should be unrolled prior to any behavioral transformation.

  ```
  // ixc bc_unroll_loop
  ```

- To *mark* a portion of a module that should be simulated while in software simulation mode.

  ```
  // ixc rtl_on
  // ixc rtl_off
  ```

Earlier, these pragmas were prefixed with `//axis` instead of `//ixc`. Support for //axis will be deprecated in future releases.

© 2021 All Rights Reserved.

Using Pragmas to Enable xrun to Handle Sparse Memories

Add the following pragma `/* sparse */` in the original HDL, to enable *xrun* to recognize and handle the memory declaration for sparse memories. This is especially useful in the case of large memories that would otherwise cause a compilation error.

For example:

```
---- test.v -----
module dut;
   reg [575:64] mem [0:16777215];
   /* sparse */
endmodule // dut
```

To compile the above design, the following command is used:

```
ixcom -clean +dut+dut -ua test.v
```

Compiling the above design to run using *xrun* will generate an error if this pragma is not added.

This pragma indicates that the memory declaration immediately preceding the pragma is to be implemented as a sparse memory during simulation (that is, physical memory is allocated dynamically for it at run time in the simulation). IXCOM recognizes this pragma and passes it to *xrun* for processing. This pragma has no effect on how the memory is implemented in hardware.

The `xrun` or `xmelab` commands enable you to specify that all arrays over a specified size are to be treated as sparse arrays by using the `-sparsearray` option when you elaborate the design. Specify the `-sparsearray` option with the`-enableLWD -lwdelabargs` command.

For example:

```
-enableLWD -lwdelabargs -sparsearray 2000000
```

The argument to the `-sparsearray` option is a positive number that indicates the number of array elements.

For more information on the `/* sparse */` pragma, refer to the *Sparse Arrays* section in the Xcelium documentation.

Pragmas to Mark Design Portions for Bypassing IXCOM Compilation

The following pragmas are used to mark regions within a given design unit declaration that are to be bypassed in the IXCOM front-end compilation and passed directly to Xcelium or HDL-ICE Compiler.

© 2021 All Rights Reserved.

Pragmas to Mark Code for Bypassing IXCOM Compilation

To mark UVM code for Xcelium compilation instead of IXCOM, use the `ixc_embed_sw` pragma as follows:

```
// pragma ixc_embed_sw header | endheader
// pragma ixc_embed_sw footer | endfooter
```

To mark low-level RTL code for compilation by HDL-ICE Compiler instead of IXCOM, use the `ixc_embed_hw` pragma as follows:

```
// pragma ixc_embed_hw header | endheader
// pragma ixc_embed_hw footer | endfooter
```

You can specify the location for dumping the pragma-marked code as either the beginning or the end of the module by using `header | endheader` and `footer | endfooter`.

If the region is marked with the following `header` and `endheader`, the region is dumped at the beginning of the module:

```
// pragma ixc_embed_sw header
// pragma ixc_embed_sw endheader
```

If the region is marked with the following `footer` and `endfooter`, the region is dumped at the end of the module:

```
// pragma ixc_embed_sw footer
// pragma ixc_embed_sw endfooter
```

Consider the following example of unsupported HDL:

```
module Tb;
  int count;
     << UVM Code >>
initial $monitor(count);
endmodule
```

The above code can be rewritten to mark the unsupported portions for Xcelium compilation as follows:

```
module Tb;
  int count;
    // pragma ixc_embed_sw header
          << UVM Code >>
    // pragma ixc_embed_sw endheader
  initial $monitor(count);
endmodule
```

Limitations when Passing Code for Compilation to Xcelium or HDL-ICE Compiler:

- These pragmas are not supported in compilation unit. Also, use of any declarations in compilation unit scope is not supported inside the blocks specified by these pragmas.
- These pragmas must not cross a design element, such as a module.

© 2021 All Rights Reserved.

- IXCOM dumps all the embedded software region at the beginning or end of the module. But the relative ordering of the regions is maintained. So, the first region always appears before the second region.
- You can use the pragma at least at statement level. This means that a part of statement cannot be inside the ixc_embed_sw pragma. For example, the following statement is invalid:

  ```
  a= /* pragma embed_hdl_sw on b + */  c + d;
  ```

- Pragmas should be the only text in the line.
- The `header`, `endheader`, `footer`, and `endfooter` are case-sensitive text.

Pragmas to Mark Unsupported HDL in Functions or Tasks for Xcelium Compilation

The following pragmas are used to bypass compilation of unsupported HDL in a function or a task in the IXCOM and pass the marked region to the Xcelium.

```
pragma ixc_svtb begin
pragma ixc_svtb end
```

For example:

```
function automatic void dut_send (bit [WIDTH-1:0] data, int mLen);
    int payload[] = new[mLen];
    for(int i=0; i < mLen; i++) payload = DUT_Buffer[i];
        //pragma ixc_svtb begin
            apb_rw rx = apb_rw::type_id::create("rx",, get_full_name());
            <put packed data and payload to rx>
            uvm_monitor.collect_rx(rx);
        // pragma ixc_svtb end
endfunction
```

Limitations when Marking Unsupported HDL in Functions or Tasks for Xcelium Compilation:

- IXCOM does not check for any errors in the code embedded between the `//pragma ixc_svtb begin|end` statements. The errors, if any, are reported during Xcelium compilation.
- The `ixc_svtb` pragma can enclose only statements.
- The `ixc_svtb` pragma should not enclose data declaration.
- IXCOM converts the code between `// pragma ixc_svtb begin|end` in to an `ixc_t` statement. So, if a data declaration is marked with this pragma, this data declaration is dumped with the statements. This might cause compilation errors in the Xcelium compile.

© 2021 All Rights Reserved.

- IXCOM cannot detect the scenarios in which the data declaration is embedded in the code.
- IXCOM cannot compile the following function, and generates an error specifying that `i` is not defined because the `int i = uvmfuncCall();` statement will be converted into `ixc_t task("ixc_svtb", ....);` and `i` will not be declared.

```
function automatic void dut_send (bit [WIDTH-1:0] data, int mLen);
    //pragma ixc_svtb begin
        int i = uvmfuncCall();
    // pragma ixc_svtb end

    int j= constFunc(i);
endfunction
```

- Partial marking of a statement is not valid. For example:

```
function automatic void dut_send (bit [WIDTH-1:0] data, int mLen);
    int payload[] = new[mLen];
    for(int i=0; i < mLen; i++) payload = DUT_Buffer[i];
    rx = // Invalid pragma - ixc_svtb
         // pragma should mark the complete assignment statement
    //pragma ixc_svtb begin
        apb_rw::type_id::create("rx",, get_full_name());
        uvm_monitor.collect_rx(rx);
    // pragma ixc_svtb end

endfunction
```

- Function ports cannot be marked with the `ixc_svtb` pragma. For example:

```
function automatic void dut_send ( //pragma ixc_svtb begin
    bit [WIDTH-1:0] data,
    // pragma ixc_svtb end
    int mLen);
endfunction
```

  Here, `ixc_t` is created in the same scope in which the `ixc_svtb` pragma appears.

- Pragmas should be the only text in the line.

© 2021 All Rights Reserved.

Reading and Writing Signals in DUT Modules

IXCOM lets you cross-reference nets and registers in DUT modules from your testbench for the purpose of using them in PLI calls, assuming the following conditions are met:

- DUT nets and registers can be read only; you cannot write to these objects.
- DUT nets and registers can be read at run time, only if they are exported at compile-time. Exporting a signal from the DUT enables you to read its value during simulation. Signals are automatically exported if:
  - You reference the DUT net or register in your testbench or other module running in software simulation mode, for example:

    ```
    always @(posedge testbench.chip.sys_clk)
                                // read reference to DUT signal
    ```

  - For Verilog, you explicitly export the signal using either the `$export_read` or `$export_frcrel` Verilog system tasks from a testbench module. Refer to *Command and System Tasks for SA Mode* chapter of the *VXE Command Reference Manual* for detailed information on these tasks.
  - For VHDL, you explicitly export the signal using either the `export_read` or `export_frcrel` VHDL procedures from a testbench module. Refer to *Command and System Tasks for SA Mode* chapter of the *VXE Command Reference Manual* for detailed information on these procedures.
- You can use VHDL `nc_mirror` procedures and Verilog `$nc_mirror` system tasks to trace the value of any VHDL or Verilog code in the design from a VHDL architecture or Verilog module. These procedures and system tasks are especially useful for monitoring the values of signals in the DUT during simulation and processing them in the testbench. Refer to Using nc_mirror in IXCOM for detailed information on these procedures and system tasks.
- If you use access routines in your PLI code, and they attempt to access signals that were not exported, you will get the following run-time error:

  ```
  Error! Cannot fetch value|set value|add vcl because the handle passed
  to acc_routine_name corresponds to an object signal_name_and_scope
  in hardware
  ```

- If you use `tf` routines, and you attempt to pass a reference to a DUT signal as a parameter to the PLI call, you will get an error during compile-time by default, for example:

  ```
  initial
      $my_pli(testbench.chip.my_reg);
  --
  Error: XMR defined in DUT cannot be assigned in testbench
  ```


© 2021 All Rights Reserved.

To address this issue, you must enable read access to the PLI routine. This is done with the `+tfReadOnly+pli_routine` compiler option. To enable read access for several PLI routines, append their names to this option:

```
+tfReadOnly+<pli_routine1>+<pli_routine2>+...
```

If you do not specify any routine names, read access is enabled for all routines in the design.

You can specify the `+tfReadOnly` option with the `ixcom` command.

For a Mixed HDL design:

```
ixcom -top <top_level_design_unit> [other_options] +tfReadOnly+<my_pli>
```

For a Verilog-only design:

```
ixcom <Verilog_sources> [other_options] +tfReadOnly+<my_pli>
```

Accessing DUT Signals from Non-HDL Interfaces

To access DUT signals from non-HDL interfaces like PLI, VPI, VHPI, Systemc, or Tcl, you need to write certain directives (`$export_read`, `$export_deposit`, and `$export_frcrel`). You can choose to generate these directives automatically by using the `-genhwaccessfile <hwaccess_file_name>` option with the `xrun` or `xmelab` command at the time of elaborating or compiling the design in pure simulation mode.

For example:

```
xrun [other_xrun_options] \
     -genhwaccessfile <hwaccess_file_name> \
     -input <sim.tcl>
```

During simulation, the objects accessed by the non-HDL interfaces are monitored along with the types of access required for each object. When you exit the simulation, a hardware access file is created with the specified filename and extension (`.v` or `.sv`). If no extension is specified, by default, the file is given a `.sv` extension.

The name of the generated module matches the file name specified with the `-genhwaccessfile` option.

The hardware access file contains the export directives for the signals to be read, written, and forced, enclosed within an `initial` statement. The `$export_read`, `$export_deposit`, and `$export_frcrel` directives are used to specify the signals to be read, written, and forced, respectively. Following is an example of a hardware access file:

```
(* ixcom_instrumented_mod *)
module hwaccess;
      `ifdef IXCOM_COMPILE
       initial
        begin
```


© 2021 All Rights Reserved.

```
          $export_read(<pathname1>, <pathname2>, ..., <pathname10>);
          $export_read(<pathname11>, <pathname12>, ..., <pathname20>);
          $export_deposit(<pathname1>, <pathname2>, ..., <pathname10>);
          $export_read( <pathname21>, <pathname22>);
          $export_frcrel( <pathname1>, <pathname2>, ..., <pathname8>);
          $export_deposit(<pathname11>);
        end
      `endif
endmodule
```

The signals are qualified by their full path and the directives can appear in any sequence. If a signal is read, written, and forced, all the three directives, `$export_read`, `$export_deposit`, and `$export_frcrel` are generated for that signal qualified by its full path.

Then, at the time of compilation, specify this hardware access file with the `ixcom` or `xrun -hw` command (when using single-step compilation method) or with the `vlan` command (when using multi-step compilation method) as an additional source file. No other changes are needed to your source files:

- When using `ixcom` to compile the designs in single-step, use:

  ```
  ixcom {[other_ixcom_options] [<hwaccess_file_name>]}
  ```

- When using `ixcom` to compile the designs in multi-step, use:

  ```
  vlan {[other_vlan_options] <hwaccess_file_name>}
  ixcom [-top <hwaccess_file_module_name>] [other_ixcom_options]
  ```

- When using *xrun* to compile designs for Simulation Acceleration mode:

  ```
  xrun -64 -hw -ua [{-top <hwaccess_file_module_name>}] \
           [other_ixcom_options] [<hwaccess_file_name>] \
           [other_xrun_options]
  ```

Considerations for Using the -genhwaccessfile Option

Keep in mind the following points when using the `-genhwaccessfile` option:

- The `-genhwaccessfile` option should be used to generate an access file in pure simulation environment only. This option should not be used on an IXCOM-compiled snapshot by running *xrun* on it because this adds certain unwanted signals (like those in `xc_top` and other internally instrumented signals) in the hardware access file, which can degrade performance.
- Avoid use of *xrun* options, such as `-ncinitialize` that can dump redundant signals in the hardware access file, thereby impacting performance.
- The `-genhwaccessfile` option is not supported with Specman flow.

© 2021 All Rights Reserved.

- For SystemVerilog compilation unit scope names, the names are generated in the following manner in the hardware access file:

  ```
  $export_read(worklib.$unit_0x7189c4db::<var_name>)
  ```

  So, to compile this file with IXCOM, you need to manually modify such names as follows:

  ```
  $export_read($unit::<var_name>)
  ```

Generating Directives Automatically for OOMRs and Import-Export Tasks

This section describes the process of automatic generation of export and `$ixc_ctrl` directives for testbench-DUT OOMRs and export-import task and function calls.

1. Use ***xrun*** to parse the design and generate the required directives.

   ```
   xrun -genhwaccessfile <directives>.v file -ixcom_dut_info <dut_spec_file>
   ```

   Refer to Format of the DUT Specification Information File on page 1135 for details about the <*dut_spec_file*> that you need to specify with the `-ixcom_dut_info` option of the `xrun` command.

   Alternatively, you can parse the design and generate the required directives in multiple steps as follows:

   ```
   xmvlog -sv [other_xmvlog_options]
   xmelab -genhwaccessfile <directives>.v [other_elab_options] \
          -ixcom_dut_info <dut_spec_file>
   ```

2. Specify the <*directives*>`.v` file along with the design files and compile using IXCOM.

   ```
   ixcom +xcDesignTop -f <design_files>.f <directives>.v
   ```

   Important

   Specify both `-genhwaccessfile` and `-ixcom_dut_info` options to generate export directives for DUT OOMR references from testbench. If the `-ixcom_dut_info` option is not specified, the file for DUT OOMRs will not be generated.

3. Elaborate the modified source of the design using *xrun*.

   ```
   xrun -elaborate -f <IES_files>.f -f ./xc_work/xrun.f ...
   ```

   Here, <*IES_files*>`.f` are the set of files that were not specified with the `+xcDesignTop` option of the `ixcom` command in the previous step.

   After elaboration, a module named <*hwaccess_file_module_name*> is generated in the `initial` block in the <*directives*>`.v` file. This is the filename specified with the `-genhwaccessfile` option of the `xrun` command. This module includes all the

© 2021 All Rights Reserved.

directives for testbench-DUT OOMRs from the testbench side and the `$ixc_ctrl` directives for task and function calls as illustrated in the following code:

```
module <hwaccess_file_module_name>;
`ifdef IXCOM_COMPILE
initial
begin
    $export_read(<pathname1>, <pathname2>, ..., <pathname10>);
    $export_deposit(<pathname1>, <pathname2>, ..., <pathname10>);
    $export_frcrel( <pathname1>, <pathname2>, ...);
    $export_read((<pathname11>, <pathname12>);
    $ixc_ctrl("tb_export", "foo");
    $ixc_ctrl("tb_import", "bar");
end
`endif
endmodule
```

In this module, `tb_export` directive is used for a task that is scheduled in the hardware. For example, the `foo` task executes in hardware. Similarly, the `tb_import` directive is used for a task that is scheduled in the software. For example, the `bar` task executes in software.

DUT tasks are specified with `tb_export` directive and testbench tasks are specified with `tb_import` directive. Therefore, DUT tasks execute in hardware when called from testbench, and similarly, testbench tasks execute in software when called from DUT.

4. If you want to generate the export directives for signals accessed through TCL, VPI, or VHPI interfaces, run the design using ***xrun***. After the simulation run, all directives (for OOMRs and signals accessed through TCL, VPI, or VHPI interfaces are added to the *<hwaccess_file_module_name>* module in the `initial` block of the *<directives>*`.v` file.

```
xrun -sv -genhwaccessfile <directives>.v file \
     -ixcom_dut_info <dut_spec_file>
```

Alternatively, you can run the design and generate the required directives in multiple steps as follows:

```
xmvlog -sv [other_xmvlog_options]
xmelab -genhwaccessfile <directives>.v [other_elab_options] \
       -ixcom_dut_info <dut_spec_file>
xmsim [other_xmsim_options]
```

Supported Interfaces with Hardware Access Files

IXCOM adds instrumentation to ensure proper accessibility for signals read, written, and forced through non-HDL interfaces, such as PLI/VPI, VHPI, SystemC, and Tcl. The generated

© 2021 All Rights Reserved.

directives corresponding to the functions of PLI/VPI, VHPI and TCL interfaces are listed in the following table:

Interface	Directive	Functions

PLI 1.0	`$export_read`	`acc_vcl_add()`
		`acc_fetch_value()`
		`acc_fetch_paramval()`
		`acc_fetch_paramval_mtm()`

	`$export_deposit`	`acc_set_value()` with delay model : `accAssignFlag`, `accNoDelay`, `accInertialDelay`, `accTransportDelay`, `accPureTransportDelay`
	`$export_frcrel`	`acc_set_value()` with delay model : `accForceFlag`

VPI	`$export_read`	`vpi_get_value()`
		`vpi_register_cb()`
		`cbValueChange`
		`cbForce/`
		`cbRelease` (on the expression)
		`cbAssign/`
		`cbDeassign` (on the register assigned)

	`$export_deposit`	`vpi_put_value()` with the `vpiAssignFlag`, `vpiNoDelay`, `vpiInertialDelay`, `vpiTransportDelay`, and `vpiPureTransportDelay` flags
	`$export_frcrel`	`vpi_put_value()` with the `vpiForceFlag` flag

© 2021 All Rights Reserved.

Interface	Directive	Functions

VHPI	`$export_read`	`vhpi_get_value()`
		`vhpi_register_cb()`
		for `vhpiCbValueChange`

	`$export_deposit`	`vhpi_put_value()` with the `vhpiDeposit` and `vhpDepositPropagate` flags
	`$export_frcrel`	`vhpi_put_value()` with the `vhpiForcePropagate` flag Note: The `vhpiForce` is not supported.
SystemC	`$export_read`	`observe_foreign_signal()` `control_foreign_signal()`
	`$export_deposit`	`control_foreign_signal(<signal-name>, false)`
	`$export_frcrel`	`control_foreign_signal(<signal-name>, true)`

For the Tcl command interface, the export directives are dumped into the hardware access file according to the information available in the *Access Requirements for Tcl Interactive Commands* section of the *Xcelium Direct Invocation: Elaboration* document in the Xcelium documentation set.

Using nc_mirror in IXCOM

The `nc_mirror` procedure or system task enables you to trace the value of a VHDL signal, port, or variable or a Verilog variable or wire lying anywhere in the design from a VHDL architecture or Verilog module. These procedures and system tasks can also be used to mirror the value of a source signal to a destination signal.

With an `nc_mirror` call, whenever the value of the specified source object is modified, the modified value is copied to the specified destination object. An `nc_mirror` call does not create additional drivers between the source and destination objects.

The procedure or system task performs syntax and error checks on the "*<source>*" and "*<destination>*" parameters to ensure that the two signals are of a compatible data type, same size, and valid range. If the two do not match, it reports an error message for the unsupported data types.

© 2021 All Rights Reserved.

Note: In *xrun*, mirroring of the value starts after nc_mirror is executed. In VXE, mirroring starts at the beginning of simulation regardless of when nc_mirror is actually executed.

VHDL nc_mirror() procedure

To use the `nc_mirror` procedure in the VHDL code, add the following lines before the entity declaration:

```
library AXIS;
use AXIS.axis_util.all;
```

Syntax:

```
nc_mirror ("<destination>", "<source>", "verbose");
```

Or,

```
nc_mirror (destination => "<destination>",
           source => "<source>",
           verbose => "verbose");
```

Verilog $nc_mirror() system task

To use the `$nc_mirror` system task in the Verilog code, use the following syntax:

Syntax:

```
$nc_mirror ("<destination>", "<source>", "verbose");
```

Parameters

Table 16-2, "Parameters for nc_mirror," on page 802 lists the parameters for the `nc_mirror` procedures and system tasks.

Table 16-2 Parameters for nc_mirror

Parameter	Description
"<destination>"	Specifies the full hierarchical path or a relative path of a VHDL signal, port, or variable or a Verilog variable or wire to which the value of the source signal is to be mirrored. You must specify either the destination signal, or pass a null string using `""`. If you do not specify a destination signal, the new value of the source signal is displayed.

© 2021 All Rights Reserved.

Table 16-2 Parameters for nc_mirror, *continued*

Parameter	Description
"*<source>*"	Specifies the full hierarchical path or a relative path of a VHDL signal, port, or variable or a Verilog variable or wire to be mirrored or traced. This parameter is mandatory.
"verbose"	Optional parameter used for status reporting. It displays extra information.

- Full Hierarchical Path: Full hierarchical paths begin with the top-level module in the hierarchy.

  The paths begin with the top-level module name in the Verilog code.

  For example:

  ```
  $nc_mirror("top_mod.dest", "top_mod.I1.src", "verbose");
  ```

  The paths begin with a colon (:) in the VHDL code.

  For example:

  ```
  nc_mirror(":dest", ":I1:src", "verbose");
  ```

- Relative Path: Relative paths begin with a module instance name in the current module.

  ```
  nc_mirror("dest", "I1:src", "verbose");
  ```

  The upscope operator (^) enables you to move one level up in the hierarchy.

  For example:

  ```
  nc_mirror("dest", "^.^.I2.signal1", "verbose");
  ```

Supported Data Types

Table 16-3, "Supported data types for nc_mirror," on page 804 lists the supported data types for the `nc_mirror` procedures and system tasks.

© 2021 All Rights Reserved.

Table 16-3 Supported data types for nc_mirror

Language	Signal[1]		Supported Data Types	
	Source	**Destination**	**Source**	**Destination**
VHDL-to-VHDL	Testbench	Testbench	Scalar types and one dimensional array [2]	Scalar types and one dimensional array
	DUT	Testbench	std_ulogic, std_logic, std_ulogic_vector, or std_logic_vector	std_ulogic, std_logic, std_ulogic_vector, or std_logic_vector
	DUT	DUT	std_ulogic, std_logic, std_ulogic_vector, or std_logic_vector	std_ulogic, std_logic, std_ulogic_vector, or std_logic_vector
VHDL-to-Verilog	Testbench	Testbench	std_ulogic, std_logic, std_ulogic_vector, or std_logic_vector	Packed array
	DUT	Testbench	std_ulogic, std_logic, std_ulogic_vector, or std_logic_vector	Packed array
	DUT	DUT	std_ulogic, std_logic, std_ulogic_vector, or std_logic_vector	Packed array
Verilog-to-Verilog	Testbench	Testbench	Packed array	Packed array
	DUT	Testbench	Packed array	Packed array
	DUT	DUT	Packed array	Packed array
Verilog-to-VHDL	Testbench	Testbench	Packed array	std_ulogic, std_logic, std_ulogic_vector, or std_logic_vector
	DUT	Testbench	Packed array	std_ulogic, std_logic, std_ulogic_vector, or std_logic_vector
	DUT	DUT	Packed array	std_ulogic, std_logic, std_ulogic_vector, or std_logic_vector

1. Testbench-to-DUT signal: Not Supported. nc_mirror does not support mirroring a value from the source signal in testbench to the destination signal in DUT.

© 2021 All Rights Reserved.

2. Record and array data types: Record types group elements of many types and each element can be accessed by its field name. You can access an element of a record that has a supported data type. Array types group elements of same types. You can access an array that has a supported data type.

Limitations

Note the following restrictions on the use of `nc_mirror`:

- When both the "*<destination>*" and "*<source>*" signals are VHDL, the data-types *must* be assignment compatible.
- When copying from Verilog to VHDL or vice versa, the Verilog bit width *must* match that of VHDL.
- Mirroring a value from the "*<source>*" signal in testbench to the "*<destination>*" signal in DUT is not supported for nc_mirror.
- When the "*<source>*" signal is in DUT and the "*<destination>*" signal is in testbench, the primary output must be created for the "*<source>*" signal.

Using Plus Arguments

IXCOM supports the use of *plus* arguments at run time, which are checked with the `$test$plusargs` or `$value$plusargs` system task in the Verilog part of your design only. By using this task and by specifying different plus arguments, you can cause different code, such as different tests, to run when you simulate your design. All plus options passed to ixcom are passed to run-time software.

For example, use the `$test$plusargs` system task to selectively enable VCD-on-demand dumping at run time, as follows:

```
initial
begin
if ($test$plusargs("dumpon"))
 begin
     $dumpfile("design.vcd");
     $dumpvars;
 end
end
```

Specifying `+dumpon` at run time gives you the flexibility of enabling or disabling dumping without recompiling:

```
xrun -R +dumpon /*runs with dumping enabled*/
xrun -R /*runs with dumping disabled*/
```


© 2021 All Rights Reserved.

Support for Encryption in SA Mode

IXCOM supports the following encryption methods, by default:

- Verilog-2001 standard encryption by `xmprotect`
- Verilog-XL encryption by Verilog-XL
- Verilog-2005 standard encryption by `xmprotect`

By default, the `xmprotect` encryption is used. You can import all files with any encryption style using IXCOM compiler without using any additional options.

IXCOM also supports a combination of these encryption styles across different analyzer sessions. For example, suppose:

- ❑ `a.v` is encrypted to `a.vp` using Verilog-XL style
- ❑ `b.v` is encrypted to `b.vp` using xmprotect style

Then, these files need to be analyzed separately, as follows:

```
vlan +sv a.vp
vlan +sv b.vp
```

Reading Encrypted Files

You can use the Cadence `xmprotect` utility to encrypt files for both Verilog and VHDL by using the following ways:

- Use the `-autoprotect` option to encrypt the entire source file.

  ```
  xmprotect -autoprotect <source_file>
  ```

- Specify a source file that contains the `protect begin` and `protect end` pragmas to mark the regions for encryption.

  ```
  xmprotect <source_file>
  ```

  Or,

  ```
  xmprotect -language {vlog|vhdl} <source_file>
  ```

  For example, the following `<source_file>` contains a protection pragma:

  ```
  //pragma protect
  //pragma protect begin
  module Add1 (sum, cout, ain, bin, cin);
  input ain, bin, cin;
  output sum, cout;
  //pragma protect end
  ```


© 2021 All Rights Reserved.

By default, when the `-autoprotect` option is not used, `xmprotect` encrypts the text between `//pragma protect begin` and `//pragma protect end`.

For the Verilog-2005 standard, `xmprotect` encrypts the text between `'pragma protect begin` and `'pragma protect end`.

Support for IEEE Encryption Standards

To provide a secure environment for sharing encrypted files with third-party software and Cadence tools, IXCOM compiler provides support for the IEEE standards - P1800 and P1735-V1 as Xcelium. In addition to the default Cadence-generated key (`CDS_KEY`), you can specify your own encryption keyword to encrypt RTL and gate-level Verilog and/or SystemVerilog IPs. IXCOM compiler preserves the same keyword for all IPs, including the generated gate-level netlist and the intermediate VHDL files. All of these files are encrypted with the same encryption mechanism and user-defined keyword. The specified user-defined keyword is used at all stages of processing encrypted IPs - including reading or decrypting IPs and writing or encrypting IPs. In addition, you can specify different `Key_block` algorithm and `Data block` algorithm to generate the encrypted-IP modules. The support for user-defined keywords is provided at all stages of compilation - design import, HDL-ICE Compiler, `vavlog`, `vavhdl`, `vaelab`, IXCOM compiler, `vlan`, and `vhan`. The tools generate the same level of secured gate-level IP as the encrypted RTL IP, and the output is encrypted according to the user-specified keyword and user-selected mechanism.

To provide the encryption keys and algorithms to `xmprotect`, use one of the following ways:

- Specify the encryption information in the same RTL file enclosed within `` `pragma protect ``.

Or,

- Specify the encryption information in a separate file and save all the `` `pragma `` lines in that file. Use the `-parameter` <*`pragmas_input_file`*> option of the `xmprotect` command to specify the name of the file.

For more information on using the `xmprotect` utility, refer to *Protecting IP Source Files -- IP Protection* manuals available in the Xcelium documentation set.

Encryption with Levels of Debug Capability

You can use the Cadence `xmprotect` utility to encrypt files and specify the levels of debug capability by using the following ways:

© 2021 All Rights Reserved.

- Use the `-autoprotect` and `-simulation {portview|debugall|none}` options to encrypt the entire source file and specify the level of debug capability.

  ```
  xmprotect -autoprotect -simulation {portview|debugall|none} <source_file>
  ```

  You can specify the following levels of debug capability:

  - `portview`: Provides access to module and entity names, ports, generics, and parameters. Access to internal registers, wires, local parameters, signals, and variables is not allowed.
  - `debugall`: Provides the ability to traverse the complete design.
  - `none`: Denies debug capability.

- Specify a source file that contains `rights` pragma to specify the level of debug capability as `portview`, `debugall`, or `none`.

  ```
  xmprotect <source_file>
  ```

  For example, a source file can contain the following `rights` pragma:

  ```
  //pragma protect rights=(simulate=(viewers="portview"))
  ```

  If two pragmas with different protection rights are specified in the same region, then the first pragma is used to encrypt the source file.

Application of Protection Rights

A protection right is applied to a source file or a module based on the following conditions:

- When a module is protected with the same protection right, it is dumped with the specified protection right in the following cases:
  - The entire source file or module is protected with one pair of `rights` pragma and the `-autoprotect` option is not specified to `xmprotect`.
  - A source file is protected by specifying the `-autoprotect` and `-simulation{portview|debugall|none}` options to `xmprotect`.

- When a module has multiple protected regions with different protection rights, then the IXCOM compiler will honor all protection rights and the protected regions of the module are dumped with the same protection right.

  For example, the following module is protected with the `xmprotect -lang vlog test.v` command:

  ```
  //pragma protect begin
  //pragma protect rights=(simulate=(viewers="portview"))
  module m (p1, p2);
  //pragma protect end
  wire w1;
  ```


© 2021 All Rights Reserved.

```
//pragma protect begin
//pragma protect rights=(simulate=(viewers="debugall"))
assign p2 = p1;
//pragma protect end

endmodule
```

It is dumped as the following module:

```
//pragma protect begin
//pragma protect rights=(simulate=(viewers="portview"))
module m (p1, p2);
//pragma protect end
wire w1;
//pragma protect begin
//pragma protect rights=(simulate=(viewers="debugall"))
assign p2 = p1;
//pragma protect end

endmodule
```

- When a source file or a module is protected with multiple protection rights, but the `-autoprotect` and `-simulation{portview|debugall|none}` options are specified to `xmprotect`, then it is dumped with the protection right specified with the `-autoprotect` option.

  For example, the following module is protected with the `xmprotect -autoprotect -simulation debugall test.v` command:

```
//pragma protect begin
//pragma protect rights=(simulate=(viewers="debugall"))
module m (p1, p2);
//pragma protect end
wire w1;
//pragma protect begin
//pragma protect rights=(simulate=(viewers="none"))
assign p2 = p1;
//pragma protect end

endmodule
```

  It is dumped as the following module:

```
//pragma protect begin
//pragma protect rights=(simulate=(viewers="debugall"))
module m (p1, p2);
wire w1;
assign p2 = p1;
endmodule

//pragma protect end
```

Support for HDL-ICE Compiler Pragmas in Side File

This section describes the support for directive-side file in which you can specify directives or pragmas for HDL-ICE Compiler, such as `keep_net`, `preserve`, `external_ref`, `use_hardmacro`, `no_hardmacro`, and so on. Specify the directive-side file to IXCOM by

© 2021 All Rights Reserved.

using the `-directiveFile` <*directivefile_name*> option of the `ixcom` command. IXCOM processes the directive file and marks the pragmas on the variables or signals.

Format of the Side File

The directive file can have two types of entries for specifying the pragmas:

- Module-based:

  `M <library_name> <module_name> <net_name> <directive> [fatal]`

- Path-based

  `P <hierarchical path to net name> <directive> [fatal]`

  Note: The hierarchical path should be specified in valid RTL name format.

By default, warning is generated if:

- For Module-based entries: the net is not found in the module or library or a module is not found in the library.

Or

- For Path-based entries: a path does not exist.

If, instead of warning, you want error message to be generated, add `fatal` at the end of the line in the directive file.

For both types of entries, the pragma is applied to the specified signal or variable. The effect of an entry of the directive file is the same as the target signal or variable having RTL embedded with the pragma. If multiple pragmas are specified for the same signal, all of these are added to that signal or variable.

- Error messages are generated if:
  - The directive file is empty or does not exist.
  - The entries in the directive file are written in wrong or incorrect format.
- All pragmas other than `keep_net`, `preserve`, `external_ref`, `use_hardmacro`, and `no_hardmacro` are ignored with a warning.
- Bit-select, part-select, and field-select are ignored and the pragma is applied to the whole signal.
- Extra whitespace and blank lines are ignored.
- C++ style single-line comments are allowed.

© 2021 All Rights Reserved.

- The `keep_net`, `preserve`, `external_ref` pragmas are supported in both Module-based and Path-based entries. The `use_hardmacro` and `no_hardmacro` pragmas are supported only with Module-based entries.
- For Module-based entries, pragmas can be specified only for module-level signals.
- Directive file is supported for SystemVerilog designs.
- If you specify the `keep_net` pragma for a signal, the `keep_net` property is automatically propagated to xeCompile so that you can force or deposit the signal at run time without needing to specify the signal as a user data `keepNet`.

Support for Strengths in SA Mode

IXCOM supports the following strengths on the primitives and continuous assignments:

- ❑ `SUPPLY`
- ❑ `STRONG`
- ❑ `PULL`
- ❑ `WEAK`

The following strengths are mapped to `WEAK`:

- ❑ `HIZ`
- ❑ `SMALL`
- ❑ `MEDIUM`
- ❑ `LARGE`

The strengths are supported on primitives that can be mapped to hardware. The bus-resolution logic will incorporate strength information during the value resolution. Any conflicts that should result in a value of `X` in simulation will be mapped to `0` in hardware.

Handling of UDPs by IXCOM

Specify the `{+convertUdp[<filename>]}` option of the `ixcom` command to direct IXCOM to map all the UDPs in the design to modules. Each mapped module has the same name and functionality as that of the corresponding UDP. If IXCOM fails to map a UDP to a corresponding module, a `MAPUDP_FAILED` warning message is issued. Compilation

© 2021 All Rights Reserved.

continues and the UDP (which IXCOM failed to map to a module) continues to be treated as a UDP.

If the <*filename*> optional parameter is specified, the original UDP and the module it is mapped to, are both output in the <*filename*>. This helps debug and ensure that the mapped module has the same functionality as the original UDP.

Instantiations of UDPs are not allowed in the DUT or hardware partition, and if found, IXCOM, by default, maps those UDPs to modules, even if the `+convertUdp` switch was not specified with the `ixcom` command. If IXCOM fails to map a particular UDP to a module, the `MAPUDP_FAILED` warning is issued, followed by the `DUT_UDPCHILD` error message.

© 2021 All Rights Reserved.

Support for Verilog Configuration Blocks

IXCOM supports Verilog configuration blocks to facilitate sharing of designs.

Verilog configuration blocks are a set of rules that specifies the exact source description to bind to each instance in a design. The following configuration blocks are supported by IXCOM:

- To design the global parameters of the configuration:
  - The `design` statement
  - The `default` clause
- To specify an instance or a module:
  - The `instance` clause
  - The `cell` clause
- To specify what to use for an instance or a module:
  - The `liblist` clause
  - The `use` clause
- To specify hierarchical configurations use the `config` tag.

Note: IXCOM reports an error when Verilog configuration blocks are used for binding VHDL design units.

Following is the syntax of supported Verilog configuration blocks:

```
config <config_name>;
    design [<library>.]<cell>
    default liblist [{<libraries>}];
    cell [<library>.]<cell> liblist [{<libraries>}];
    cell [<library>.]<cell> use [<library>.]<cell>[:config];
    instance <full_hierarchical_path> liblist [{<libraries>}];
    instance <full_hierarchical_path> use [<library>.]<cell>[:config];
endconfig
```

You can use a library map file to specify libraries. The library map file is used to specify which file goes to which library. The files are analyzed and then the top-level configuration is used to elaborate the design. You can use the format specifier `%l` or `%L` to display the actual library binding information for module instances during simulation.

© 2021 All Rights Reserved.

Examples

Following is an example of Verilog configuration blocks:

```
config cfg1;
    design rtlLib.top ;
    default liblist aLib rtlLib;
    instance top.a3 liblist gateLib;
    cell myaddr use rtlLib.adder1;
endconfig
```

Following is an example of hierarchical configurations:

```
config cfg1;
    design rtlLib.top ;
    default liblist aLib rtlLib;
    instance top.a3 use worklib.foo:config;
endconfig
```

In this example, the instance `top.a3` binds with the design hierarchy specified by the configuration `worklib.foo:config`.

Compiling Designs with Verilog Configuration Blocks

Perform the following steps to compile designs with Verilog configuration blocks:

1. Use the `vlan` tool to analyze the configurations and the Verilog modules into respective logical libraries.

   ```
   vlan -work <logical_library> {<Verilog_files>}
   ```

2. Use IXCOM to elaborate the analyzed Verilog design, specifying the relevant `ixcom` command options. With the `-top` option, specify a previously compiled Verilog configuration, where the configuration design statement specifies the top-level module.

   ```
   ixcom -top [<library>.]<Verilog configuration name> [other_ixcom_options]
   ```

Support for Overriding VHDL Generics from the IXCOM Command Line

Use the `-generic` option of the `ixcom` command to override the value of a VHDL generic from the `ixcom` command line. Following is the syntax of this option:

```
-generic "<generic_name> => <value>"
```

Here, <*generic_name*> is the name of the generic as it appears in the VHDL source. Specify the full path for the generic to be overridden at the command line, using '`.`' (period) as the hierarchy separator.

© 2021 All Rights Reserved.

This option can be used with any mixed-language designs for any combination of Verilog, VHDL, and SystemC hierarchies, such as Verilog->VHDL->VHDL->Verilog, VHDL->Verilog->VHDL->VHDL->Verilog->VHDL->SystemC, and so on.

Important

The generic overridden through this option should belong only to a VHDL design unit.

The <*value*> argument specifies an appropriate value for the declared data type of the generic. This overridden value has the highest precedence as it overrides the default generic value declared in the entity, the value passed in the instance declaration or through any configuration declaration or specification. Values can be passed to a generic of an instance at any level of the design hierarchy. The default or initial value expression in the original VHDL is optional. However, if the default value is left out in the original HDL, it is mandatory to specify the value using the generic override command-line feature.

For example, assume that there is a top-level generic called gen, in the following top-level entity:

```
entity test is
generic (gen : integer);
port (x : in bit;
sum : out bit);
end test;
```

To pass a value to this top-level generic, specify the name of the generic, or the hierarchical path of the generic, on the command line as follows:

```
% ixcom -ua +dut+top -top test -generic "test.gen=>4"
```

Or,

```
% ixcom -ua +dut+top -top test -generic "gen=>4"
```

To pass a value to a lower-level generic, provide the full hierarchical path to the generic, as follows:

```
% ixcom -ua +dut+top -top top -generic "top.I1.I2.test.gen=>4"
```

To pass a value to a case-sensitive generic declared with the escape character, you need to provide the escaped name in the argument, as follows:

```
% ixcom -ua +dut+top -top test -generic "\gen\ =>4"
% ixcom -ua +dut+top -top test -generic "\GEN\ =>6"
```

Specify the `-generic` option multiple times on the same command line to specify values for multiple generics. For example:

```
% ixcom -ua +dut+top -top test -generic "\gen\ =>4" \
        -generic GEN_STRING=>"abc" \
        -generic "GEN_TIME =>1us"
```


© 2021 All Rights Reserved.

The order in which the generics are specified does not matter unless you are specifying two values for the same generic. In that case, the value specified with the last `-generic` option on the command line takes precedence, and overrides the former value. Also, a warning message is generated in verbose mode to highlight this usage of the `-generic` option.

Tip

The `-generic` command-line options can also be included in an arguments file that you can include using the `-f` option of the `ixcom` command.

Supported Data Types for Generic Overrides

Following is the list of supported data types for generic overrides. Note that all these data types are supported by Xcelium.

- `(generic G: INTEGER := ...)`
- `(generic G: REAL := ...)`
- `(generic G: STD_LOGIC := ...)`
- `(generic G: STD_LOGIC_VECTOR := ...)`
- `(generic G: BIT := ...)`
- `(generic G: BIT_VECTOR := ...)`
- `(generic G: TIME := ...)`
- `(generic G: STRING := ...)`
- `(generic G: NATURAL := ...)`
- `(generic G: POSITIVE := ...)`
- `(generic G: BOOLEAN := ...)`
- `(generic G: user-defined enumerated type)`

Considerations for the Supported Data Types for Generic Overrides

For the supported data types, the following requirements and limitations need to be kept in mind:

© 2021 All Rights Reserved.

INTEGERS

- Integers must be decimal literals and not based literals.
- The literals can contain underscores. The restrictions on leading, trailing, and double underscores are not enforced.

TIME

- Time literals must be decimal physical literals.
- The abstract literal portion (number) must be present.
- The preceding comments for integers apply to TIME literals.
- The unit portion of the physical literal is required and can be any time unit name from `fs` through `hr`.

Strings

- Strings with spaces or other special characters must be delineated by double quotes.
- Simple alphanumeric strings without spaces need not have quotes.
- Double quotes can be embedded in the string by using the escape character '`\`'.

BIT / STD_LOGIC

Values for `BIT` and `STD_LOGIC` types must be enclosed in single quotes.

BIT_VECTOR and STD_LOGIC_VECTOR

- If the constraint and direction is specified in the HDL code:
  - The size of the generic being passed with the `-generic` option must match the size specified in the HDL code.
  - If the size does not match, a warning message is generated. The elaboration continues, but the generic override directive is ignored.
- If the constraint and direction is not specified in the HDL code:
  - The constraint is determined from the value of the generic specified on the command line.

© 2021 All Rights Reserved.

- The direction is set to `TO`.
- The left constraint is set to `0`.
- The right constraint is set to the length of the vector minus 1 (`length - 1`).

Example for Generic Override

Given the following entity and architecture declarations:

```
library ieee;
use ieee.std_logic_1164.all;
library std;
use std.textio.all;

entity E is
    generic (GNRC_INTEGER : INTEGER := 0;
    GNRC_TIME: TIME := 0 ns;
    GNRC_STRING : STRING := "";
    GNRC_BOOLEAN : BOOLEAN := FALSE;
    GNRC_NATURAL : NATURAL := 10;
    GNRC_POSITIVE : POSITIVE := 5;
    GNRC_STD_LOGIC : STD_LOGIC := '1';
    GNRC_STD_LOGIC_VEC_1 : STD_LOGIC_VECTOR (0 TO 3) := "1111";
    GNRC_STD_LOGIC_VEC_2 : STD_LOGIC_VECTOR;
    GNRC_BIT : BIT := '1';
    GNRC_BIT_VEC : BIT_VECTOR := "0011";
    GNRC_REAL : REAL := 2.0);
end;

architecture A of E is
begin
    ...
end;
```

For the above example, the following `-generic` options are supported with the `ixcom` command:

```
-generic "GNRC_INTEGER => 99"
-generic "GNRC_INTEGER => -99"
-generic "GNRC_TIME => 17 ns"
-generic "GNRC_STRING => abc"
-generic 'GNRC_STRING => "abc xyz"'
-generic 'GNRC_BOOLEAN => "TRUE"'
-generic "GNRC_NATURAL => 12"
-generic "GNRC_POSITIVE => 7"
-generic "GNRC_STD_LOGIC => 'U'"
-generic "GNRC_STD_LOGIC_VEC_1 => U100"
-generic "GNRC_STD_LOGIC_VEC_2 => UUUU"
-generic "GNRC_BIT => '0'"
-generic "GNRC_BIT_VEC => 1100"
-generic "GNRC_REAL => 9.0"
```

On the other hand, the following `-generic` options are not supported with the `ixcom` command:

© 2021 All Rights Reserved.

```
-generic "GNRC_INTEGER => 16#FF#"   -- No based literals
-generic "GNRC_TIME => 0.1 ns"   -- No real literals. Use 100 ps.
-generic "GNRC_BOOLEAN => 0"   -- Value must be “true”/”false”
-generic "GNRC_BIT => 1"   -- Value must be in single quotes
-generic "GNRC_STD_LOGIC_VEC_1 => U1" -- Ignored with a warning. Value
                                      -- is two bits but GNRC_STD_LOGIC_VEC_1
                                      -- declared as (0 TO 3)
```

Comparison with Xcelium Support

The generic override feature of IXCOM is based on the corresponding feature in Xcelium, and its usage and support is similar to Xcelium. The supported set of data types for generic override in IXCOM is the same as Xcelium as listed in Supported Data Types for Generic Overrides on page 816.

However, there are a few differences between the Xcelium and IXCOM support, which are listed in the following sections.

- Generics Defined Within Named Blocks on page 819
- Backward Compatibility in the Syntax for Specifying Generics on page 820
- Usage of the -relax Option on page 820

Generics Defined Within Named Blocks

Generics inside named blocks are not supported by IXCOM. This means that a generic defined in a block statement like the following generic `G1` cannot be overridden from the command line:

```
B1: block
generic (G1: integer); --or generic map(….)
begin
   ...
end
```

This is different from Xcelium because Xcelium enables generic definition in named blocks.

Note that a named block, which is merely a part of the hierarchical path for the generic that is to be overridden, is supported. In other words, an instance defined in the named block whose instance needs to be overridden is supported. For example:

```
B1: block
begin
I1 : dut generic map(….) port map(…);
end
```

You can override a generic in the instance `Top....B1.I1.GEN` that needs to be overridden from the command line, it can be done so by using the full hierarchical path.

© 2021 All Rights Reserved.

Backward Compatibility in the Syntax for Specifying Generics

When using Xcelium, you specify the `-generic` option as follows:

```
-generic "Top.A1.G1(0).I1.GEN => 5"
```

The IXCOM compiler supports the above syntax. But, for backward compatibility, the IXCOM compiler also supports previously used syntax, such as the use of:

- an underscore (_) in place of parentheses
- the = sign instead of =>

For example, note that `G1_0` is used instead of `G1(0)` for generate block and the = sign instead of => below:

```
-generic "Top.A1.G1_0.I1.GEN = 5"
```

Usage of the -relax Option

The Xcelium utilities support a `-relax` switch, which when used with the `-generic` option, changes the behavior of `bit_vector` and `std_logic_vector` support. If the `-relax` option is used, and the size of the generic being passed with the `-generic` option does not match the size specified in the HDL code, Xcelium enables the generic override after truncating or padding the new value to match the size defined in the original HDL. IXCOM does not support a similar `-relax` option, and, therefore, the sizes specified in the HDL code and the generic override from the `ixcom` command line must match.

Assigning a Common Value to All Parameters with Specified Name

To assign values to Verilog and SystemVerilog parameters from the IXCOM command line, you can use the `-defparam` option by specifying the complete hierarchical path of the parameter. However, to specify a value for all objects that have the same name, you need to identify all of the hierarchical paths and specify the `-defparam` option multiple times. To avoid multiple specifications while assigning a value to all the parameters in the design with a specified name, you can use the `-gpg` option.

Note: You cannot use the `-gpg` option or the `-defparam` option to change the value of a parameter within protected code.

Using the `-gpg` option, you can specify the value in the following ways:

- Assigning Value by Name

© 2021 All Rights Reserved.

- Assigning Value by Instance and Name
- Assigning Value by Full Hierarchical Path

Note: For SystemVerilog (or Verilog) parameters, only the following value types are supported:

- Packed integral type
- Real
- String data type

Assigning Value by Name

To assign a value to all the parameters identified by a particular name, use the following syntax of the `-gpg` option:

```
-gpg "<object_name> => <value>"
```

Note: Unlike Xcelium, IXCOM support is limited to only simple constants in decimal format. The based, signed, and other formats used in Verilog are not supported.

For example, you can assign the value of `256` to all parameters by the name of `obj1` in the design as follows:

```
-gpg "obj1 => 256"
```

Assigning Value by Instance and Name

To assign a value to all the parameters identified by a particular name in all of the instances specified by a particular name, use the following syntax of the `-gpg` option:

```
-gpg "<instance_name>.<object_name> => <value>"
```

For example, you can assign the value of `2` to all parameters by the name of `p` in all of the instances named `blk` in the design as follows:

```
-gpg "blk.p => 2"
```

Assigning Value by Full Hierarchical Path

To assign a value to all the parameters specified by a particular hierarchical path, use the following syntax of the `-gpg` option:

```
-gpg "<hierarchical_pathname> => <value>"
```


© 2021 All Rights Reserved.

For example, you can assign the value of `2` to all parameters with the hierarchical pathname `Tb.dut.blk.p` in the design as follows:

```
-gpg "Tb.dut.blk.p => 2"
```

Note: The hierarchical path must be a legal name as allowed by the language. Unlike Xcelium, a hierarchical path starting with a forward slash (/) as starting symbol is not supported.

Examples of the Value Assignment Using the -gpg Option

In all of the above sub-sections, Assigning Value by Name, Assigning Value by Instance and Name, Assigning Value by Full Hierarchical Path, the value must be a simple constant value or a string. The value is interpreted as a string if it starts with a letter or if it is enclosed in quotes. Quotes are required for strings that do not start with a letter. Quoted strings must be escaped with backslash character.

Consider the following examples:

- Following is a correct specification of the integer value:

  ```
  -gpg "obj => 2"
  ```

- Following is a correct specification of the string value:

  ```
  -gpg "obj => abc"
  ```

- Following is a correct specification of the string value because the string does not start with a letter and quotes are required:

  ```
  -gpg "obj => \"555"\"
  ```

- Following is an incorrect specification of the string value because the quoted string is not escaped:

  ```
  -gpg "obj => "555""
  ```

In IXCOM, binaries are invoked through a shell wrapper in the release area. Therefore, all quoted strings must be specified using `-f <args_file>`.

Multiple formats can be used. The more granular the specification of the format, the higher the precedence. For example, if you use the following command:

```
> ixcom -gpg "obj => 2" -gpg "i.obj => 3" -gpg "Tb.dut.blk.i.obj => 4" ....
```

Then:

- Parameter specified by `Tb.dut.blk.i.obj` gets the value of `4`.
- All parameters named `obj` in the instance `i` get a value of `3`, except for `Tb.dut.blk.i.obj`.

© 2021 All Rights Reserved.

- All other parameters named `obj` get a value of `2`.

Values assigned with the `-defparam` option override the value assigned with the `-gpg` option. For example, with the following syntax, the value of `2` is assigned to all the parameters called `obj`, but the value of `9` is assigned to the parameter with the pathname `Tb.dut.blk.i.obj`.

```
-gpg "obj=>2" -defparam Tb.dut.blk.i.obj=9
```

Support for VHDL Packages to be Imported Into SystemVerilog Design Files

IXCOM supports import of VHDL packages in SystemVerilog design files like normal SystemVerilog packages. To import the VHDL package into SystemVerilog designs, perform the following steps:

1. Analyze VHDL packages using ***vhan***.
2. Analyze SystemVerilog files using ***vlan***.
3. Use ***vhan*** to analyze the VHDL portion of the design.
4. Elaborate the design database generated in the previous step using the `ixcom -elaborate` command.
5. Continue with IXCOM compilation as for a normal mixed-language design.

IXCOM supports only the following constructs in VHDL packages to be imported in SystemVerilog designs:

- Constant declaration
- Type declaration
- Subtype declaration

  Following are the data types supported in type and subtype declarations:

  - Record
  - Enum
  - Real
  - String without constraint
  - Integer

© 2021 All Rights Reserved.

- Natural
- Character
- Bit
- Boolean
- Std_logic
- Std_ulogic
- Bit vector
- Std_logic vector
- Std_ulogic vector
- Constrained arrays

- Following are the expressions supported for initial value of constant:
  - Unary expression
  - Binary expression
  - Object
  - Simple name
  - Selected name
  - Qualified expression
  - Conversion expression
  - Aggregate expression
  - Attributes names
    - Left
    - Right
    - High
    - Low
    - length
- Unconstrained arrays

© 2021 All Rights Reserved.

Unconstrained element array type is referred in Section 5.3.2.1 of IEEE STD 1076-2008 Language Reference Manual. Following is an example of such an array:

```
type slv_array is array (natural range <>) of std_logic_vector;
signal sig_c : slv_array(1 downto 0)(10 downto 0);
```

Following are the limitations in the support for unconstrained array type:

- OPEN is not supported.
- Only constant literals are supported in the constraints.
- Access type is not supported.
- Unconstrained arrays cannot be resolved type at composite level.
- Multi-dimensional unconstrained arrays are not supported.
- XMRs to this type from Verilog are not supported.

Limitations in Support for Importing VHDL Packages in SystemVerilog Designs

Following VHDL constructs are not supported:

- Unconstrained array
- Signal Declaration
- Shared variable declaration
- File declaration
- Alias declaration
- Component declaration
- Attribute declaration
- Attribute specification
- Disconnection specification
- Group template declaration
- Function Call
  - User-defined
  - System-defined

© 2021 All Rights Reserved.

- Operators
  - `rol`
  - `ror`
  - `abs`
  - `mem`
  - 200x Operators

HDL-ICE Compiler does not support importing of VHDL package in SystemVerilog design. So, IXCOM sends two versions of the same package to backened compilers.

- Original VHDL package
- Converted SystemVerilog package with modified package name, which is converted to uppercase and suffixed with `_VHDL_ZYYY`.

Keep the following points in mind while importing VHDL packages:

- Enumeration data type in VHDL can have same literals name in different declarations. But in SystemVerilog enumeration, literals are unique.
  - Enumeration literal name is changed during conversion. So, you need to use the changed name in SystemVerilog design. IXCOM does not convert the name in SystemVerilog design.

    Consider the following examples:

    - Here, `first` is converted to `ENUM_first`, where `ENUM` is the enumeration data type identifier.

      ```
      type ENUM is (first, second, third, fourth,fifth,default);
      ```

      Is converted to:

      ```
      typedef enum int
             {ENUM_first,ENUM_second,ENUM_third,ENUM_fourth,ENUM_fifth,
              ENUM_default } ENUM ;
      ```

    - Character literal `0` is converted to `anotherEnum_c0`.

      ```
      type anotherEnum is ('0', '1', L, H);
      ```

      Is converted to:

      ```
      typedef enum int
             {anotherEnum_c0,anotherEnum_c1,anotherEnum_L,
              anotherEnum_H } anotherEnum ;
      ```


© 2021 All Rights Reserved.

- You can use SystemVerilog keywords in VHDL package. But these are changed to escaped identifiers. So, you need to use escaped identifiers in SystemVerilog design to access these.

  ```
  type endpackage is (AA,BB,CC);
  ```

  Is converted to:

  ```
  typedef enum int {endpackage_AA,endpackage_BB,endpackage_CC } \
                    endpackage ;
  ```

- VHDL package names are case insensitive. But SystemVerilog package names are case-sensitive. Converted SystemVerilog packages will also be case-sensitive. This means that the identifiers and names used in declarations should be used in the same case as used in the SystemVerilog design.
- System-defined function calls and some of the operators are supported by Xcelium; but not by IXCOM.
- Representation of some of the VHDL constructs in SystemVerilog is not supported. For example, parameters with data types as associative arrays and unpacked arrays. For all such constructs, errors are generated during elaboration.

Support for specify Blocks in SA Mode

The `specify` block is used to describe the module path delays and timing checks in a Verilog design. To ensure that the `specify` block is supported, use the `+specblk` option at the command line. To enable support for `specify` block in SA mode, use the `+sv` option with the `ixcom` command. For example:

```
ixcom +sv +specblk test.v
```

The `specify` blocks are supported only for testbench modules. IXCOM ignores the `specify` blocks for modules that are mapped to the hardware.

Timing checks can be specified within the `specify` block to ensure that critical events, such as transitioning of the clock and data signals, occur with the specified time duration. Timing checks, if included in a `specify` block are used to verify the timing performance of the DUT.

The syntax of the `specify` block is:

```
specify_block ::=
   specify { specify_item } endspecify
specify_item ::=
   specparam_declaration |
   pulsestyle_declaration |
   showcancelled_declaration |
   path_declaration |
   system_timing_check
```


© 2021 All Rights Reserved.

The `specify` block consists of the `specify..endspecify` block, within which you can provide various declarations and timing checks. The following declarations can be specified within the `specify..endspecify` block:

- `specparam`
- `pathpulse specparam`
- `pulsestyle` and `showcancelled`
- Module Path
- System-timing Checks

The `specify` block and delayed signals can be included either in DUT or testbench modules or in both.

Using specify Blocks in a DUT Module

If a `specify` block is included in a module that has been specified as a DUT module using the `+dut+<module name>` option, this `specify` block (except `specparams`) is ignored even if the `+specblk` option is provided and an appropriate warning message is generated. If the `+specblk` option is not specified, the `specify` block construct along with `specparams` is ignored.

Following is the command that needs to be used when a `specify` block is included in the DUT:

```
ixcom +sv +specblk +dut+<mod_name> <testbench.v_files>
```

If the module contains extended timing checks, continuous assignments driving the delayed signal using the original signal are created and appropriate information is generated. In the following example, the delayed signals, *dCLK* and *dip* will be driven by *CLK* and *ip*, respectively.

```
$setuphold (posedge CLK, posedge ip, -5, 10, , , , dCLK, dip);
```

Using specify Block in a Testbench Module

If a `specify` block construct is included in a testbench module and the `+specblk` option is specified, the `specify` block is processed. If the `+specblk` is not specified, the `specify` block is ignored with a warning.

If the module contains extended timing checks, and the `+specblk` option is not specified, continuous assignments are created to drive the delayed signal using the original signal. If

© 2021 All Rights Reserved.

the `+specblk` option is specified, the `specify` block is processed and no continuous assignments are created.

Deviations from Verilog 2005 LRM for specify Blocks

The `specify` block constructs are supported in SA mode, but with certain deviations from the Verilog 2005 LRM. The following semantic checks or constraints as specified by the LRM are ignored by:

- `pulseparam` declaration:
  - The error limit (if specified explicitly) should be equal to or greater than the reject limit.
  - The error limit (if specified explicitly) should be less than the delays specified in a module path.
- Module path declaration:
  - The module path destination can have only one driver (or no driver) inside the module.
  - The `pulsestyle` declaration (if specified) should precede the module path declaration.
- `showcancelled` declaration:

  The showcancelled declaration for a particular destination should always precede its module path declaration. If the module path appears before a `showcancelled` declaration, an error message is displayed.

Support for Compressed Files

IXCOM supports files that have been compressed using the gzip and compress utilities. IXCOM determines the language or HDL type according to the specified file extension after removing the `.gz` and `.Z` suffixes.

For example, consider a file, such as `vFile.sv`. After compression by using the gzip utility (`gzip vFile.sv`), this file is saved as `vFile.sv.gz`. When you specify this `vFile.sv.gz` file with the `ixcom` command, IXCOM treats this as a SystemVerilog file named `vFile.sv`.

> **Note:** One compressed file should contain only one source file.
> Names of compressed files must end with `.gz` or `.Z`. If IXCOM is unable to decompress a source file, it issues an error.

© 2021 All Rights Reserved.

Following is the format that IXCOM considers for the names of the compressed files:

<filename>.<default_file_extension>.<compressed_file_extension>

Here, *<filename>* must be a valid name. The *<default_file_extension>* can be any string. The *<compressed_file_extension>* can be `.gz` or `.Z`. All other extensions are treated as Verilog file extensions.

For compressed file support, the HDL file extensions that you specify using options, such as the `-vlog_ext -sysv_ext` options are treated in the same way as built-in HDL file extensions.

For example, the filename `vFile.k.gz` is treated as `vFile.k`.

By default, IXCOM does not know the HDL type (`v97, v2k, or SV`, and so on of the `vFile.k` file. Therefore, it cannot determine the HDL type for this file. If you specify `-vlog_ext .k`, IXCOM recognizes `vFile.k` as a Verilog file, and, hence, it regards `vFile.k.gz` as a Verilog file.

IXCOM can handle and search compressed files while searching a file in the library file specified with the `-v` or `-y` option.

IXCOM can decompress protected files and netlist files. Only HDL files (VHDL, Verilog, SystemC, and netlist) are supported for decompression. Any other files, such as an argument file or a feature file should be plain text files; not compressed files.

IXCOM cannot support compressed file with the `xrun -hw` command.

You can pass compressed files to IXCOM in single-step and multi-step modes.

Consider the following example for single-step mode:

```
% ixcom test.v.gz dut.Z
% ixcom test.vvv.gz -vlog_ext .vvv #this changes the default Verilog extension to
vvv
```

Consider the following example for multi-step mode:

For analyzing Verilog files:

```
vlan test.v.gz dut.Z
vlan test.vvv.gz -vlog_ext .vvv #this changes the default Verilog extension to vvv
```

For analyzing VHDL files:

```
vhan top.vhd.gz dut.vhd.z bot.vhdl.gz foo.vhdl.Z
```


© 2021 All Rights Reserved.

Enabling Run-time Access of For-generate Blocks and Instance Arrays Using Original RTL Names

By default, during elaboration, IXCOM unrolls the for-generate blocks and instance arrays written in the original design files. This transformation changes the hierarchical address of the design element present inside the for-generate block because IXCOM modifies the name of the block by appending underscores in place of the square brackets. So, IXCOM generates `<module_name>_0_` instead of `<module_name>[0]` for Verilog, and `<<module_name>_0` instead of `<module_name>(0)` for VHDL.

Following are example of these conversions:

- Verilog examples
  - IXCOM converts `bank_ram_gen0[0]` to `bank_ram_gen0_0_0`.
  - IXCOM sets the hierarchy path as `enet_dut.InstDUT_0_.reg_if` instead of `enet_dut.InstDUT[0].reg_if`.
- VHDL examples
  - IXCOM converts `bank_ram_gena(0)` to `bank_ram_gena_0`.
  - IXCOM sets the hierarchy path of design element `enet_dut.InstDut(1).reg_sig` to `enet_dut.InstDut_1.reg_sig`.

Using the -rtlNameForGenerate option for Run-time Access

At run time, the design elements cannot be accessed with their original-RTL names because of this change in the name. To avoid these access issues, use the `-rtlNameForGenerate` option of the `ixcom` command.

Note: If you do not use the `-rtlNameForGenerate` option, you must manually add the underscore suffix in the hierarchical names in VHDL designs along with the integral value of the index. Else, the design cannot be compiled.

With the `-rtlNameforgen`erate option, local OOMRs to variables declared within for-generate of complex datatypes, such as struct and union, and declared within a package or compilation unit scope, are not supported.

Known Issues and Limitations of Using the -rtlNameForGenerate option:

- The `-rtlNameForGenerate` option cannot be used along with the `+hmpi` option.

© 2021 All Rights Reserved.

- While using the `xmdc` utility to decompile the simulation snapshot, note that it does not show the rolled up versions of the for-generate and instance arrays. For details about the `xmdc` utility, refer to the *Using the Xcelium Utilities* manual available with the Xcelium documentation set.
- Source Browser in SimVision shows the unrolled versions of the for-generate and instance arrays. Design Browser, however, shows only the rolled up instances of for-generate and instance arrays and not the unrolled versions.
- Escaped names cannot be probed during debug.

Reporting Delays in Initial Blocks

The `initial` blocks in the DUT are executed in software before the design is swapped to hardware. Use the `+reportInitDelays` option of the `ixcom` command to specify that the `initial` blocks are analyzed for delay constructs and these delay constructs are reported in the `ixcom.log` file.

When any of the delay constructs are identified, the following kind of warning message is reported in the `ixcom.log` file:

```
Warning! in file test_ixcom.sv at line 16 [TIME_CONSUMING_INITIAL_BLK]
Time consuming initial block exist in the module dut
```

This warning is repeated for each delay element found in the `initial` blocks.

Following are some examples of time-consuming constructs in DUT initial blocks that can prevent swap-in to hardware before these constructs finish execution in software:

- #delay:

```
initial begin
    #10 a = b;
end
```

- Event Control:

```
initial begin
    while(counter<5) begin:
        @(negedge clk);
            Toggle = ~toggle;
    end
end
```

- `wait` statement:

```
initial begin
    wait(testbench.waitSignal);
        counter = counter + 1;
end
```

- Task call:

© 2021 All Rights Reserved.

```
initial begin
    task_called(); //Task is called
end
```

- #delay in intra-assignment:

```
initial begin
        L = #10 R;
end
```

- Cycle delay:

```
initial begin
    ##5 count = count + 1;
end
```

- Cycle delay in intra-assignment:

```
initial begin
    L = ##5 R;
end
```

- Event control in intra-assignment statement:

```
initial begin
    L = @(negedge clk)R;
end
```

- Intra assignment with repeat event:

```
initial begin
    counter = repeat(2) @(toggle)count;
end
```

- `fork...wait`:

```
initial begin
    fork
    counter = counter + 1;
    toggle = ~toggle;
    join_any
    wait fork;
end
```

- `fork...join`:

```
initial begin
    fork
    #10 statement 1
    statement 2
    join
end
```

- `fork...join_any`:

```
initial begin
    fork
    #10 statement1
    #5 statement2
    statement3
    #4 toggle = ~toggle;
```


© 2021 All Rights Reserved.

```
    join_any
end
```

Or:

```
initial begin
    fork
        #10 statement1
        #5 statement2
        statement3
    join_any
end
```

The following constructs are ignored and not reported:

- `fork...join_none`:

```
initial begin
    fork
    #10 statement1
    #5 statement2
    statement3
    join_none
end
```

- `#0` in `initial` block:

```
initial begin
    #0 statement1
    #1 statement2
    statement3
end
```

  Note: Only statement 2 will be reported from the `initial` block and statement 1 will not be reported.

- The `wait_order` statement is not supported.
- Non-blocking statements with intra-assignment delay constructs are not blocking so these are not reported as warning messages:
  - Intra-assignment delay:

    ```
    expression1 <= #delay expression2
    ```

  - Cycle delay in intra-assignment:

    ```
    expression1 <= ##2 expression2;
    ```

  - Intra-assignment waiting for some event:

    ```
    expression1 <= @(signal) expression2;
    ```

  - Intra-assignment waiting for repetition of the event to get completed:

    ```
    expression1 <= repeat(2) @(signal) expression2;
    ```

- `fork...join_any` case:

© 2021 All Rights Reserved.

```
initial begin
    fork
        #5 counter = 0;
    fork
        counter1 = 10;
    join
    join_any
end
```


© 2021 All Rights Reserved.

Direct Access to DUT Memory Arrays and Registers

IXCOM provides a built-in `marg` library to enable direct access to the following objects in the DUT:

- 2-D memory arrays
- Scalar registers

The following sections outline:

- Using the marg Library
- C and Verilog Interface marg Functions
- SystemVerilog DPI marg Functions
- Specman marg Functions

Using the marg Library

Following are the key points about the `marg` library objects:

- A `marg` object handle is created dynamically by path.
- For C-based testbenches, built-in C functions are used to create, delete, and directly access a `marg` object.
- For Verilog-based testbenches, built-in `$ixc_t` and `$ixc_f` interface functions are used when compiling with IXCOM. Note that these functions are supported in the testbench partition only when the entire design (DUT and testbench) has Verilog-only files and is compiled only with IXCOM. If the testbench partition needs to be compiled separately using ***xrun***, `$ixc_t` or `$ixc_f` functions should not be used. Instead, use the SystemVerilog DPI functions when the testbench needs to be compiled separately.

For each C interface function, there is an equivalent Verilog task or function.

Using marg Interface in Pure Simulation

When using the `marg` interface in pure simulation, the <*IXCOM_INSTALL_HOME*>`/share/vxe/etc/ixcom/marg_sim.c` file must be compiled as a DPI application to the *xrun* simulation.

Following are the two ways to do this:

© 2021 All Rights Reserved.

- Provide the `marg_sim.c` file and the design files to the `xrun` command.

  For example:

```
xrun -access \
     +rw test.sv \
     -I <IXCOM_INSTALL_HOME>/share/vxe/etc/ixcom \
     -cpost \
     <IXCOM_INSTALL_HOME>/share/vxe/etc/ixcom/marg_sim.c test.c \
     -end
```

- Create a shared library and then simulate with the shared library.

  For example:

```
gcc -I <IXCOM_INSTALL_HOME>/share/vxe/etc/ixcom \
    -I <IXCOM_INSTALL_HOME>/tools/include \
    -fPIC -shared -Wall \
    -o libmarg.so \
    <IXCOM_INSTALL_HOME>/share/vxe/etc/ixcom/marg_sim.c
xrun -R \
     -sv_lib ./libmarg.so
```

Note that compiling the `marg_sim.c` file in the above-mentioned manner is applicable only for C testbenches, not Verilog testbenches.

Linking marg and SCE-MI Applications in a Single Pure Simulation

The pure-simulation library for SCE-MI is `<IXCOM_INSTALL_HOME>/tools.lnx86/lib/64bit/libuatba_dc.so`, which already contains the `marg_sim.c` file. If the design uses both `marg` and SCE-MI, do not include the `marg_sim.c` file. Just use the `libuatba_dc.so` file in simulation.

C and Verilog Interface marg Functions

This section describes the `marg` interface functions provided for use in C and Verilog testbenches. The marg C interface functions are located at `<IXCOM_INSTALL_HOME>/share/vxe/etc/ixcom/marg_user.h`. In general, most `marg` functions return 0 if the function call is successful. A returned value of non-zero indicates that an error has occurred during the function call. However, the returned value 0 from all the `marg` "get handle" functions is not an error. Rather, a negative returned value from these "get handle" functions indicates an error. These "get handle" functions are `marg_new`, `marg_vmem_handle`, `marg_deposit_handle`, and `marg_value_handle`.

© 2021 All Rights Reserved.

Tasks for Memory-Array Buffer

The C and Verilog functions of the `marg` interface can be used to perform the following tasks for memory-array buffer:

Creating a Memory-Array Buffer

To create a memory-array buffer of the given width and depth, use the following functions:

- C function:

  ```
  int ma_handle = marg_new(int width, int depth);
  ```

  The `marg_new` function returns a negative number if there is an error.

- Verilog function:

  ```
  int ma_handle = $ixc_f("marg_new", width, depth);
  ```

Deleting a Memory-Array Buffer

To delete a memory-array buffer, use the following functions:

- C function:

  ```
  void marg_delete(int ma_handle);
  ```

- Verilog function:

  ```
  $ixc_t("marg_delete", ma_handle);
  ```

Writing a Memory-Array Buffer at a Given Address

To write to a memory-array buffer at the given address, use the following functions:

- C function:

  ```
  int marg_write(int ma_handle, int addr, unsigned *value);
  ```

- Verilog function:

  ```
  int errno = $ixc_f("marg_write", ma_handle, addr, value);
  ```

Reading a Memory Buffer at a Given Address

To read from a memory buffer at the given address, use the following functions:

- C function:

  ```
  int marg_read(int ma_handle, int addr, unsigned *value);
  ```


© 2021 All Rights Reserved.

Note: The address for `marg_write` and `marg_read` is the address of the `marg` buffer starting from 0. You have not associated this `marg` address with any DUT memory yet. A single `marg` buffer can be used to transfer data to multiple DUT memories at multiple locations.

- Verilog function:

```
int errno = $ixc_f("marg_read", ma_handle, addr, value);
```

Note: The address for `marg_write` and `marg_read` is the address of the `marg` buffer starting from 0. You have not associated this `marg` address with any DUT memory yet. In fact, one `marg` buffer can be used to transfer data to multiple DUT memories at multiple locations.

Writing a Block of Data to a Memory Buffer

To write a block of data to a memory buffer, use the following functions:

- C function:

```
int marg_write_block(int ma_handle, int addr, int len, unsigned* value);
```

Where,

  - `ma_handle` identifies the `marg` buffer used for writing or reading. The width of the `marg` buffer is already initialized when `marg_new` is used to get the `marg` buffer handle `ma_handle`.
  - `len` specifies the number of marg words to be transferred.
  - `value` for `marg_write_block` is expected to be packed to the byte boundary of a marg word. The second word of `value` should be packed to the next byte immediately following the first word. Likewise, the returned `value` from `marg_read_block` is packed with the same fashion.

In the following example, the first marg word is filled with content `"0"` from the first three bytes of `value`, The second marg word is filled with content `"1"` from the next three bytes of `value`.

```
unsigned char value[24];
int mah = marg_new(20, 8);  // each marg word is 20 bits (3 bytes)
int i;
for(i = 0; i < 8; i++)
     value[i*3] = i;
marg_write_block(mah, 0, 8, (unsigned*)value);
```

- Verilog function:

```
int errno = $ixc_f("marg_write_block", ma_handle, addr, len, value);
```


© 2021 All Rights Reserved.

Reading a Block of Data from a Memory Buffer

To read a block of data from a memory buffer, use the following functions:

- C function:

  ```
  int marg_read_block(int ma_handle, int addr, int len, unsigned* value);
  ```

  Note: The address for `marg_write_block` and `marg_read_block` is the address of the `marg` buffer starting from 0. You have not associated this `marg` address with any DUT memory yet. A single `marg` buffer can be used to transfer data to multiple DUT memories at multiple locations.

- Verilog function:

  ```
  int errno = $ixc_f("marg_read_block", ma_handle, addr, len, value);
  ```

Obtaining a DUT Memory Handle by Path

To obtain a DUT memory handle by path, use the following functions:

- C function:

  ```
  int vm_handle = marg_vmem_handle(char *vmem_path);
  ```

  The `marg_vmem_handle` function returns a negative number if there is an error.

- Verilog function:

  ```
  int vm_handle = $ixc_f("marg_vmem_handle", vmem_path);
  ```

Getting the Width and Depth of DUT Memory

To get the width (in bits) and depth of a DUT memory, use the following functions:

- C function:

  ```
  int marg_vmem_size(int vm_handle, int* width, int* depth);
  ```

- Verilog function:

  ```
  int errno = $ixc_f("marg_vmem_size", vm_handle, width, depth);
  ```

Transfering Memory-Array Buffer to DUT Memory

To transfer a memory-array buffer (`ma_handle`) to a DUT memory (`vm_handle`), use the following functions:

- C function:

© 2021 All Rights Reserved.

```
int marg_put(int ma_handle, int ma_addr, int len, int vm_handle,
             int vm_offset);
```

- Verilog function:

```
int errno = $ixc_f("marg_put", ma_handle,  ma_addr, len, vm_handle, vm_offset);
```

Transfering DUT Memory to Memory-Array Buffer

To transfer a DUT memory (`vm_handle`) to a memory-array buffer (`ma_handle`), use the following functions:

- C function:

```
int marg_get (int ma_handle, int ma_addr, int len, int vm_handle,
              int vm_offset);
```

Note: For `marg_put` and `marg_get`, the starting address of the `marg` block is `ma_addr` and the destination offset of the DUT memory is `vm_offset`, which is the normalized DUT memory address. For example, if the DUT memory is `reg[31:0] mem [4:16]` or `reg[31:0] mem [16:4]`, the `vm_offset` for `mem[4]` is 0.

- Verilog function:

```
int errno = $ixc_f("marg_get, ma_handle, ma_addr, len, vm_handle, vm_offset);
```

Note: For `marg_put` and `marg_get`, the starting address of the `marg` block is `ma_addr` and the destination offset of the DUT memory is `vm_offset`, which is the normalized DUT memory address. For example, if the DUT memory is `reg[31:0] mem[4:16]` or `reg[31:0] mem[16:4]`, the `vm_offset` for `mem[4]` is 0.

Loading a DUT Memory from a File

To load a DUT memory from a data file, use the following functions:

- C function:

```
extern int marg_load_vmem(int vm_handle, const char* file, int offset,
       int len, const char* format, int flags);
```

- Verilog function:

```
$ixc_t("marg_load_vmem", vm_handle, file, offset, len, format, compress);
```

Dumping a DUT Memory to a File

To dump a DUT memory to a data file, use the following functions:

- C function:

© 2021 All Rights Reserved.

```
extern int marg_dump_vmem(int vm_handle, const char* file, int offset,
                          int len, const char* format, int flags);
```

Note: For `marg_load_vmem` and `marg_dump_vmem`, the offset is the normalized starting address of the DUT memory. The file format can be `%readmemh`, `%readmemb`, `%h`, `%b`, `%o`, `%pd_raw`, and `%pd_raw2`, `%pd_b`, `%pd_o`, `%pd_d`, `%pd_h`, or `%pd_memtran`. For all interfaces, the last argument name for `marg_dump_vmem` and `marg_load_vmem` should be `flags`. The `flags` argument can be `0` or OR'ed (|) of `MARG_Z`, `MARG_SKIPZERO`, `MARG_APPEND`, or `MARG_DONTPRINTOFFSET`:

- `MARG_Z`: generate compressed `PD raw` or `raw2` format.
- `MARG_SKIPZERO`: skip all-zero words in `PD ASCII` format.
- `MARG_APPEND`: append to the dump file in `PD raw2` format.
- `MARG_DONTPRINTOFFSET`: don't print the `@` offset line in `readmem` format

For example, to append to an existing compressed `pd_raw2` file named `my_file`:

```
marg_dump_vmem(id, "my_file", 0, 32, "%pd_raw2", MARG_Z | MARG_APPEND);
```

- Verilog function:

```
$ixc_t("marg_dump_vmem", vm_handle, file, offset, len, format, compress);
```

Note: For `marg_load_vmem` and `marg_dump_vmem`, the offset is the normalized starting address of the DUT memory. The file format can be `%readmemh`, `%readmemb`, `%h`, `%b`, `%o`, `%pd_raw`, and `%pd_raw2`, `%pd_b`, `%pd_o`, `%pd_d`, `%pd_h`, or `%pd_memtran`. For all interfaces, the last argument name for `marg_dump_vmem` and `marg_load_vmem` should be `flags`. The `flags` argument can be `0` or OR'ed (|) of `MARG_Z`, `MARG_SKIPZERO`, `MARG_APPEND`, or `MARG_DONTPRINTOFFSET`:

- `MARG_Z`: generate compressed `PD raw` or `raw2` format.
- `MARG_SKIPZERO`: skip all-zero words in `PD ASCII` format.
- `MARG_APPEND`: append to the dump file in `PD raw2` format.
- `MARG_DONTPRINTOFFSET`: don't print the `@` offset line in `readmem` format

Setting the Same Value to a DUT Memory

To set the same value to a DUT memory, use the following functions:

- C function:

```
int marg_set_vmem(int vm_handle, int offset, int len, unsigned* value);
```

- Verilog function:

```
$ixc_t("marg_set_vmem", vm_handle, offset, len, value);
```


© 2021 All Rights Reserved.

Tasks for DUT Register Access

The C and Verilog functions of the `marg` interface can be used to perform the following tasks for DUT register access:

Obtaining a Handle of an Exported Register for Deposit

To obtain a handle of an exported register for deposit, use the following functions:

- C function:

  ```
  int marg_deposit_handle(const char* vsig_path);
  ```

  Note: `vsig_path` must be exported by `$export_deposit` in IXCOM. The `marg_deposit_handle` function returns a negative number if there is an error.

- Verilog function:

  ```
  int dp_handle = $ixc_f("marg_deposit_handle", vsig_path);
  ```

  Note: `vsig_path` must be exported by `$export_deposit` in IXCOM. `marg_deposit_handle` returns a negative number if there is an error.

Depositing a Value to an Exported Register

To deposit a value to an exported register, use the following functions:

- C function:

  ```
  int marg_deposit(int dp_handle, unsigned* value);
  ```

- Verilog function:

  ```
  int errno = $ixc_f("marg_deposit", dp_handle, value);
  ```

Obtaining a Handle of an Exported Signal for Read

To obtain a handle of an exported signal for read, use the following functions:

- C function:

  ```
  int marg_value_handle(const char* vsig_path);
  ```

  Note: `vsig_path` must be exported by `$export_read` in IXCOM. The `marg_value_handle` function returns a negative number if there is an error.

- Verilog function:

  ```
  int handle = $ixc_f("marg_value_handle", vsig_path);
  ```

  Note: `vsig_path` must be exported by `$export_read` in IXCOM.

© 2021 All Rights Reserved.

`marg_value_handle` returns a negative number if there is an error.

Getting the Value of an Exported Signal for Read

To get the value of an exported signal for read, use the following functions:

- C function:

  ```
  int marg_value(int handle, unsigned* value);
  ```

  Note: The `marg` functions `marg_new`, `marg_vmem_handle`, and `marg_deposit_handle` return an integer. A handle of zero is a legal handle. If there is an error during the function call, a negative number is returned.

- Verilog function:

  ```
  int errno = $ixc_f("marg_value", handle, value);
  ```

  Note: The `marg` functions `marg_new`, `marg_vmem_handle` and `marg_deposit_handle` return an integer. A handle of zero is a legal handle. If there is an error during the function call, a negative number is returned.

Obtaining a Handle of an Exported Register or Wire for Force or Release

To obtain a handle of an exported register or wire for force or release, use the following C function:

```
int marg_frcrel_handle(const char* path);
```

Note: The `path` must be exported by using `$export_frcrel` in IXCOM. The `marg_frcrel_handle` returns a negative number if there is an error.

Forcing a Value to an Exported Register or Wire for Force or Release

To force a value to an exported register or wire, use the following function:

```
int marg_force(int fr_handle, unsigned* data);
```

Releasing an Exported Register or Wire that was Previously Forced

To release an exported register or wire, use the following function:

```
int marg_release(int fr_handle);
```

To issue an `xc` command from C, use the following C function:

```
int marg_xc_cmd(const char* cmd);
```


© 2021 All Rights Reserved.

Note: Do not prefix the `cmd` string with `xc`. For example, to issue `xc on -tbrun`, specify the following command:

```
marg_xc_cmd("on -tbrun");
```

To get the value of the full path of a parameter in string form, use the following:

```
const char* marg_get_param(const char* path, const char* format);
```

The `%s` format is recommended for string parameters.

For example, `parameter p = "abcd"` with `%s` will return `"abcd"`.

Use other formats for non-string parameters.

For example, `parameter p = 'habcd` with `%h` will return `"0000abcd"`.

Examples: Passing Memory Data from Testbench to DUT

Sample code for the DPI-exported task in DUT to which the testbench needs to pass the memory data:

```
module DUT
    reg [31:0] lm[511:0];
    task dpi_export_task;
       begin
       <
             Use lm[addr] to process TB data;
       >
       end
    endtask
    endmodule
```

Sample code for passing memory data from a C-based testbench to the DPI-exported task in DUT (DUT.lm):

```
#include "marg_user.h"
int ma = -1;
int vm = 0;
void c_task(){
    int i;
    if(ma < 0){
        ma = marg_new(32, 512);
        vm = marg_vmem_handle("DUT.lm");
    }
    for(i=0; i< 512; i++){
       < generate some value and fill the marg>
       marg_write(ma, i, value);
    }
    marg_put(ma, 0, 512, vm, 0);
    dpi_export_task(); //call dpi_exported task;
    marg_get(ma, 0, 512, vm, 0);
}
```


© 2021 All Rights Reserved.

Sample code for passing memory data from a Verilog testbench to the DPI-exported task in DUT (DUT.lm):

```
module TB;
    integer ma, vm;
       initial begin
          ma = $ixc_f("marg_new", 32, 512);
          vm = $ixc_f("marg_vmem_handle", DUT.lm);
       end
       task v_task;
          begin
             for(i=0; i< 512; i++)
             begin
                <generate some values and fill the marg>
                $ixc_t("marg_write", ma, i, value);
             end
             $ixc_t("marg_put", ma, 0, 512, vm, 0);
             DUT.dpi_export_task();
             $ixc_t("marg_get", ma, 0, 512, vm, 0);
          end
       endtask
    endmodule
```

SystemVerilog DPI marg Functions

This section describes the `marg` interface functions provided for use in SystemVerilog testbenches. The `marg` SV DPI functions are located at `<IXCOM_INSTALL_HOME>/share/vxe/etc/ixcom/marg_user.svh`.

The value parameter for the `marg_deposit`, `marg_value`, `marg_write`, `marg_read`, and `marg_set_vmem` functions cannot be an open array (as required by SystemVerilog DPI). So, you must generate these import functions by `` `MARG_IMPORT `` before using them. You can generate these import functions by `` `MARG_IMPORT(N) ``, where `N` can be a Verilog parameter or a constant integer. Then, call these functions by appending `"_N"` to the function name.

For example:

```
parameter vecw = 64;
parameter memw = 32;
`MARG_IMPORT(vecw)
`MARG_IMPORT(memw)
bit [vecw-1: 0] dp_value;
bit [memw-1: 0] ma_value;
if( marg_deposit_vecw(dp_id, dp_value)!=0 )
    $display("failed in marg_deposit");
if( marg_read_memw(ma_id, offset, ma_value)!=0 )
    $display("failed in marg_read");
void'( marg_write_memw(ma_id, offset, ma_value) );
```

Following is a description of these functions:

- To create a `marg` object of the given width and depth, use:

© 2021 All Rights Reserved.

```
int ma_handle = marg_new(int width, int depth);
```

The `marg_new` returns a negative number if there is an error.

- To delete a `marg` object, use:

```
void marg_delete(int ma_handle);
```

- To obtain a Verilog memory handle by path, use:

```
int vm_handle = marg_vmem_handle(string vmem_path);
```

The `marg_vmem_handle` returns a negative number if there is an error.

- To get the width (in bits) and depth of a Verilog memory, use:

```
int marg_vmem_size(int vm_handle, output int width, output int depth);
```

- To write to a `marg` object at the given address, use:

```
int marg_write_N(int ma_handle, int addr, bit[N:1] value);
```

- To read from a `marg` object at the given address, use:

```
int marg_read_N(int ma_handle, int addr, output bit[N:1] value);
```

Note: The address for `marg_write` and `marg_read` is the address of the marg buffer starting from 0. This `marg` is not associated with any Verilog memory. One `marg` buffer can be used to transfer data to multiple Verilog memories at multiple locations.

- To transfer a block of data of the specified length (`len`) from a `marg` object (`ma_handle`) to a Verilog memory in DUT (`vm_handle`), use:

```
int marg_put (int ma_handle, int ma_addr, int len, int vm_handle,
              int vm_offset);
```

Note: The starting address of the `marg` block is `ma_addr` and the destination offset of the Verilog memory is `vm_offset`, which is the normalized Verilog memory address. For example, if the Verilog memory is `reg [31:0] mem [4:16]` or `reg[31:0] mem [16:4]`, the `vm_offset` for `mem[4]` is 0.

- To transfer a block of data of the specified length (`len`) from a Verilog memory in DUT (`vm_handle`) to a `marg` object (`ma_handle`), use:

```
int marg_get (int ma_handle, int ma_addr, int len, int vm_handle,
              int vm_offset);
```

Note: For `marg_put` and `marg_get`, the starting address of the `marg` block is `ma_addr` and the destination offset of the Verilog memory is `vm_offset`, which is the normalized Verilog memory address. For example, if the Verilog memory is `reg [31:0] mem [4:16]` or `reg[31:0] mem [16:4]`, the `vm_offset` for `mem[4]` is 0.

- To load a Verilog memory from a data file, use:

```
int marg_load_vmem(int vm_handle, string file, int offset, int len,
                   string format, int compress);
```


© 2021 All Rights Reserved.

- To dump a Verilog memory to a data file, use:

```
int marg_dump_vmem(int vm_handle, string file, int offset, int len,
                   string format, int compress);
```

Note: For `marg_load_vmem` and `marg_dump_vmem`, the offset is the normalized starting address of the Verilog memory. The file format can be `%readmemh`, `%readmemb`, `%h`, `%b`, `%o`, `%pd_raw`, and `%pd_raw2`. The compress argument is 0 or 1 which you can use to specify the compression used in the raw data file.

- To uniformly set the same data to a Verilog memory, use:

```
int marg_set_vmem_N(int vm_handle, int offset, int len, bit[N:1] value);
```

- To obtain a handle for depositing an exported hardware signal, use:

```
int marg_deposit_handle(string vsig_path);
```

Note: The `vsig_path` must be exported by `$export_deposit` in IXCOM. The `marg_deposit_handle` returns a negative number if there is an error.

- To deposit a value to a deposit handle, use:

```
int marg_deposit_N(int dp_handle, bit[N:1] value);
```

- To obtain a handle for getting an exported hardware signal value, use:

```
int marg_value_handle(string vsig_path);
```

Note: The `vsig_path` must be exported by `$export_read` in IXCOM. The `marg_value_handle` returns a negative number if there is an error.

- To get a value of a value handle, use:

```
int marg_value_N(int handle, output bit[N:1] value);
```

Note: The `marg` functions `marg_new`, `marg_vmem_handle`, and `marg_deposit_handle` return an integer. A handle of zero is a legal handle. If there is an error during the function call, a negative number is returned. All other `marg` functions return 0 if the function call is successful. A returned value of non-zero indicates that an error has happened during the function call.

- To obtain a handle of an exported register or wire for force or release, use:

```
int marg_frcrel_handle(string path);
```

Note: The `path` must be exported by using `$export_frcrel` in IXCOM. The `marg_frcrel_handle` returns a negative number if there is an error.

- To force a value to an exported register or wire, use:

```
int marg_force_N(int fr_handle, bit[N:1] value);
```

- To release an exported register or wire, use:

```
int marg_release(int fr_handle);
```


© 2021 All Rights Reserved.

- To issue an `xc` command from C, use:

  ```
  int marg_xc_cmd(string cmd);
  ```

  Note: Do not prefix the `cmd` string with `xc`. For example, to issue `xc on -tbrun`, specify the following command:

  ```
  marg_xc_cmd("on -tbrun");
  ```

- To get the value of the full path of a parameter in string form, use the following:

  ```
  string marg_get_param(string path, string format);
  ```

  The `%s` format is recommended for string parameters.

  For example, `parameter p = "abcd"` with `%s` will return `"abcd"`.

  Use other formats for non-string parameters.

  For example, `parameter p = 'habcd` with `%h` will return `"0000abcd"`.

- To enable accepting data of any widths, use:

  ```
  import "DPI-C" function int marg_oba_read(int ma_handle, int addr,
                                            output bit [] data);
  import "DPI-C" function int marg_oba_write(int ma_handle,
                                             int addr, bit [] data);
  import "DPI-C" function int marg_oba_deposit(int dp_handle, bit [] data);
  import "DPI-C" function int marg_oba_value(int vl_handle, output bit [] data);
  int marg_oba_read(int ma_handle, int addr, output bit [] data);
  int marg_oba_write(int ma_handle, int addr, bit [] data);
  int marg_oba_deposit(int dp_handle, bit [] data);
  int marg_oba_value(int vl_handle, output bit [] data);
  int marg_oba_force(int fr_handle, bit [] data);
  ```

  The caller must provide a SystemVerilog bit array to pass the data argument. It cannot be a Verilog reg array or wire.

  These marg interface functions perform the same functions as other marg functions with similar names except that these accept data arguments in different formats.

  For example:

  - `marg_oba_read` is similar to `marg_read_N`
  - `marg_oba_write` is similar to `marg_write_N`
  - `marg_oba_deposit` is similar to `marg_deposit_N`
  - `marg_oba_value` is similar to `marg_value_N`

  Here, `oba` stands for Open Bit Array. The `oba` version accepts SystemVerilog packed bit array of any size in its data. The `N` version accepts any Verilog packed array of size `N` in its data. The SystermVerilog LRM standard for passing DPI arguments is followed.

  For example:

© 2021 All Rights Reserved.

```
bit [31:0] a;
bit [63:0] b;
reg [31:0] c;
wire [31:0] d;
```

The `marg_oba_write` only accepts `a`, `b`.

The `marg_write_32` only accepts `a`, `c`, and `d`.

Specman marg Functions

The `marg` Specman interface functions are located at *<IXCOM_INSTALL_HOME>*/`share/vxe/etc/ixcom/marg_user.e`.

marg_t

`marg_t` is an ***e*** macro that defines a parameterized struct. You must define an ***e*** type by specifying the width of the `marg` object you are going to use. For example, to define a type for 64-bits width `marg` objects, use:

```
type marg64_t: marg_t of 64 bits;
```

You can then declare a `marg` object variable using that type, for example,

```
margW: marg64_t;
```

The following methods can be used on a `marg` object:

- To create a `marg` object of the specified depth and bind this `marg` object to a Verilog memory:

  ```
  new(depth: int, hdl_path_name: string);
  ```

- To delete a `marg` object:

  ```
  delete();
  ```

- To write data from `uint` (of 64-bits, in this example) to the `marg` object at the specified address:

  ```
  write_uint(addr: int, data: uint (bits: <width'exp>));
  ```

- To read data in `uint` (of 64-bits, in this example) from the `marg` object at the specified address:

  ```
  read_uint(addr: int, data:* uint (bits: <width'exp>));
  ```

- To write data in list of bit to the `marg` object at the specified address:

© 2021 All Rights Reserved.

```
write_lob(addr: int, data: list of bit);
```

For performance reasons, this method does not resize the data to align to the `marg` object width.

- To read and return data in list of bit (64 bits, in this example) from the `marg` object at the specified address:

```
read_lob(addr: int): list of bit;
```

- To transfer a block of data from the `marg` object to its associated Verilog memory:

```
put(addr: int, len: int);
```

- To transfer a block of data from the associated Verilog memory to the `marg` object:

```
get(addr: int, len: int);
```

marg_deposit_t

`marg_deposit_t` is an ***e*** macro that defines a parameterized struct. First, define an ***e*** type by specifying the width of the `marg_deposit` object you are going to use. For example, to define a type for 64-bits width `marg_deposit` objects, use:

```
type marg64_deposit_t: marg_deposit_t of 64 bits;
```

You can then declare a `marg_deposit` variable.

For example:

```
data_in: marg64_deposit_t;
```

The following methods can be used on a `marg_deposit_t` object:

- To bind the `marg_deposit` object to a Verilog variable:

```
bind(hdl_path_name: string);
```

Note: The width of the `marg_deposit` object must be greater than or equal to the width of the Verilog variable.

- To write data in `uint` (of 64-bits, in this example) to the associated Verilog variable:

```
write_uint(data: uint (bits: <width'exp>));
```

- To write data in list of bit to the associcated Verilog variable:

```
write_lob(data: list of bit);
```

Note: The number of bits in the data must be greater than or equal to the width of the Verilog variable.

© 2021 All Rights Reserved.

marg_value_t

`marg_value_t` is an ***e*** macro that defines a parameterized struct. First, define an ***e*** type by providing the width of the `marg_value` object you are going to use. For example, to define a type for 64-bits width `marg_value` objects, use:

```
type marg64_value_t: marg_value_t of 64 bits;
```

You can then declare a `marg_value` variable.

For example:

```
data_out: marg64_value_t;
```

The following methods can be used on a `marg_value` object:

- To bind the `marg_value` object to a Verilog variable:

  ```
  bind(hdl_path_name: string);
  ```

  Note: The width of the `marg_value` object must be greater than or equal to the width of the Verilog variable.

- To read data in `uint` (of 64-bits, in this example) from the associated Verilog variable:

  ```
  read_uint(data:* uint (bits: <width'exp>));
  ```

- To read and return data in list of bit (64-bits, in this example) from the associcated Verilog variable:

  ```
  read_lob(): list of bit;
  ```

Enabling and Disabling Features at Compilation Time

When compiling the design with the IXCOM compiler, you can specify the various features to be enabled or disabled for a particular instance hierarchy, design unit, or HDL file names by using the `-featureFile` <*`feature_file`*> option of the `ixcom` command.

Using this option, you can enable or disable:

- Instance hierarchy - for example, `top.FOO1`, and optionally, the entire hierarchy under it.
- Module hierarchy - for example, module `boo`, and optionally, all the modules instantiated by it.
- HDL file - for example, all the modules in the `file2.v` file.
- Support for assertions in uncontrolled-clock domains in specific design units or instances

© 2021 All Rights Reserved.

Important

The `-featureFile` <*`feature_file`*> option can be used only with the `ixcom` command. It does not work with the `vlan` or `vhan` command. This option is not the same as the `-covfile` option of the `xrun` command.

By default, SVA are enabled in the whole design. The feature file allows you to enable SVA for a specific instance hierarchy, such as `top.F001`. In such situations, add the following line in a <*`feature_spec_file`*>, and specify the name of this file with the `-featureFile` <*`feature_spec_file`*> option of the `ixcom` command:

```
sva disable
sva enable instance=top.FOO1+0
```

You can configure a feature to apply to a module, file, instance of a module, a part of the hierarchy of a design, and the entire design. The command-line options cannot provide that much granularity for configuration. If you use both feature-file configuration for a feature and the command-line option, the command-line option is ignored, and the feature-file configuration is enforced.

For example, if you use the following, the coverage counters for `assert` and `assume` directives will not be created and the `+assertCover` option of the `ixcom` command will be ignored:

```
$cat assn.cfg
   assert_cover disable
$ixcom -ua +dut+dut test.sv +assertCover -featureFile assn.cfg
```

A warning message is generated if a feature-file configuration is provided for a feature for which a command-line option is also provided.

Syntax of the <feature_file> File

In the <*`feature_file`*> file, you need to list the set of features that you want to enable or disable, along with the configuration options in the form of <*`feature_spec_lines`*>. Each line identifes the <*`feature`*>, the corresponding <*`control_option`*>, and the configuration to which this control is applied based on the <*`configuration_option_opt`*>.

Following is the syntax of the <*`feature_spec_line`*>:

```
<feature> <control_option> <configuration_option_opt>
```

See the following sections to know the list of valid values for each of the above-mentioned parameters:

- Specifying the <feature> Parameter

© 2021 All Rights Reserved.

- Specifying the <control_option> Parameter
- Specifying the <configuration_option_opt> Parameter

Depending on your requirements, you can specify the features to be enabled for the following types of configurations:

- Instance-Based Configuration
- Design-Unit-Based Configuration
- File-Based Configuration

Specifying the <feature> Parameter

Following are the valid values for the <*feature*> parameter:

<feature>	Denotes
`sva`	SystemVerilog Assertions feature.
`psl`	PSL Assertions feature.

`assert`	Both PSL and SystemVerilog Assertions features.
	Note that `assert` implies both the `sva` and `psl` features.
	So, for example, you need not use multiple lines as follows:
	`sva <control_option> <configuration_option_opt>`
	`psl <same_control_option_as_above> \ <same_configuration_option_opt_as_above>`
	Instead, you can specify a single line as follows:
	`assert <control_option> <configuration_option_opt>`

`coverage_functional`	SystemVerilog covergroups feature.
`coverage_block`	Block coverage feature.
`coverage_toggle`	Toggle coverage feature.
`coverage_expression`	Expression coverage feature.
`uncontrolled_clock_domain`	Support for assertions in uncontrolled-clock domains.

© 2021 All Rights Reserved.

The values for features related to assertions and covergroups are described in Controlling Assertions and Covergroup Through <feature_file> File.

Following is a sample <*feature_spec_line*> that specifies that the Block Coverage feature should be enabled in the `top.DUT` instance hierarchy:

```
coverage_block enable instance=top.DUT
```

Note: You can include empty lines (with only white space) and comments (lines beginning with # sign) in the <*feature_file*> file. Leading and trailing white space are allowed, and ignored.

To continue a <*feature_spec_line*> on the next line add a \ <*backslash*> character immediately before the <*newline*> character on the current line.

Alternatively, you can use specific `ixcom` command options to specify the features, such as assertions, coverage, and so on to be enabled or disabled in a particular module or analyzer session.

Specifying the <control_option> Parameter

Following are the valid values for the <*control_option*> parameter:

<*control_option*>	Denotes
`enable`	The feature should be enabled.
`disable`	The feature should be disabled.

The above values can be specified for all features. An invalid value for the <*control_option*> parameter results in compile error.

Specifying the <configuration_option_opt> Parameter

The <*configuration_option_opt*> parameter is optional. When it is not specified, the corresponding <*control_option*> is applied to the corresponding <*feature*> for the entire design (that is, globally).

© 2021 All Rights Reserved.

Following are the valid values for the <*configuration_option_opt*> parameter:

<configuration_option_opt>	**Denotes**
instance=<*instance_spec*>{,<*instance_spec*>} <*instance_spec*> ::= <*hierarchical_instance_path*>[+<*depth*>]	Instance-based configuration
du=<*design_unit_spec*>{,<*design_unit_spec*>} <*design_unit_spec*> ::= <*design_unit_name*>[+<*depth*>]	Design-unit-based configuration
file= <*filename*>{,<*filename*>}	File-based configuration
<*nothing*>	If no configuration option is specified, the <*control_option*> parameter for the specified feature is applied to the entire design.

An invalid value for the <*configuration_option_opt*> parameter results in compilation error.

Instance-Based Configuration

Instance-based configuration is defined as a cone of hierarchy starting at the instance specified as the <*hierarchical_instance_path*> having the depth as specified by the optional <*depth*> parameter.

Specify an integer value greater than or equal to zero as the <*depth*> parameter.

The interpretation for the value of the <*depth*> parameter is as shown below:

<depth>	**Indicates that the feature will be enabled for**
0	The instance at the beginning of the <*hierarchical_instance_path*> and the entire hierarchy under it.
1 (default)	The instance at the beginning of the <*hierarchical_instance_path*>.

© 2021 All Rights Reserved.

<depth>	**Indicates that the feature will be enabled for**
2	The instance at <*hierarchical_instance_path*> and one level of hierarchy under it.
<*N*> (a positive integer *N* greater than 2)	The instance at <*hierarchical_instance_path*> and (*N*-1) levels of hierarchy under it.
<*any_other_values*>	Compilation error

Note:

- If the specified <*hierarchical_instance_path*> cannot be resolved in the design hierarchy, or an invalid value is specified for the <*depth*> parameter, a compilation error is generated.
- Use of wildcards and environment variables for specifying the <*hierarchical_instance_path*> parameter is not supported.
- If the optional <*depth*> parameter is not specified, the <*depth*> is assumed to be 1.

Design-Unit-Based Configuration

The design-unit-based configuration is defined as a cone of hierarchy starting at each of the instances of the specified <*design_unit*> having the depth specified by the optional <*depth*> parameter.

The <*depth*> parameter takes an integer value that is greater than or equal to zero.

The interpretation for the value of the <*depth*> parameter is as shown below:

<depth>	**Indicates that the feature will be enabled for**
0	The instance at the beginning of the <*design_unit*> and the entire hierarchy under each instance.
1 (default)	All the instances of the <*design_unit*>.
2	All the instances of the <*design_unit*> and one level of hierarchy under each of the instances.
<*N*> (a positive integer *N* greater than 2)	All the instances of the <*design_unit*> and (*N*-1) levels of hierarchy under each of the instances.

© 2021 All Rights Reserved.

<depth>	**Indicates that the feature will be enabled for**
<any_other_values>	Compilation error

Note:

- If any of the instances of the specified *<design_unit>* cannot be found in the design hierarchy, or an invalid value is specified for the *<depth>* parameter, a compilation error is generated.
- Use of wildcards and environment variables for specifying the *<design_unit>* parameter is not supported.
- If the optional *<depth>* parameter is not specified, the *<depth>* is assumed to be `1`.

File-Based Configuration

The file-based configuration is defined as all of the design units defined in the specified file. The hierarchy under those instantiations is not included in this configuration. Unlike the instance and module based configurations, the *<depth>* parameter is not specified with file-based configuration.

Following is the syntax of file-based configuration:

```
<feature> enable|disable file=<filename>
```

The *<filename>* parameter must at least be a base filename, which means that it should at least contain a filename and extension(s), such as `test.v`, `flop.v`, `adder.sv`. In this case, IXCOM will look for the file only in the current compilation directory.

You can provide an absolute or relative (that is, relative to the current working directory) path while specifying the *<filename>*. There should be no ambiguity between files with the same name in different directories or libraries.

You can also use a symbolic link to a file or an environment variable while specifying the file-based configuration.

For example:

```
file=/home/test_path1/test.v
file=../test.v
file=$PWD/test.v
file=$DESIGNFILE1
```

Note:

© 2021 All Rights Reserved.

- Files included by using the ‘`include` directive in the <*filename*> file are not considered a part of the corresponding configuration.
- Use of wildcards is not supported in <*filename*>.
- If a module begins in one file and ends in another file, the enabling and disabling of such modules by using file-based configuration is defined by the file in which the module begins.
- Packages are not considered a part of the corresponding configuration. However, features in a package are disabled or enabled if any module refers to it. For example:

  Assume that a package named `pack1` is defined in the file named `test.v`. Then, the following configuration does not apply to `pack1`:

  ```
  sva disable file=test.v
  ```

  If the `pack1` package is defined in `test.v`, and referred to by the `dut` module in the `dut.v` file, the following configuration applies to the `dut` module, and to all the references to `pack1` package from the `dut` module. Note that the feature is still enabled for any calls to `pack1` from modules in `test.v` file.

  ```
  sva disable file=dut.v
  ```

- Compilation unit scopes are treated like packages as defined above.
- The file-based configuration is the equivalent of specifying design-unit-based configuration for each of the design units specified in the particular file.

  This means that:

  ```
  sva enable file=test.v
  ```

  Is equivalent to the following line for each of the design units defined in the `test.v` file:

  ```
  sva enable du=<design_unit_name>
  ```

Precedence Rules

You can specify multiple <*feature_file*> files by using this option multiple times, and each corresponding <*feature_file*> file for each option is processed in the order in which the option appears with the `ixcom` command. If the same <*feature_file*> file is specified multiple times, an informational message is issued that the previous specifications of this <*feature_file*> file are redundant and will have no effect.

Note: If required, you can improve your compile script by removing the previous specifications of this <*feature_file*> file.

The <*feature_files*> are processed or applied in the order in which they are specified on the command line and with a given <*feature_file*>, the <*feature_spec_lines*>

© 2021 All Rights Reserved.

are processed or applied in the order in which they appear. As a result of this, if two or more <*feature_spec_lines*> specify conflicting <*control_option*> for a given instance in the design hierarchy, the last specified <*feature_spec_line*> will be the one that takes precedence.

For example, consider the following command:

```
ixcom -assert top.sv block.sv chip1.sv chip2.sv -featureFile ff1 -featureFile ff2
```

In this example, the `ff1` file has the following <*feature_spec_lines*>:

```
sva enable
instance=top.block+0
sva disable file=chip1.sv
```

The `ff2` file has the following <*feature_spec_lines*>:

```
sva enable du=chip1_tester
```

In this example, the first line in the `ff1` file enables all the SVA in the `top.block` instance of the `block` design unit and all the instances within this hierarchy at all levels. The SVA in design units named `chip1` and `chip2` that are instantiated in `block` are also enabled.

However, the second line in `ff1` disables all the SVA that are present in modules within `chip1.sv`. Therefore, even though all the SVA in `chip1` should be enabled as per line 1, the SVA in `chip1` are disabled because of line 2.

Next, the `ff2` feature file is considered. It contains a line that enables SVA in the design unit called `chip1_tester`. This design unit might be in the `chip1.sv` file. So, now, even though the SVA in file `chip1.sv` have been disabled because of line 2 of the `ff1` file, the SVA in the `chip1_tester` design unit in the `chip1.sv` file are now enabled.

Refer to "Enabling and Disabling Assertions at Compile Time" on page 1213 for more details about the changes in the functionality of the options for enabling the SVA features when the `-featureFile` <*feature_file*> option is specified.

Note that the compound rules are expanded before precedence rules are applied. Refer to the Compound Features and Rules section for details.

Examples of <feature_spec_lines>

Consider the following example:

```
// file1.v                //-------- file2.v ------------      // file3.v
    module top;             module foo;       module boo;         module coo;
      foo FOO1();             boo BOO1();       cooCOO1();        endmodule
      foo FOO2();             coo COO1();     endmodule
    endmodule               endmodule
```


© 2021 All Rights Reserved.

Some sample <*feature_spec_lines*> and their interpretation for the above example are listed in the following sections:

- No Operation Example
- Example with No <configuration_option_opt> Parameter
- Examples with Instance-Based, Design-Unit-Based, and File-Based Configurations

No Operation Example

In the above example, the `sva` and `coverage_toggle` features are assumed to be enabled, by default.

If a feature is enabled by default, adding <*feature_spec_lines*> to enable the same feature in some design units, instances, files is regarded as a `No Operation` and the IXCOM compiler issues a corresponding informational message.

The following <*feature_spec_line*> specifies that the Toggle Coverage feature is enabled, but this is already enabled, by default, in the example. So, if this line is not preceded by a <*feature_spec_line*> to disable the coverage_toggle feature, this line is regarded as a `No Operation` and the IXCOM compiler generates a corresponding INFO message.

```
coverage_toggle
```

Similarly, the SystemVerilog Assertions feature was already enabled, by default, in the example. Without a preceding <*feature_spec_line*> to disable the `sva` feature, the following line results in a `No Operation` and the IXCOM compiler generates a corresponding INFO message.

```
sva enable
```

Example with No <configuration_option_opt> Parameter

The following <*feature_spec_line*> disables the Toggle Coverage feature in the entire design:

```
coverage_toggle disable
```

The following <*feature_spec_line*> disables the SystemVerilog assertions feature in the entire design:

```
sva disable
```


© 2021 All Rights Reserved.

Examples with Instance-Based, Design-Unit-Based, and File-Based Configurations

The following <*feature_spec_lines*> enable Toggle Coverage	**For the following instances**
Example: Instance-based Configuration	
coverage_toggle disable coverage_toggle enable instance=top.FOO1+0	top.FOO1 top.FOO1.BOO1 top.FOO1.BOO1.COO1 top.FOO1.COO1
coverage_toggle disable coverage_toggle enable instance=top.FOO1+0 coverage_toggle disable instance=top.FOO1.BOO1	top.FOO1 top.FOO1.BOO1.COO1 top.FOO1.COO1
Example: Design-Unit-Based Configuration	
coverage_toggle disable coverage_toggle enable du=boo+0	top.FOO1.BOO1 top.FOO1.BOO1.COO1 top.FOO2.BOO1 top.FOO2.BOO1.COO1
coverage_toggle disable coverage_toggle enable du=boo	top.FOO1.BOO1 top.FOO2.BOO1
coverage_toggle disable coverage_toggle enable du=coo	top.FOO1.COO1 top.FOO1.BOO1.COO1 top.FOO2.COO1 top.FOO2.BOO1.COO1
coverage_toggle disable coverage_toggle enable du=foo+0 coverage_toggle disable du=coo+0	top.FOO1 top.FOO1.BOO1 top.FOO2 top.FOO2.BOO1
coverage_toggle disable coverage_toggle disable du=coo+0 coverage_toggle enable du=foo+0	top.FOO1 top.FOO1.BOO1 top.FOO1.BOO1.COO1 top.FOO1.COO1 top.FOO2 top.FOO2.BOO1 top.FOO2.BOO1.COO1 top.FOO2.COO1
Example: File-Based Configuration	
coverage_toggle disable coverage_toggle enable file=file2.v	top.FOO1 top.FOO1.BOO1 top.FOO2 top.FOO2.BOO1

© 2021 All Rights Reserved.

The following <*feature_spec_lines*> control synthesis of assertions	**Description**
`assert enable instance=top.DUT`	Enables synthesis of all assertions in the `top.DUT` instance.
`assert enable du=dut`	Enables synthesis of all assertions in the `dut` design unit.
`assert enable label=AL1,AL2`	Enables synthesis of assertions with `AL1` and `AL2` label names within the design.
`assert disable label=AL1,AL2`	Disables synthesis of assertions with `AL1` and `AL2` label names within the design.
`assert enable label=AL1,AL2 instance=top.DUT`	Enables synthesis of assertions with `AL1` and `AL2` label names in the `top.DUT` instance.
`assert disable label=AL1,AL2 instance=top.DUT`	Disables synthesis of assertions with `AL1` and `AL2` label names in the `top.DUT` instance.
`assert enable label=AL1,AL2 du=dut_top`	Enables synthesis of assertions with `AL1` and `AL2` label names in the `dut_top` module.
`assert disable label=AL1,AL2 du=dut_top`	Disables synthesis of assertions with `AL1` and `AL2` label names in the `dut_top` module.
`assert disable instance=top.DUT`	Disables synthesis of all assertions in the `top.DUT` instance.
`assert disable du=dut`	Disables synthesis of all assertions in the `dut` design unit.

Controlling Assertions and Covergroup Through <feature_file> File

This section describes the Feature-File-related information relevant to functional coverage (assertions and covergroup) and assertion. Using <*feature_file*> file ensures a reasonable compilation time for configuring assertions and any other features compared to command line options.

© 2021 All Rights Reserved.

Typically, the <*feature_file*> works according to the following rules:

Rule	Descriptions
`assert_status enable`	Enable transformation for assertions control on the entire design.
`assert_status disable instance=top.dut_inst.b_inst2 +1`	Disable transformation for assertions control starting from the `top.dut_inst.b_inst2` instance up to `1` level of its sub-instances.
`assert_cover enable instance=top.dut_inst.b_inst3`	Enable creation of assertions coverage counter for assert and assume directives for the `top.dut_inst.b_inst3` instance.
`assert_counter_finish 20 du=dut_top`	Configure finish counter width for the `dut_top` module as 20 bits.
`assert_counter_check 24 file=test.v`	Configure check counter width as 24 bits for assertions inside modules defined in the `test.v` file.

The `assert_status`, `assert_cover`, and `assert_counter_finish` referred above are examples of feature names used for controlling transformations in a design using the feature file. Boolean and Integer are the two types of features. The boolean type of feature can have `enable` or `disable` as configuration value. The integral type of feature can have integers as configuration value.

The scope of a feature-file rule is specified using `du`, `instance`, and `file`. The scope specifies the part of design (or the whole design) on which the rule is applicable.

You can control the following features related to assertions and covergroups:

Feature Name	Config Type	Default Config Value	Description
`psl`	Boolean	enable	Feature to enable or disable PSL
`sva`	Boolean	enable	Feature to enable or disable SVA
`assert`	Boolean	enable	Feature to enable or disable SVA and PSL

© 2021 All Rights Reserved.

Feature Name	Config Type	Default Config Value	Description
`uncontrolled_clock_domain`	Boolean	disable	Feature to configure Assertions and Covergroups in dynamic target mode
`assert_status`	Boolean	disable	Feature to enable or disable run time assertions control
`assert_cover`	Boolean	disable	Feature to enable creation of coverage counter for assert and assume directives
`assert_counter`	Integer		Feature to configure counter width of Finish, Check and Fail counter of assertion
`assert_counter_finish`	Integer	1	Feature to configure width of Finish counter for assertion
`assert_counter_check`	Integer	1	Feature to configure width of Check counter for assertion
`assert_counter_fail`	Integer	32	Feature to configure width of Fail counter for assertion
`assert_counter_fail_hw`	Boolean	disable	Feature to enable or disable creation of Fail counter on HW. By default, the Fail counter is created in SW.
`coverage_functional`	Boolean	enable	Feature to enable or disable functional coverage of assertions and covergroup
`coverage_covergroup`	Boolean	enable	Feature to enable or disable functional coverage of covergroup
`coverage_assertion`	Boolean	enable	Feature to enable or disable functional coverage of assertion

© 2021 All Rights Reserved.

Feature Name	Config Type	Default Config Value	Description
`assert_message`	Boolean	enable	Feature to enable or disable reporting of assertions failure message when design is run on hardware. The `assert_message disable` rule stops incrementing failure counter when design is run on hardware. Specify `assert_counter_fail_hw enable` to continuously increment failure counter.
`assert_sw_only`	Boolean	enable	Feature to enable or disable synthesis of software-only assertions.
`functional_bc`	Boolean	enable	Feature to enable or disable synthesis of behavioral assertions and behavioral covergroup.
`assert_loop_ia`	Boolean	disable	Feature to enable or disable synthesis of immediate assertions inside loop statements.
`assert_deferred_ia`	Boolean	disable	Feature to enable or disable treating deferred immediate assertions as simple immediate assertions.
`assert_user_action_block`	Boolean	enable	Feature to enable or disable synthesis of user-written action blocks of assertions.
`assert_hw_action_block`	Boolean	disable	Feature to enable or disable execution of fail action block on hardware.
`assert_lps`	Boolean	enable	Feature to enable or disable low power (CPF/UPF) handling of assertion

© 2021 All Rights Reserved.

Feature Name	Config Type	Default Config Value	Description
`assert_past_ticks`	Integer	32	Feature to configure maximum number past ticks or sample for $past.
`assert_max_iter`	Integer	32	Feature to configure internal limit to select internal model for assertions synthesis
`gfifo_functional`	Boolean		Feature to enable or disable GFIFO-based execution of illegal bin hit for covergroup and assertions failure report
`gfifo_assertion`	Boolean	disable	Feature to enable or disable GFIFO-based execution of assertions failure report
`gfifo_covergroup`	Boolean	enable	Feature to enable or disable GFIFO-based execution of illegal bin hit for covergroup
`assert_optimize_const_property`	Boolean	enable	Feature to enable or disable identification and ignoring of static assertions during elaboration in the design

Compound Features and Rules

You can specify a rule with a compound feature or multiple rules with constituent sub-features.

The following table specifies the compound features and the corresponding sub-features.

Compound Feature	Sub-Feature
`assert`	`sva`
	`psl`
`assert_counter`	`assert_counter_finish`
	`assert_counter_check`
	`assert_counter_fail`

© 2021 All Rights Reserved.

Compound Feature	Sub-Feature
coverage_functional	coverage_covergroup
	coverage_assertion
uncontrolled_clock_domain	assert_message
	assert_counter_fail_hw
	functional_bc
gfifo_functional	gfifo_assertion
	gfifo_covergroup

The following table specifies the compound rules and the corresponding sub-rules.

Compound Feature	Sub-feature
assert enable	sva enable
	psl enable
assert disable	sva disable
	psl disable
assert_counter <*NUM_BITS*> <*NUM_BITS*> can be an integer with value ranging between 1 and 32.	assert_counter_finish <*NUM_BITS*>
	assert_counter_check <*NUM_BITS*>
	assert_counter_fail <*NUM_BITS*>
coverage_functional enable	coverage_covergroup enable
	coverage_assertion enable
coverage_functional disable	coverage_covergroup disable
	coverage_assertion disable
uncontrolled_clock_domain enable	assert_message disable
	assert_counter_fail_hw enable
	functional_bc disable

© 2021 All Rights Reserved.

Compound Feature	Sub-feature
uncontrolled_clock_domain disable	assert_message enable
	assert_counter_fail_hw disable
	functional_bc enable
gfifo_functional enable	gfifo_assertion enable
	gfifo_covergroup enable
gfifo_functional disable	gfifo_assertion disable
	gfifo_covergroup disable

Enabling Assertions in Uncontrolled Clock Domains

Specify the `uncontrolled_clock_domain` option in a feature file to enable support for assertions in uncontrolled-clock domains as follows:

- The following line enables assertions in uncontrolled clock domains in the design unit named `dut_top`.

  ```
  uncontrolled_clock_domain enable du=dut_top
  ```

- The following line enables assertions in uncontrolled clock domains in the design unit named `dut_top` and its next-level child units.

  ```
  uncontrolled_clock_domain enable du=dut_top+2
  ```

- The following line enables assertions in uncontrolled clock domains in the `tb_top.inst1.mem1` instance of the design unit named `memory`. Assertions in another instance, such as `tb.inst1.mem2` would not be considered as a part of the uncontrolled clock domain in that instance.

  ```
  uncontrolled_clock_domain enable instance=tb.inst1.mem1
  ```

- The following line enables assertions in uncontrolled clock domains in the `tb.inst1` instance and in the instances two levels below `tb.inst1`.

  ```
  uncontrolled_clock_domain disable instance=tb.inst1+2
  ```

Keep the following points in mind when specifying `uncontrolled_clock_domain` in feature file:

- No messages are generated for assertion failures for assertions inside uncontrolled clock domain.
- By default, the assertion-failure counters are implemented on the software side. But, the assertion-failure counters for uncontrolled-clock domains are implemented on the

© 2021 All Rights Reserved.

hardware side. The default width of failure counters is 1 for such assertions. For the assertions outside uncontrolled-clock domains, the assertion-failure counters are implemented on the software side, and the default width for these counters is 32.

Configuring Assertion Coverage Counters

This section describes the support for configurable assertion coverage counters. The following properties of assertion and coverage counters can be configured:

- Assertion control signal
- Assertion coverage counter for assertion and assume directives
- Width of counters of Finished, Failed, and Checked types
- Counter on hardware

The command-line options for assertions, such as the `+assertStatus` and `+assertCheckedCntr+<N>` options are applicable to all of the assertions or the entire design. The options listed in the following table, on the other hand, facilitate control for assertions in a single module. So, you can enable the assertions in one module and the assertions in the other module can be disabled.

Note: A command-line option and corresponding feature configuration in a feature file during the same compile are not allowed.

Feature	Description	Rule	Notes
`assert_status`	Assertion status control signal	`assert_status enable|disable du=<dut_top+0>`	Enables or disables creation of assertion status control signal assertion in the `dut_top` module and its sub-hierarchy.

© 2021 All Rights Reserved.

Feature	Description	Rule	Notes
assert_cover	Coverage counter for assert and assume directive	assert_cover enable|disable du=<*dut_top*+0>	Enables or disables creation of assertion coverage counter for assert and assume directive in the dut_top module and its sub-hierarchy.
assert_counter	Configure all counters that are mapped to hardware	assert_counter <*Nbits*> du=<*dut_top*+0>	Configures the minimum width of assertion coverage counter to *Nbits* for assertion in the dut_top module and its sub-hierarchy. If *Nbits* is 0, the assertion coverage counters are not created.
assert_counter_finish	Finish counter	assert_counter_finish <*Nbits*> instance=<*dut_top.dut_ip2.dut_ip3+2*>	Configures the minimum finished counter width to *Nbits* for assertion for the dut_top.dut_ip2.dut_ip3 instance and up to two levels of its sub-hierarchy. If *Nbits* is 0, the finished counter is not created.

© 2021 All Rights Reserved.

Feature	Description	Rule	Notes
`assert_counter_check`	Check counter	`assert_counter_check` <*`Nbits`*> `du=`<*`dut_ip5+0`*>	Configures the minimum checked counter width to *`Nbits`* for assertion for the `dut_ip5` module and its sub-hierarchy. If *`Nbits`* is 0, the checked counter is not created.
`assert_counter_fail`	Fail counter	`assert_counter_fail` <*`Nbits`*> `du=`<*`dut_top`*`+0`>	Configures the minimum failed counter width to *`Nbits`* for assertion for the `dut_top` module and its sub-hierarchy. If *`Nbits`* is 0, the failed counter is not created. **Note:** This rule or feature is significant only if the failed counter is mapped to hardware,

© 2021 All Rights Reserved.

Feature	Description	Rule	Notes
`assert_counter_fail_hw`	Fail counter on hardware	`assert_counter_fail_hw enable|disable du=<dut_top+0>`	Enables or disables creation of assertion coverage fail counter on hardware for `assert` and `assume` directive in the `dut_top` module and its sub-hierarchy. By default, the fail counter is created in software.

Execution of Software Tasks for Assertions and Covergroups Using GFIFO-Based Mechanism

This section describes the GFIFO-based mechanism for execution of software tasks in the assertion (software task for execution of fail action block) and covergroup (software task for execution of illegal bin hit reporting) context. If this feature is not enabled, these software tasks are executed using the TBCALL-based mechanism. The failed action block execution using the default TBCALL mechanism requires to halt hardware execution and forces a testbench synchronization event to execute the action block task on the simulator. This `tbsync` needed to report an assertion failure can affect run-time-performance significantly if many such action blocks trigger during simulation.

The GFIFO-based mechanism to execute the action block for assertion failure reporting uses the non-blocking and asynchronous IXCOM GFIFO technology that does not require `tbsync` and can address this performance concern.

The buffered GFIFO mechanism does not require a virtual logic for implementing a global stop condition of the hardware for immediate (in the same simulation time) reporting of assertion failure message it can scale better without affecting step-count if the number of assertions in the design is large.

Execution of action blocks using GFIFO mechanism is non-blocking, buffered, and asynchronous. So, it cannot support any generic failed action block tasks. Hence, the tool internally specifies the `assert_user_action_block` disable rule when GFIFO mechanism is enabled and implements a generic tool-generated assertion failure message.

© 2021 All Rights Reserved.

The assertion-failure-message reporting might be delayed due to buffering. However, the simulation time reported for assertion failure is accurate. In addition, the SDL trigger for assertions is not supported when GFIFO interface is used for execution fail action block.

Note: The `assertion -onFailure` command might not function properly when GFIFO is enabled for execution of fail action of assertions.

The feature for GFIFO-based execution of software task for assertions and covergroups is configured using the `gfifo_functional` compound feature in the Feature file by enabling or disabling the following rules:

`gfifo_assertion`: The `gfifo_assertion` rule is used to configure assertion fail action block execution using GFIFO based software Task. This remains disabled by default.The `gfifo_assertion enable` rule enables execution of fail action block using GFIFO based software task execution.

`gfifo_covergroup`: The `gfifo_covergroup` rule is used to configure reporting illegal bin hit using GFIFO base software task. The feature remains enabled by default. The `gfifo_covergroup enable` rule enables execution of illegal bin hit reporting using GFIFO based software task execution.

© 2021 All Rights Reserved.

Using Conformal to Verify IXCOM-Generated Output

Encounter Conformal (commonly called **LEC**) is the Cadence Equivalence Checker tool that is used to compare and validate two models of the same design.

For comparing the models at the HDL-ICE Compiler stage, refer to the Using Conformal to Verify HDL-ICE Compiler-Generated Netlist section.

IXCOM provides the functionality of comparing various models as follows:

- Comparison of IXCOM-generated HDL with HDL-Compiler-synthesized netlist (specified with the solid purple lines —— in the diagram below). This is called the **modNet LEC** mode.
- Comparison of IXCOM-elaborated model with the original RTL that was provided as input to IXCOM (specified with the dashed purple lines – – – in the diagram below). This is called the **OrigElab LEC** mode.

© 2021 All Rights Reserved.

The following diagram illustrates the use of LEC tool for comparisons of the models at IXCOM compilation stage:

Generating the Files for Comparison with the LEC Tool

IXCOM provides the comparison functionality when you use the `-conformal` option with the `ixcom` command.

© 2021
All Rights Reserved.

This option generates the following files:

Filename / location	Description
`./xc_work/lec/lec.mak`	The Makefile to run equivalence check.
`./xc_work/lec/v/*`	HDL model having just the elaborated view
`./xc_work/lec/[other_support_files]`	Support files used by primary script.
`./xc_work/lec/lec.scr`	The primary script for running the LEC tool in modNet mode that creates the log files, and other secondary script files after the LEC tool is run. (For comparing IXCOM-generated RTL and HDL-Compiler-generated netlist)
`./xc_work/lec/lec_origElab.scr`	The primary script for running the LEC tool in origElab mode that creates the log files, and other secondary script files after the LEC tool is run. (For comparing original RTL and IXCOM-elaborated RTL model)

These scripts include the LEC commands customized for your design, and are used to invoke LEC after HDL-ICE Compiler has finished synthesizing the design. These scripts closely match the script that is generated by HDL-ICE Compiler to compare the models at the HDL-ICE Compiler stage.

The IXCOM compiler automatically includes most of the necessary LEC-specific options in the LEC scripts that it generates. If required, you can also customize these generated scripts according to your requirements. By default, the root module for LEC comparisons is the `xcva_top` module. If you want to specify an alternative module instead of the default `xcva_top` module, use the `-conformalTop` <*`module_name`*> option.

Note: To use Conformal, you must purchase its licenses from Cadence. Refer to the *Conformal Equivalence Checking User Guide* for installation and platform support of Conformal software.

The `lec.mak` script generated by the IXCOM compiler invokes the Conformal Ultra tool to ensure better performance. The LEC tool is invoked as follows:

```
lec -xl
```

Important

This requires a Conformal Ultra license.

© 2021 All Rights Reserved.

Comparing the Models Using the xeLecRun Command

To compare the models, use the `xeLecRun` command.

```
xeLecRun [-modNet | -OrigElab] | -help | -clean
```

- The `-modNet` argument enables comparison of the IXCOM-modified RTL model and synthesized netlist model. The default comparison mode is `-modNet`.
- The `-OrigElab` argument enables comparison of the original RTL model and IXCOM-elaborated model.
- The `-clean` option removes all the output files from the last run.
- The `-help` option prints the help information for the `xeLecRun` command.

Keep the following points in mind while using the `xeLecRun` command:

- If any invalid options are used, or multiple comparison modes are specified with one `xeLecRun` command, the tool exits after generating an error message.
- If the `-clean` option is specified along with a valid comparison mode (either `ModNet` or `Origelab`), first the logs of previous run will be cleared; then the specified comparison mode is executed.

Comparing Models in the OrigElab Mode

The following option runs the LEC tool for the **OrigElab** mode, which can be executed using either of the following commands:

```
xeLecRun -origElab
```

Or

```
make run_lec_origElab -f xc_work/lec/lec.mak
```

Or

```
lec -64 -XL -NOGUI -DOFILE xc_work/lec/lec_origElab.scr
```

The output of the LEC run is stored in the `lec_script.log` file.

Comparing Models Using the LEC Tool in the modNet Mode

Run the LEC tool for the **modNet** mode using either of the following commands:

```
xeLecRun -modNet
```

Or

© 2021 All Rights Reserved.

```
make run_lec -f xc_work/lec/lec.mak
```

Or

```
lec -64 -XL -NOGUI -DOFILE xc_work/lec/lec.scr
```

The output of the LEC run is stored in the `lec_script.log` file.

Files Generated by the LEC Tool

After the LEC tool has been invoked by the IXCOM-generated makefile as mentioned above, the LEC tool generates the following files in the same directory as the IXCOM compilation directory:

- Log file: `lec_script.log` (for comparing IXCOM-generated RTL and HDL-Compiler-generated netlist)
- Intermediate `do` file (removed after the LEC tool finishes execution)
- File of black box definitions

These files are not removed after Conformal finishes. If required, you can use these for analysis and debugging.

Customizing the LEC Tool Invocation

For better performance, and as per your requirements, you can customize the scripts generated by the IXCOM compiler for invoking and running the LEC tool in the following ways.

Excluding Modules from Equivalence Comparison

To avoid long verification times, you can choose to exclude certain modules from equivalence comparison by specifying the module names in a text file named `lec_script.moduleSkipEC`. Each module name should be on a separate line. The Conformal script created by IXCOM will read this file, and apply it to the modules to be excluded. Place this script in the design directory.

Customizing LEC Script File Using a Configuration File

You can customize the default LEC script that is generated by the IXCOM compiler by including the required commands in a configuration file. The configuration file should be a text file named `lec_script.config`, and it should be placed in the design directory from where the LEC tool is being invoked.

© 2021 All Rights Reserved.

In this file, you can include a set of LEC commands, with each command appearing on a separate line. The original LEC script generated by IXCOM sources this configuration file.

For example, you can use this configuration file to enable the multi-threading feature in the LEC tool. The multi-threaded feature is not set, by default, in the LEC script.

```
set parallel option -threads 4
```

The above command in the configuration file sets the maximum number of threads to be used to `4`.

Alternatively, in the configuration file, you can specify additional attributes to certain modules as follows:

```
add module attribute  <module_name> -compare_effort low
```

Or

```
add module attribute  <module_name> -cpu_limit 3600
```

Such attributes help in limiting the time spent in the comparison of these modules.

Caution

You can alter the configuration of the LEC script file generated by IXCOM. But it is advisable that you make these alterations only if you are an expert in the LEC commands to ensure that you do not inadvertently corrupt the script.

One way to configure the IXCOM-generated LEC script is by using the following predefined files that you can create and add the required LEC commands in these files:

- `lec_pre.config`

  The LEC commands in this file are executed before LEC reads and compiles the golden and revised models.

- `lec_script.config`

  The LEC commands in this file are executed after the elaboration of the golden and revised models, but before the hierarchical comparison is begun.

- `lec_before_exit.config`

  The LEC commands in this file are executed just before LEC executes the `exit` command.

© 2021 All Rights Reserved.

Limitations with the Comparisons of Models at IXCOM Compilation Stage

A meaningful result might not be generated in the following cases:

- All the cases from LRM that are not supported by Conformal LEC.
- All the tool-specific cases that are not a part of the LRM, such as `XM_MIRROR`, parameter override from command, and so on might or might not be supported depending on the corresponding support by Conformal LEC.
- A case when the DUT part has any clocking blocks or behavioral RTL. Modules having such RTL are not black-boxed, but will not be compared. This is because the hierarchies below these modules are considered for equivalence check.
- Design is compiled in multiple sessions. But in the final elaborated model, some modules are not part of it.
- The `` `define `` statement in testbench is used in DUT; both are in separate files. The testbench file is not passed to LEC and it fails to parse the DUT file for the `` `define `` statement.
- Certain cases of VHDL like the following:
  - In VHDL 93, `std_logic_vector` and `std_ulogic_vector` are different types. `std_logic_vector` is not a subtype of `std_ulogic_vector`; it is defined as a new type:

    ```
    TYPE std_logic_vector IS ARRAY (NATURAL RANGE <>) OF std_logic;
    ```

  - In VHDL-2008 standard, definition of `std_logic_vector` is changed to be a subtype of `std_ulogic_vector`:

    ```
    SUBTYPE STD_LOGIC_VECTOR is (resolved) STD_ULOGIC_VECTOR;
    ```

  - In VHDL 2008, these are not different. Therefore, the following kinds of functions are not considered valid:

    ```
    function cva_countones(in1 : std_ulogic_vector) return integer;
    function cva_countones(in1 : std_logic_vector) return integer;
    ```

- IXCOM includes some of its packages (like `axis.vhd`) that has the above kind of repeated definition. This also causes errors during LEC comparison.
- VHDL packages used in SystemVerilog. This is a non-standard tool specific functionality where a VHDL package can be used by SystemVerilog design directly like SystemVerilog package. VHDL packages cannot be used in SystemVerilog design for LEC comparison. This might generate errors in LEC comparisons, or the affected module might be blackboxed.
- SystemVerilog configuration is not supported if it is used to bind the module instance within generate block. Bound modules are not compared. Consider the following

© 2021 All Rights Reserved.

example that illustrates the case for which LEC does not compare the `ip1` module that has the `ip1_inst` instance in the generate block and is bound through a Verilog configuration.

```
// Test.sv
module top;
genvar i;
    for (i = 0; i < 1; i = i + 1) begin:gen
      ip #(i)ip1_inst();
    end
endmodule
module ip1;
  parameter p1 = 20;
  initial $display("%m: ip1");
endmodule
```

```
// Config.sv
config cfg;
  design top;
    instance top.gen[0].ip1_inst use ip1;
endconfig
```

- Cases that use the `+dutexcl` or the `+dutIgnore` option of the `ixcom` command.
- Cases in which parameters are overridden on the `ixcom` command line. LEC compares the design without considering the effect of parameter overrides using the `-defparam`, `-generic`, or `-gpg` option.
- Modules with Verilog behavioral code are ignored during LEC comparisons.
- Modules that have tasks defined in testbench are not compared.
- The OrigElab mode is not available in cases in which both the DUT and testbench modules are in the same original RTL file. In this case, the following warning message will be displayed.

```
Warning!                  [LEC_RTL_NOCMPR_TBDUT]
        At least one file (../dut/hw_top.v) has both TB and DUT module; So lec
script for user-rtl compare flow wouldn't be generated.
```

Classifying Memories as Black-boxes

In certain cases, the HDL-ICE Compiler infers memories from the RTL arrays and uses VXE memory primitives (MPR and MPW), which are not understood by the LEC tool. So, either the LEC tool takes a long time for large memories to finish, or reports that the modules are non-equivalent in many instances.

The IXCOM compiler attempts to infer the target map for the memories in the design, and back-annotates the memory information after reading the HDL-compiler-generated files. Correspondingly, it excludes all those modules that infer VXE memories by writing the required black-boxing commands in the script that it generates for the LEC comparison.

© 2021 All Rights Reserved.

Using Conformal for External Netlists

External netlists are passed to xeCompile without any modifications when the `-externalNetlist` option is used with the `ixcom` command. In the external netlist flow, as described in Importing External Netlists in the SA Mode section, LEC comparison is as follows.

The part of the design that is imported and the design hierarchy below this part are not fed to LEC, and, hence, these are not compared.

This includes the netlist modules that are compiled using IXCOM as follows:

```
vlan -netlist
```

The part of the design that is compiled and synthesized using IXCOM and is exported is compared.

To enable LEC flow with external netlists, use the following command:

```
ixcom -conformal -externalNetlists [other_ixcom_options]
```

HDL-Compiler Options Necessary for Invoking the LEC Tool from the IXCOM Compiler

When you use the IXCOM compiler to invoke the LEC tool, at the HDL compilation stage, it internally switches on the following options for the `hdlSynthesize` command and vaelab utility:

- `-keepAllFlipFlop`: This option ensures that certain registers are not optimized during synthesis. If this option is not enabled, the LEC tool might not be able to map all the key points, and it might, subsequently, report false non-equivalence. Note that this option, however, slightly increases the gate count of the HDL output netlist.
- `-keepRtlSymbol`: This option ensures that Conformal-compatible names are generated when the output Verilog netlist is generated. This option is needed to improve the Conformal-mapping process and accuracy of comparison.

By default, both of these options are included in the makefile generated by the IXCOM compiler. However, if you have disabled these options in a compilation script, you would need to create a separate target for the LEC tool with these options enabled.

© 2021 All Rights Reserved.

Generating Light Weight Debug Database (LWD) from IXCOM

This section describes the capability of generating the Indago-compatible debug database and Xcelium scripts from IXCOM. This capability is provided through the use of the `-enableLWD` option of the `ixcom` command. This option generates the required files to generate the Indago-compatible debug database for your original RTL files.

The `-enableLWD` option dumps the `lwd_work` directory that includes the scripts required for Xcelium-snapshot generation for original RTL. It also generates the `-createdebugdb` Xcelium-elaboration option.

Using the `-enableLWD` option for SystemVerilog and VHDL designs:

- Translates the `vlan` and `vhan` options to equivalent `xrun -compile` command.
- Translates the library mapping to equivalent `cds.lib` mapping.
- Translates the relevant IXCOM-elaboration related commands to `xrun -elaborate` commands and adds the `-createdebugdb` option to generate the Indago database.
- Generates the `xrun` single-step compilation command, if the `ixcom` single-step compilation command is used.
- Creates the Xcelium elaboration snapshot during IXCOM compilation process.

Following are the files generated by the `-enableLWD` option:

- `cds.lib` and `hdl.var`
- `elab.f` (Option file containing the Xcelium elaboration options)
- `lwd_make` (Makefile)

For designs compiled with single-step (single session) `ixcom` command, the Makefile (`lwd_work/lwd_make`) includes the `xrun` (Xcelium single step) and for designs compiled in multi-step IXCOM compilation mode, that is with the `vlan`, `vhan`, and `ixcom -elaborate` commands, the Makefile includes the multi-step Xcelium compilation commands, that is the `xrun -compile` and `xrun -elaborate` commands.

VHDL or mixed-language designs are compiled in IXCOM multi-step mode. Correspondingly, for such designs, the Xcelium multi-step commands, that is `xrun -compile` and `xrun -elaborate` are generated.

The following command generates the Xcelium snapshot with Indago debug database:

```
make -f lwd_work/lwd_make
```


© 2021 All Rights Reserved.

This command is invoked automatically within the same IXCOM compilation process and the Xcelium-elaboration snapshot is generated. This `make` command is useful you want to modify (or correct) the script and run only the Xcelium-snapshot-generation step.

Only those options that have similar options in Xcelium compilation are mapped in the Makefile; All options of the `ixcom` command that do not have any Xcelium-side options are ignored.

Handling of Multi-Step IXCOM Compilation Commands

Consider the following multi-step IXCOM compilation commands:

```
vlan -32 -sv2005 mid.sv
vlan -32bit -sv2009 +incdir+dir1+dir2 dut.sv top.sv
vhan -32 -200x top.vhd -verbose
ixcom -32bit -io_port_relax -ua -top test +dut+test -verbose \
      -rtlNameForGenerate -enableLWD -generic "gen => 20"
```

After IXCOM compilation, a directory named `lwd_work` is generated in the `xc_work` directory with the following files:

```
cds.lib
hdl.var
elab.f
lwd_make
```

Additionally, the following work directories are created for IESXcelium compilation:

```
<user_lib_directories> worklib
```

Here, `Makefile lwd_make`includes all the multi-step Xcelium compilation commands required to generate the Xcelium snapshot with the `-createdebugdb` option.

Content of the `lwd_work/lwd_make` Makefile for Multi-Step Xcelium Compilation

```
XMVLOG_OPTIONS = -nowarn DLNCML -nowarn TMTPAR \
                 -xmlibdirpath ./xc_work/lwd_work
XMVHDL_OPTIONS = -nowarn DLNCML -xmlibdirpath ./xc_work/lwd_work
XMELAB_OPTIONS = -nowarn DLNCML -xmerror CUNOTB \
                 -xmlibdirpath ./xc_work/lwd_work -createdebugdb
XMSIM_OPTIONS =
HDLVAR = -append_log -logfile xc_work/lwd_work/xrun.log \
         -cdslib xc_work/lwd_work/cds.lib \
         -hdlvar xc_work/lwd_work/hdl.var
xmelab : xmvlog
                @/bin/rm -rf xc_work/lwd_work/xcelium.d/run.lnx*.d
                xrun -elaborate $(XMELAB_OPTIONS) \
                     -f xc_work/lwd_work/elab.f -message
xmvlog :
         xrun -compile $(HDLVAR) $(XMVLOG_OPTIONS) -sysv2005 mid.sv
         xrun -compile $(HDLVAR) $(XMVLOG_OPTIONS) -sysv2009 \
              +incdir+dir1+dir2 dut.sv top.sv
         xrun -compile $(HDLVAR) $(XMVHDL_OPTIONS) -mess -v200x top.vhd
```


© 2021 All Rights Reserved.

For multi-step Xcelium compilation, `DEFINE LIB_MAP (+=> WORK)` is added in `lwd_work/hdl.var` file so that the `-y` and `-v` flags do not create a sub-library for each module in the `xrun -compile` step.

SystemVerilog configuration is supported only in multi-step compilation. For SystemVerilog configuration, the `-compcnfg` option needs to be passed to the `xmvlog` binary. Therefore, during LWD generation, the `-compcnfg` option is added to `XMVLOG_OPTIONS` variable in the `hdl.var` file to compile the configuration during LWD generation. Also, IXCOM consumes the configuration during the `vhelab` step. So, the top-level module generated after `vhelab` is passed to `xrun` instead of the top-level module identified by IXCOM.

Handling of Single-Step IXCOM Compilation Command

For designs run with single-step (single session) `ixcom` command, the Makefile has the `xrun` command (Xcelium-single-step).

For example, consider the following `ixcom` command:

```
ixcom +sv -ua +dut+top ./fifo.sv clkgen.sv +gfifoDisp +define+INST \
      +define+DISPLAY -defparam top.B=8 -defparam top.f_inst.W=4 -enableLWD
```

After IXCOM compilation, a directory named `lwd_work` is generated in the `xc_work` directory with the following files:

```
cds.lib
hdl.var
elab.f
lwd_make
```

Additionally, the `worklib` work directory is created for Xcelium compilation.

Here, `Makefile lwd_make` includes the single-step Xcelium compilation command required to generate the Xcelium snapshot with the `-createdebugdb` option.

Content of the lwd_work/lwd_make File for Single-Step Xcelium Compilation

```
XMVLOG_OPTIONS          = -nowarn DLNCML -nowarn TMTPAR
XMVHDL_OPTIONS          = -nowarn DLNCML
XMELAB_OPTIONS          = -nowarn DLNCML -xmerror CUNOTB \
                          -xmlibdirpath ./xc_work/lwd_work -createdebugdb
XMSIM_OPTIONS           =
HDLVAR                  = -64bit -append_log \
                          -logfile xc_work/lwd_work/xrun.log \
                          -cdslib xc_work/lwd_work/cds.lib \
                          -hdlvar xc_work/lwd_work/hdl.var
xrun : @/bin/rm -rf xc_work/lwd_work/xcelium.d/run.lnx*.d
       xrun -elaborate $(XMELAB_OPTIONS) -f xc_work/lwd_work/elab.f \
            -message $(HDLVAR) $(XMVLOG_OPTIONS) -sv +define+INST \
            +define+DISPLAY ./fifo.sv
```


© 2021 All Rights Reserved.

Handling of +xcDesignTop and -xmcompile Options with -enableLWD

When you include the `+xcDesignTop` and `-xmcompile` options with the `-enableLWD` option of the `ixcom` command, only the `xmvlog` and `xmvhdl` targets are generated. The `xmelab` target is not generated and you need to compile the testbench files separately, and elaborate the combined (IXCOM-compiled and Xcelium-compiled) snapshot.

When you include the `-xmcompile` or `+xcDesignTop` options, the following options are included in the generated file `xc_work/lwd_work/elab.f`.

```
-xmlibdirpath ./xc_work/lwd_work
-createdebugdb
```

For compiling testbench files you need to specify the `-f xc_work/lwd_work/elab.f` file with the <*tb_source_files*> testbench files as follows:

```
xrun -elaborate -f xc_work/lwd_work/elab.f \
     -top <top_module_name> <tb_source_files> [other_options]
```

Generating the Indago-Compatible Debug Database with LWD Feature

Using the `-createdebugdb` option during Xcelium elaboration generates the debug database to support post-simulation debug for the Indago tool.

Then, you can invoke the Indago tool by using the following command:

```
indago -lwd xc_work/lwd_work/worklib -db trace.shm &
```

Restrictions with the LWD Feature

The LWD feature is not supported with the following:

- Netlist flow
- Feature files
- SystemVerilog configuration with single-step compilation
- VHDL packages imported in SystemVerilog source files
- Parallel `vlan` flow
- `+moduleStub` Option
- `-ixclkgen` Option
- `-generic` Option
- `+rtlCommentPragma` Option

© 2021 All Rights Reserved.

- `-Z`, `-z`, or `-m` Options (These might result in Xcelium compilation failure.)
- Any algorithmic differences in resolution of the `-v`, `-y`, or similar options result in different resolutions in the generated snapshot.
- IXCOM-specific constructs and defines, such as `axis_ primitives`, `ixc_` primitives, `$export` directives, `$ixc_ctrl` directives might result in Xcelium compile failure.

Recommendations for the LWD Feature

Examine the generated scripts files for correctness, and keep the following points in mind:

- Examine the generated LWD script for any missing files. Basically, any testbench files that were not supplied to IXCOM but are still a part of the testbench from your perspective should be added into the script.
- It is possible that `xrun` and IXCOM handle the RTL code differently. These differences can manifest themselves when you debug the design with the Indago tool.
- To pass the options specific to Verilog, VHDL, or elaboration directly without needing to modify the LWD scripts after generation, use the `-lwdvlogargs`, `-lwdvhdlargs`, and `-lwdelabargs` options. In single-step compilation, specify these options with the `ixcom` command. In multi-step compilation, you need to specify the `-lwdvlogargs` option with the `vlan` command and specify the `-lwdvhdlargs` option with the `vhan` command.These options append the specified option string to the LWD scripts, minimizing the need for script modification after generation.
- Generation of the LWD database is not fully automated and IXCOM generates scripts to help the generation of the database. So, errors are possible. In such cases, review the scripts in the `lwd_work` directory and the specified source files. Then, regenerate the LWD database by using the following staged-compilation command:

  ```
  ixcom -target lwdmodel
  ```

  The above command invokes just the LWD database and is equivalent to the following command in which <*LWD_database_directory_path*> is `xc_work/lwd_work`.

  ```
  make -f <LWD_database_directory_path>/lwd_make
  ```

  The following command generates the LWD scripts and compiles the software portion of the design by invoking Xcelium.

  ```
  ixcom tb.sv dut.sv ... -enableLWD -target swmodel
  ```

  Then, you can generate the LWD database and compile the hardware portion of the design by using the following commands:

  ```
  ixcom -target lwdmodel
  ixcom -target hdlice
  ```


© 2021 All Rights Reserved.

The `lwdmodel`option enables you to generate the lightweight debug database for the design for debugging with the Indago tool. The `hdlice` option generates the synthesized netlist for the design for acceleration on the Palladium Z1 emulator.

- For designs analyzed with only SystemVerilog configuration file with `vlan` command, that is without any module files, the corresponding `xmvlog` command is not generated and for these cases, either you should map and add only the SystemVerilog configuration session command in the `xc_work/lwd_work/lwd_make` makefile or specify the SystemVerilog configuration file along with an RTL module file used in the design.
- If you are using a SystemVerilog configuration with the `+xcDesignTop` option, then you should review, and, if required, update the configuration file for LWD generation and the generated top-level module name with the `-top` option in the `xc_work/lwd_work/elab.f` file because the top-level module will be different for LWD from the IXCOM compilation with the `+xcDesignTop` option.

Support for External Netlists with LWD

In the netlist flow, IXCOM passes the netlist to Xcelium if hot swap is enabled, and when hot swap is disabled, IXCOM passes the skeleton to Xcelium. By default, the hot-swap capability is disabled for the netlist hierarchy. To enable it, include `NL_HOTSWAP on` in the configuration file.

For consistency with this for the LWD build (enabled with the `-enbaleLWD` option), pass the netlist files to Xcelium only if hot-swap is enabled. If hot swap is disabled, a skeleton is passed and you will not be able to debug the netlist content. Note that this enables full visibility of the netlist at the cost of performance.

Note: If hotswap is disabled and the skeleton is passed to Xcelium for LWD build, errors might be generated at the `xmelab` stage if there are XMRs from RTL to the netlist hierarchy or bind into netlist hierarchy.

Enabling Generation of xrun Command with Name of File Containing List of Files

Using the `-enableLWD` option of the `ixcom` command creates a makefile `xc_work/lwd_work/lwd_make`, which specifies the paths of all of the files required during compilation. If the number of files is very large, this path list can be huge and result in the `Argument list too long` error. To avoid this error, the `xrun` command generated in the LWD scripts has the `-f` <*`filelist`*> command instead of listing all files explicitly. Here, <*`filelist`*> is the name of the file that you have specified with the `ixcom` command.

© 2021 All Rights Reserved.

Generating Lightweight Simulation Snapshot for Large Designs

With the design sizes continuing to increase to billion gates, either the Xcelium compilation time for parsing or elaborating the designs does not scale in terms of compilation time and required memory, or the Xcelium snapshot size becomes too large. The default IXCOM compilation flow supports hot-swap and requires a fully congruent simulation model for the whole design to be compiled into a an Xcelium simulation snapshot. The Lightweight flow addresses these limitations and provides better support for the very large designs (a billion gates or more) by reducing the Xcelium compilation time and snapshot size with the trade-off of restricting the hot-swap capability.

This section describes the Lightweight flow in which IXCOM prunes out the RTL-DUT partition in the Xcelium-compiled snapshot to reduce the overall dependency on Xcelium during IXCOM compilation to resolve the capacity limitations of Xcelium. This flow is referred to as **IXCOM Lightweight Simulation Snapshot flow** or simply the **Lightweight flow**. This flow supports TBA, behavioral compilation, coverage, and assertion features.

Introduction to the Lightweight Flow

Typically, the core DUV (the logic which is to be implemented on silicon) part of DUT (top-level hardware design unit of IXCOM specified with the `+dut+<du>` option) is pure RTL. The peripheral TBA models (the glue logic around DUV) that serve as the interface for the Test-Generator (or testbench) (like the Driver-BFM, monitors or receivers, scoreboard and other transactors) and facilitate the verification of the DUV have behavioral code that might require initialization and synchronization.

In the Lightweight flow, IXCOM generates a lightweight simulation snapshot by pruning away the RTL-DUT code for Xcelium partition and preserving only the TBA-essential DUT modules. The lightweight snapshot comprises only those testbench modules or TBA-essential DUT modules that are necessary to initialize the design at time zero and subsequently support TBA calls into the testbench from the DUT in hardware, including `tb_import` (hardware -> software) and `tb_export` (software -> hardware.) This flow helps maximize TBA performance with IXCOM; However, it restricts swap-out back into simulator after swap-in at time zero. This flow supports the IXCOM SA environment and the IXCOM TBA (DPI calls and GSFIFO), behavioral compilation, coverage, and assertion capabilities.

The Lightweight flow allows you to explicitly direct IXCOM that certain DUT models are pure RTL and do not have any dependency on Xcelium for the functional execution of logic at run time. IXCOM dumps a SHELL model for such hardware-only-RTL models by *emptying out* the content of these models from Xcelium compile and dumping only a skeleton with ports and the model instantiations with an unconnected port list to preserve the design hierarchy.

© 2021 All Rights Reserved.

If there are any conflicting markings, the IXCOM compiler issues warning and error messages.

```
module dut_hw_rtl ( clk, in1, in2, out);
    input clk;
    input [15:0] in1;
    input [15:0] in2;
    output [15:0] out;
    mod1 I1 ( ,  ,  ,);
    mod2 I2 ( ,  ,  ,);
endmodule
```

In the Lightweight flow, use the following options to specify the hardware-only-RTL DUT models:

Command	Option	Description
`vlan`	`-hwOnlyRtl`	Analyzes the specified HDL sources and marks the modules in the session as candidates for hardware-only-RTL optimization.
`vhan`	`-hwOnlyRtl`	Analyzes the specified HDL sources and marks the modules in the session as candidates for hardware-only-RTL optimization.
`ixcom`	`+hwOnlyRtl+<`*modA*`>+<`*modB*`>`	Specifies the names of the modules that should be marked as candidates for hardware-only-RTL modules.
`ixcom`	`+hwOnlyRtlExcl+<`*modA*`>+<`*modB*`>`	Resets the hardware-only-RTL marking done either by using `vlan -hwRtlOnly` or `ixcom +hwRtlOnly+<`*modA*`>+<`*modB*`>` commands.
`ixcom`	`+hwOnlyRtlTop+<`*mod1*`>+<`*mod2*`>`	Marks the specified module and all modules of the instances under the specified module hierarchy as candidates for hardware-only-RTL modules.

The Lightweight flow is supported in the single-step compile flow with the `+hwOnlyRtl+<`*mod1*`>+<`*mod2*`>` and `+hwOnlyRtlTop+<`*mod1*`>+<`*mod2*`>` options of the `ixcom` command to enable specification of the hardware-only-RTL modules.

© 2021 All Rights Reserved.

Important

The `+hwOnlyRtl` option augments the `-hwOnlyRtl` option and can be used to specify RTL models if there are any in common source files.

IXCOM Hardware-Only-RTL Compile Flow Specification

Any IXCOM transformation on a model that require hot-swap from software to hardware automatically resets the hardware-only-RTL marking on the model with a relevant warning.

A transformation needs hot-swap if it creates and depends on the values of the variables initialized in software at time zero or at the time of subsequent `tbcalls`. Such transformations will not work if the corresponding software model for this module is not generated. Hence, IXCOM resets the hardware-only-RTL flag (if marked) on such model to make the design functional.

IXCOM does not empty models that fall into any of the following categories:

- Requiring behavioral compilation transformations
- Having any GFIFO or SFIFO calls or `tb_import` or `tb_export` interface calls
- Having coverage enabled
- Including any assertions requiring software action block

Such models are treated as non-RTL models and are not emptied.

An external-netlist model cannot be marked as a `hwOnlyRtl` model. Appropriate error and warning messages are issued.

IXCOM preserves the exported object declaration in the `hwOnlyRtl` model. This is needed for resolving XMR references to such objects during Xcelium elaboration. However, the value in software is not updated from hardware unless it is explicitly exported.

XMR references to internal objects in the `hwOnlyRtl` model are not allowed from the following:

- C testbench
- Testbench models that are not visible to IXCOM\
- Models specified with the `+xcDesignTop` option
- Models used in the UVM compile flow

© 2021 All Rights Reserved.

IXCOM cannot detect such conditions and cannot issue any errors or warnings. These result in Xcelium elaboration failure while generating a snapshot.

Preserving the Software Models Having Initial Blocks

During hot-swap, all the `initial` blocks are executed on software side. If the signals in these blocks are initialized with non-zero values in the modules that are candidates for hardware-only RTL, the signals will be transferred with incorrect values, leading to a potential mismatch in the simulation and emulation behavior.

So, the software model of such DUT modules must be preserved for modules that meet all of the following conditions:

- Candidate for hardware
- Candidate for hardware-only RTL
- Have `initial` block(s)

Use the options mentioned in the table below to reset the hardware-only-RTL marking on the module.

Option	Description

`+hwOnlyExclInitMods+<modA>+<modB>`	Resets the hardware-only-RTL marking done by either of the following commands, if there is any initial block in the module:
	`vlan -hwOnylRtl`
	Or
	`ixcom +hwOnlyRtl+<modA>+<modB>`
	Or
	`ixcom +hwOnlyRtlTop+<modA>+<modB>`

© 2021 All Rights Reserved.

`+hwOnlyExclNonZeroInitMods +<modA>+<modB>`	Resets the hardware-only-RTL marking done by either of the following commands if there is any initial block in the module with possible non-zero initialization:
	`vlan -hwOnlyRtl`
	or
	`ixcom +hwOnlyRtl+<modA>+<modB>`
	or
	`ixcom +hwOnlyRtlTop+<modA>+<modB>`

Note: The `+hwOnlyExclNonZeroInitMods` option may not work as expected if you initialize the signals in initial block using XMRs. Whenever, the hardware-only-RTL marking is reset on a module, a warning message is dumped in the `ixcom.log` file. If module is a candidate of `+hwOnlyExclNonZeroInitMods` and `+hwOnlyExclInitMods`, then `+hwOnlyExclInitMods` option will supersede.

If there are any statements in the initial block which may result in non-zero initialization but IXCOM cannot precisely evaluate it, then IXCOM will cautiously dishonor the hardware-only-RTL marking on the module. Example 2 is one of the possible cases.

Following are some of the examples to illustrate the usage of the `+hwOnlyExclInitMods`[+<*module_name*>]* and `+hwOnlyExclNonZeroInitMods`[+<*module_name*>]* options.

Example 1:

```
module dut();
initial begin
    counter = 0;
end
endmodule
```

- In the above example, the software model is preserved because it has an `initial` block. The compilation command used is as follows:

  ```
  ixcom -ua +hwOnlyExclInitMods +dut+dut +hwOnlyRtl ...
  ```

- However, if the compilation of the above mentioned RTL in Example 1 is performed using the below mentioned command, then hardware-only-RTL marking on the module will not be reset. With the `+hwOnlyExclNonZeroInitMods` option, the IXCOM compiler will evaluate the initialization statements inside the initial block and if all the initialization evaluates to zero value, then hardware-only-RTL marking is honored.

  ```
  ixcom -ua +hwOnlyExclNonZeroInitMods +dut+dut +hwOnlyRtl ...
  ```


© 2021 All Rights Reserved.

Example 2:

```
module dut;
initial begin
    counter = 0;
    number = myFunction();
end

function int myFunction();
    return 0;
endfunction
endmodule
```

In the above example, the software model is preserved because it has an `initial` block with the possibly non-zero initialization. While evaluating the statements in the initial block, if IXCOM needs to perform deep nested evaluation, such as the function or task call, then IXCOM will not evaluate the deep nested calls, and the hardware-only-RTL marking will be dishonored.

```
ixcom -ua +hwOnlyExclNonZeroInitMods +dut+dut +hwOnlyRtl ...
```

Example 3:

```
module dut();
    initial begin
        #10 counter = 0;
    end

endmodule
```

In the above example, the hardware-only-RTL marking will be reset because the `initial` block contains the `#10` statement that results in cautiously preserving the software model of module when the following compilation command is issued:

```
ixcom -ua +hwOnlyExclNonZeroInitMods +dut+dut +hwOnlyRtl ...
```

Example 4:

```
module dut();
    initial begin
        counter = #10 0;
    end
endmodule
```

In the above example, hardware-only-RTL marking will be reset because the `initial` block contains the `#10` statement that results in cautiously preserving the software model of module when the following compilation command is issued:

```
ixcom -ua +hwOnlyExclNonZeroInitMods +dut+dut +hwOnlyRtl ...
```

Example 5:

```
module dut1();
   bit [31:0] counter;
end
endmodule

module dut2();
initial begin
```


© 2021 All Rights Reserved.

```
testbench.d1.counter = 10; //Here, d1 is the instance of dut1 in testbench module
end
endmodule
```

For the above example, the `+hwOnlyRtl` option in the following command is ignored for the `dut2` module, but not for the `dut1` module.

```
ixcom -ua +hwOnlyExclNonZeroInitMods +dut+dut +hwOnlyRtl ...
```

The `+hwOnlyRtl` option might not work as expected when signals are initialized in the form of XMR. If XMR signals are initialized within the same module that contains the actual signal, the `+hwOnlyRtl` option works as expected but if the initialization happens in a different module, then it might not work as expected. This is the limitation in the Lightweight flow with XMR-based initialization.

IXCOM Hardware-Only-RTL Run-time Checks

You cannot run software simulation in the Lightweight flow. You must directly swap-in to hardware at time zero. Running software simulation with hardware-only-RTL models might result in unexpected simulation behavior.

Run-time XMR references using VPI (`vpi_handle_by_name()`) or simulator or xeDebug TCL interface to objects inside the `hwOnlyRTl` model return `NULL` or `ERROR` for objects that are not `FOUND`. The IXCOM compiler cannot detect such conditions and you need to fix such references or revise the hardware-only-RTL markings.

The hierarchy skeleton is preserved in the Xcelium snapshot. So, scope access SystemVerilog DPI routines like `svGetScopeFromName()` and VPI (`vpi_handle_by_name(<instance-path-only>)`) work for the `hwOnlyRTl` models.

IXCOM Support for XMUTILS of Xcelium

This section describes the support for the Verilog tasks and VHDL procedures of the XMUTILS library in IXCOM for Xcelium native support.

The NCUTILS library has been renamed (or re-branded) in Xcelium tool and, consequently, IXCOM supports the re-branded name XMUTILS.

These libraries include tasks and functions that have been similarly re-branded:

Task	Branded name
nc_force	xm_force

© 2021 All Rights Reserved.

Task	Branded name
nc_release	xm_release
nc_deposit	xm_deposit
nc_mirror	xm_mirror

XLM supports the `ncutils` tasks (`nc_force`, `nc_release`, `nc_deposit` and `nc_mirror`) with deprecated warning message, and suggests use of `xm_force`, `xm_release`, `xm_deposit`, and `xm_mirror`, respectively.

Important

New task and procedure names are supported only in Xcelium releases; not in Xcelium releases.

XLM releases and IXCOM support both `xm_*` and `nc_*` tasks, that is `xm_force`, `xm_release`, `xm_deposit` and `xm_mirror` tasks in addition to the old `nc_force`, `nc_release`, `nc_deposit` and `nc_mirror` tasks.

The use model for the `xmutils` task and procedures is similar to the `ncutils` task and procedures.

© 2021 All Rights Reserved.

Importing External Netlists in the SA Mode

To compile the behavioral and RTL portions of the design, the IXCOM compiler performs design analysis, elaboration, synthesis checks to map portions of the design that can be accelerated in emulator hardware.

If the source files contain netlists either generated by HDL-ICE Compiler or produced by third-party tools, the above-mentioned compile process is very expensive - both in terms of memory consumption and the time required to compile the design. Also, such a compile process is not productive for netlists because unlike the behavioral and RTL modules (that require analysis, elaboration and synthesis), the gate-level netlists are pre-synthesized and can be passed to the back-end compiler as-is.

For synthesizable netlist files, these checks and processes are unnecessary, and the netlist can be directly presented to the hardware-mapping process of the compiler.

A netlist is a hierarchical representation of a design where each of the module in the hierarchy exclusively comprises component instances. All other module definitions containing UDPs, continuous assignments, procedural statements, and so on must be compiled as RTL using IXCOM tools.

Note: A net is considered to be a netlist, if and only if, all of its drivers belong to the netlist partition. So, a bidirectional net on the boundary between RTL and netlist partitions is considered to be an RTL net, and netlist-specific options do not apply to such a net.

This section describes the process for importing external netlists in SA mode using the IXCOM compiler.

This process enables the following:

- Instantiation of external netlist modules in RTL hierarchy
- Instantiation of RTL modules in external netlist hierarchy
  - Using Direct Instantiation
  - Using SystemVerilog Bind
- Hierarchical references from RTL hierarchy to access objects in the external netlist hierarchy
- Hot-swap with external netlists
- Netlist import in ICE mode

© 2021 All Rights Reserved.

An alternative mechanism to import netlists as blackboxes into the IXCOM compiler is described in Appendix 37, "Importing Netlists as Blackboxes in SA Mode." However, this mechanism has certain limitations:

- Certain options like `Q_ASSIGN` and bus-resolution options used for netlists do not comply with the IEEE standard semantics.

  The external nets need to be specificied using files in the compilation directory.

- External netlist object reference and binding specifications require usage of `$ixc_ctrl` statements in the shell module.

- Additional hierarchy is introduced for the netlist hierarchy due to shell module for the netlist module. This limitation requires all hierarchical references to netlist objects to be modified in the compile and run-time scripts for including the IXCOM-generated hierarchy.

Therefore, the mechanism of importing netlists as blackboxes is not recommended, and is likely to be obsolete in future release. All of these limitations are addressed by the process of importing external netlists described in the following section.

Using the Configuration File for Specifying Netlist Information

To use this mechanism, create a configuration file with directives to specify the netlist and the processing of the netlist by IXCOM. You can choose to specify detailed information about the netlist so that IXCOM does not need to do too much processing, or you can choose to specify just the minimum required information, and let IXCOM process the netlist. Note that if IXCOM needs to process the netlist, there will be a compilation-time overhead to compile the netlist. However, the amount of information that you need to gather and specify to IXCOM will be considerably less. So, there is a trade off between the increase in compilation time and the ease of use with all the available directives.

Use the `-externalNetlists` <*file*> option of the `ixcom` command to specify the configuration file.

In the configuration file, you can specify one or more of the following information:

- List of netlist files
- List of netlist files to compile or the compile option
- List of references to pre-compiled netlist libraries
- List of netlist design units to be imported
- List of RTL design units to be exported into the netlist hierarchy

© 2021 All Rights Reserved.

- List of files containing SystemVerilog bind statements for inserting RTL module instances into netlist hierarchy
- List of hierarchical references to netlist objects
- Hot-swap capability for the netlist hierarchy
- Netlist module definition to be kept if the definition is present in both the netlist and RTL domains
- Back-end compiler-control options

Netlist Configuration File Specification

To specify the netlist information in the configuration file, use the following configuration file keywords:

`NL_FILE`, `NL_FILE_HWONLY`, `NL_FILE_SWONLY`, `NL_COMPILE`, `NL_REFLIB`, `NL_IMPORT`, `NL_EXPORT`, `NL_BIND`, `NL_REF`, `NL_HOTSWAP`, `NL_DEF_KEEP`, `NL_HDLICE_REF`, `NL_XMVLOG_OP`, `NL_IMPORT_LIB`, and `NL_INFO`

Note: This document uses uppercase names for keywords in the configuration specification file. However, these are case-insensitive keywords.

You can specify the configuration file items multiple times in the configuration file. To specify line comments, use "`//`" or "`#`" at the beginning of the line.

For details about the syntax of these keywords, refer to the description of the `-externalNetlists <file>` option of the `ixcom` command in the C*ommands and System Tasks for SA Mode* chapter of the *VXE Command Reference Manual*.

Netlist Compilation

The IXCOM compiler provides a use model to analyze the netlist files into a logical library and refer to pre-analyzed netlist libraries during elaboration time. The analyzed information will be used by the compiler to extract netlist object access and design unit binding information. After collecting the required information for the netlist compile, the netlist files are passed to the back-end tools for the final design elaboration.

This feature is intended to improve ease of use to compile external netlists with minimum input from you. The netlist compile capability will have added compile-time overhead to analyze specified netlist files and elaboration of required netlist hierarchy.

The following sections provide use-model details for compiling netlist files.

© 2021 All Rights Reserved.

Netlist Compilation using ixcom Command

The netlist compile information can be specified in a configuration file as follows:

`NL_COMPILE all`

This configuration enables you to compile all netlist files specified using `NL_FILE` and `NL_FILE_HWONLY`.

`NL_COMPILE <file> [ <file> ... ]`

This configuration enables you to compile only specified list of netlist files. This use model is intended to minimize compile time overhead for designs that do not require all netlist files to be analyzed.

IXCOM internally invokes the `vlan` tool to parse netlist files as described in the following section - Netlist Analysis using vlan Command. The netlist files are analyzed into an internal logical library.

Netlist Analysis using vlan Command

To analyze the netlist files separately using ***vlan***, specify the `-netlist` option with the `vlan` command:

```
vlan -netlist <netlist_source_files>
```

All other analyzer options can also be specified in the above command. The netlist files can be analyzed into different logical libraries (similar to RTL compilation) using the `-work <library>` option. The default logical library (`WORK`) is used if the logical library is not explicitly specified.

All source files specified in the command-line are analyzed as netlist files and the analyzed information is saved into a database in a compact format, which is different from the format used by RTL compilation. Also, the database files for netlist files are kept separate from RTL files compiled into the same logical library.

Referring to Netlist Libraries with the ixcom Command

To compile netlist files using ixcom command, specify the following information in the configuration file:

`NL_REFLIB <lib> [ <lib> ... ]`

This configuration enables you to specify list of logical libraries to refer (or search) to for pre-compiled netlists.

For details, refer to Using the Configuration File for Specifying Netlist Information.

© 2021 All Rights Reserved.

For compiles that refer to multiple logical libraries, the design units are bound to component instances in the list similar to binding and search rules as described in Design Binding and Search Rules in IXCOM section. The libraries are searched in the order specified by `NL_REFLIB` specification.

Verbose Messaging

By default, the compilation process reports minimal information during the netlist compilation process. The verbose messaging for the netlist compilation can be enabled using the following configuration:

```
NL_INFO verbose
```

Importing Netlist Modules

You can import a netlist module as a:

- Top-level DUT module
- Top-level DUT module instantiated in the RTL testbench (software partition)
- Non-top DUT module instantiated in the RTL DUT (hardware partition)

For the RTL hierarchy, the design units can be written in any of languages (Verilog, SystemVerilog and VHDL) supported by the IXCOM compiler.

To import netlist modules, you must specify netlist information in the configuration file using one of the approaches listed below:

1. Compiled netlist information as described in the Netlist Compilation using ixcom Command section or the Referring to Netlist Libraries with the ixcom Command section.
2. Specify explicit list of netlist modules using `NL_IMPORT`.

Note: In the first approach, IXCOM compiler determines the netlist modules to import using the compiled information. This approach is recommended to bring up the design.

Importing Netlist Modules using Compiled Netlist Information

You can import netlist modules using information from the compiled netlist as described in the Netlist Compilation section described earlier. The IXCOM compiler infers the following information from the compiled netlist:

- Netlist modules to import

© 2021 All Rights Reserved.

- Netlist module I/O interface for the imported module
- Netlist object access from RTL hierarchy using hierarchical reference
- Netlist module definition conflict with RTL module definition

To import an external netlist module using compiled netlist information, specify one of the the following categories of information in the configuration file:

- `NL_FILE`, `NL_FILE_HWONLY`, or `NL_FILE_SWONLY` to specify netlist files and `NL_COMPILE` to specify netlist compile option or list of files to compile.

  Or,

- `NL_REFLIB` to specify that the pre-compiled (or analyzed) netlist files should be searched separately using `vlan`. If you have specified `NL_COMPILE` in the configuration file, `NL_REFLIB` does not have any effect.

  Note: The compiler extracts the netlist files for you if netlist files are not specified using `NL_FILE`.

You can also optionally specify `NL_HOTSWAP` to enable hot-swap for the design. Refer to Using Hot Swap with Imported Netlists for details.

The IXCOM compiler presents the netlist files specified using `NL_FILE` and `NL_FILE_HWONLY` to the back-end compiler. The netlist files specified using `NL_FILE` and `NL_FILE_SWONLY` are presented to Xcelium compile process if hot-swap is enabled.

The compiled netlist information is used by IXCOM compiler to import netlist module as follows:

> During elaboration of RTL hierarchy, the compiler searches definitions in netlist for any unbound module instances in the RTL. If a module is found in the netlist domain, it is marked as an imported module, and a corresponding information message is displayed. For all netlist modules marked for import, the compiler extracts the I/O interface details for the netlist module.

This section describes the process of importing netlist module (`Dut`). This section uses the following example to illustrate the netlist module import into the RTL hierarchy.

// File: top.sv

```
module Top;
   // RTL definition
      Dut dut(clk, din, dout);
   // ...
endmodule // Top
```

// File: dut-nl.v

© 2021 All Rights Reserved.

```
module Dut(clk, din, dout)
  // Netlist definition
  Blk blk(clk, din, dout);
  // ...
endmodule // Dut
```

// File: blk-nl.v

```
module Blk(clk, din, dout)
   // Netlist definition
   // ...
endmodule // Blk
```

The following configuration file is used to illustrate compile steps to import netlist modules.

// File: nl-config-compile.spec

```
NL_FILE dut-nl.v  blk-nl.v          // list of netlist files
NL_COMPILE all                      // compile all netlist files
```

// File: nl-config-reflib.spec

```
NL_REFLIB NL_work                   // logical library for pre-compiled (or
                                    // analyze) netlist library to refer to
```

1. To import `Dut` as the top-level DUT module, use the following command:

   ```
   vlan -netlist -work NL_work dut-nl.v  blk-nl.v
   ixcom -top Dut -ua +dut+Dut -externalNetlists nl-config-reflib.spec
   ```

   Note: This use model requires separate analysis using `vlan` and elaboration using `ixcom`. This is so because IXCOM compilation requires either the source files or the top-level design unit specified using the `-top` option of the `ixcom` command.

2. To import `Dut` as the top-level DUT module instantiated in the testbench (`Top`), use one of the following approaches:

   ```
   ixcom top.sv –ua +dut+Dut -externalNetlists nl-config-compile.spec
   ```

   Note: In the above command, design units are analyzed and elaborated using IXCOM. The netlist files are compiled using IXCOM.

   Or,

   ```
   vlan top.sv
   ixcom -top Top –ua +dut+Dut -externalNetlists nl-config-compile.spec
   ```

   Note: In the above command, design units are separately analyzed (using `vlan`) and elaborated (using `ixcom`). The netlist files are compiled using IXCOM.

   Or,

   ```
   vlan top.sv
   vlan -netlist -work NL_work dut-nl.v blk-nl.v
   ixcom -top Top –ua +dut+Dut -externalNetlists nl-config-reflib.spec
   ```

   Note: In the above command, design units are separately analyzed (using `vlan`) and

© 2021 All Rights Reserved.

elaborated (using `ixcom`). The netlist files are separately analyzed using vlan and referred to in the configuration file.

3. To import `Dut` as a non-top DUT module within the DUT hierarchy (`Tb`), use one of the following approaches:

```
ixcom top.sv –ua +dut+Tb -externalNetlists nl-config-compile.spec
```

Note: In the above command, design units are analyzed and elaborated using IXCOM. The netlist files are compiled using IXCOM.

Or,

```
vlan top.sv
ixcom -top Top –ua +dut+Tb -externalNetlists nl-config-compile.spec
```

Note: In the above command, design units are separately analyzed (using `vlan`) and elaborated (using `ixcom`). The netlist files are compiled using IXCOM.

Or,

```
vlan top.sv
vlan -netlist -work NL_work dut-nl.v blk-nl.v
ixcom -top Top –ua +dut+Tb -externalNetlists nl-config-reflib.spec
```

Note: In the above command, design units are separately analyzed (using `vlan`) and elaborated (using `ixcom`). The netlist files are separately analyzed using vlan and referred to in the configuration file.

You can choose to provide only a partial set of netlist file(s) to extract the above information to reduce the netlist compile overhead.

For the above example, configuration file can be specified as:

// File: nl-confing-compile.spec

```
NL_FILE dut-nl.v blk-nl.v
NL_COMPILE dut-nl.v
```

For partially compiled netlist files, the compiler reports errors if an access is made to non-existent netlist hierarchies or objects.

To import netlist module using compiled netlist libraries, the following design compilation requirements apply:

- If you specify `NL_IMPORT` explicitly, the compiler generates error messages for any unbound instances found in the RTL domain. You do not need to specify `NL_IMPORT`.
- To import a netlist module into VHDL RTL hierarchy, you must provide component declaration for the netlist module in the VHDL entity that instantiates the netlist module.

© 2021 All Rights Reserved.

- If both NL_COMPILE and NL_REFLIB specifications are present in the configuration file, the compiler ignores NL_REFLIB specification with a warning message.

Importing Netlist Modules with Explicit Information

You can import netlist module with explicit information without compiling netlist files. This use model is provided for back-ward compatibility and can also be used to avoid netlist compile overhead. However, because in this approach IXCOM does not compile the netlist files, you must provide the information required to import the netlist modules explicitly in the configuration file. With this approach, you also need to explicitly specify references to netlist objects from the command line or RTL. Therefore, this approach is not recommended to bring up the design.

To import an external netlist module, you must specify the following information in the configuration file:

- The netlist module(s) using the NL_IMPORT keyword.

  Note: Only the top-level netlist module name is required for the netlsit hierarchy to be imported.

- The netlist file(s) using the NL_FILE, NL_FILE_HWONLY, or NL_FILE_SWONLY
- An empty shell module with the complete interface (port I/O) declaration for the imported netlist module to the IXCOM compiler in RTL. This shell module is required for IXCOM compiler to build the accurate I/O interface.

This section describes the process of importing netlist module (Dut) with explicit information.

This section uses the following example to illustrate the netlist module import into RTL hierarchy.

// File: top.sv

```
module Top;
   // RTL definition
   Dut dut(clk, din, dout);
   // ...
endmodule // Top
```

// File: dut-shell.v

```
module Dut(clk, din, dout)
   input clk;
   input [3:0] din;
   output [3:0] dout;
      // Empty shell module (RTL)
endmodule // Dut
```


© 2021 All Rights Reserved.

// File: dut-nl.v

```
module Dut(clk, din, dout)
   // Netlist definition
   Blk blk(clk, din, dout);
   // ...
endmodule // Dut
```

// File: blk-nl.v

```
module Blk(clk, din, dout)
   // Netlist definition
endmodule // Blk
```

The following configuration file is used to illustrate the compile steps to import netlist modules.

// File: nl-config.spec

```
NL_FILE dut-nl.v blk-nl.v
NL_IMPORT Dut
```

1. To import `Dut` as the top-level DUT module, use one of the following approaches:

   ```
   ixcom dut-shell.v -ua +dut+Dut -externalNetlists nl-config.spec
   ```

   Note: In the above command, design units are analyzed and elaborated using IXCOM. The netlist files are compiled using IXCOM.

   Or,

   ```
   vlan dut-shell.v
   ixcom -ua +dut+Dut -top Dut -externalNetlists nl-config.spec
   ```

   Note: In the above command, design units are separately analyzed (using `vlan`) and elaborated (using `ixcom`). The netlist files are compiled using IXCOM.

2. To import `Dut` as the top-level DUT module (or VHDL entity) instantiated in the testbench (Top), use one of the following approaches:

   ```
   ixcom top.sv dut-shell.v -ua +dut+Dut -external nl-config.spec
   ```

   Note: In the above command, design units are analyzed and elaborated using IXCOM. The netlist files are compiled using IXCOM.

   Or,

   ```
   vlan top.sv dut-shell.v
   ixcom -ua +dut+Dut -top Top -external nl-config.spec
   ```

   Note: In the above command, design units are separately analyzed (using `vlan`) and elaborated (using `ixcom`). The netlist files are compiled using IXCOM.

3. To import Dut as a non-top DUT module (or VHDL entity) within the DUT hierarchy (Top), use the following command:

   ```
   ixcom top.sv dut-shell.v -ua +dut+Top -externalNetlists nl-config.spec
   ```


© 2021 All Rights Reserved.

Note: In the above command, design units are analyzed and elaborated using IXCOM. The netlist files are compiled using IXCOM.

Or,

```
vlan top.sv dut-shel.v
ixcom -ua +dut+Top -top Top -externalNetlists nl-config.spec
```

Note: In the above command, design units are separately analyzed (using `vlan`) and elaborated (using `ixcom`). The netlist files are compiled using IXCOM.

To import netlist module with explicit information, the following design compilation requirements apply:

- The configuration file must have at least one `NL_IMPORT` specification. All other specifications are optional depending on the design compile requirements.
- The netlist modules that you are importing into the IXCOM compiler must not include any ATB set of constructs. Refer to Synthesizing Designs with Acceleratable Testbench (ATB) Constructs for details about ATB constructs.
- The back-end compile process limitations apply for the imported netlist. A known limitation is:

  Hierarchical forward reference to a non-scalar object is not allowed in a port connection.
- The netlist files must be specified using `NL_FILE`, `NL_FILE_HWONLY`, or `NL_FILE_SWONLY`.

  Note: This specification is optional if netlist module is imported into ICE environment as described in Importing Netlists Using the XEL Scripts for ICE-Mode Compilation.

Exporting RTL Modules into the Netlist Hierarchy

The IXCOM compiler use model enables RTL module instantiations within the netlist hierarchy. This use model refers to such modules as *exported RTL modules*. To export RTL modules into the netlist hierarchy, RTL modules must be compiled using the IXCOM compiler. The RTL module instances in the netlist hierarchy are elaborated into the design during the back-end (`xeCompile`) compilation process.

You can export an RTL module into the netlist hierarchy using the following methods:

- ❑ Direct instantiation of the RTL module within the netlist hierarchy.
- ❑ Using the explicit binding that you have specified to insert the RTL-module instantiation into the netlist hierarchy or overwrite the existing RTL instantiation in the netlist hierarchy.

© 2021 All Rights Reserved.

Note: This use model is intended for requirements like:

- Reusing RTL memory models.
- Using SA features like assertions and coverage by inserting monitor and checker module instances into the netlist.

Caution

Netlist import into an exported RTL hierarchy is not supported.

This section uses the following example to illustrate exporting of RTL modules into the imported netlist hierarchy.

// File: top.sv

```
module Top;
   // RTL definition
   Dut dut(clk, din, dout);
   // ...
endmodule // Top
```

// File: mem.sv

```
module Mem(addr, data, read, write);
   // RTL memory model
endmodule // Mem
```

// File: bind.sv

```
bind Tb.dut.blk Chk chk(clk, din, dout); // SV bind statement
```

// File: chk.sv

```
module Chk(clk, din, dout);
   // RTL definition
endmodule // Chk
```

// File: dut-nl.v

```
module Dut(clk, din, dout)
   // Netlist definition
   Blk blk(clk, din, dout);
   Mem mem(addr, data, read, write);
                     // Direct instantiation of RTL memory model //
endmodule // Dut
```

// File: blk-nl.v

```
module Blk(clk, din, dout)
   //Netlist definition
endmodule // Blk
```


© 2021 All Rights Reserved.

Exporting RTL Modules using Compiled Netlist Information

You can choose to compile netlist files as described in Netlist Compilation using ixcom Command, Netlist Analysis using vlan Command, and Referring to Netlist Libraries with the ixcom Command sections described earlier. The IXCOM compiler infers the following cross-domain library from the compiled netlist:

- Netlist modules to import
- Netlist module I/O interface for the imported module
- RTL module to export into netlist hierarchy
- Netlist object access from RTL hierarchy using hierarchical reference
- Netlist module definition conflict with RTL module definition

To import an external netlist module into RTL hierarchy with directly instantiated RTL modules in the netlist hierarchy, you must specify the following information in the configuration file:

- `NL_FILE`, `NL_FILE_HWONLY`, or `NL_FILE_SWONLY` to specify netlist files. `NL_COMPILE` to specify netlist compile option or list of files to compile.

Or,

- `NL_REFLIB` must be specified to search pre-compiled (or analyzed) netlist files separately using `vlan`. This specification has no effect if `NL_COMPILE` is also specified in the configuration file.

  Note: The compiler extracts the netlist files if netlist files are not specified using `NL_FILE`.

The compiled netlist library is used by IXCOM compiler to import netlist module and bind RTL module instance(s) as follows:

- During elaboration of RTL hierarchy, the compiler searches definitions in netlist for any unbound module instances in the RTL. If a module is found in the netlist domain, it is marked as an imported module, and a corresponding information message is displayed. For all netlist modules marked for import, the compiler extracts the I/O interface details for the netlist module.
- During elaboration of the netlist hierarchy, the compiler searches definitions in RTL for any unbound instances in the netlist hierarchy. The compiler also searches instances specified in the SV bind statement. If a module is found in the RTL domain, it is marked as an exported module.

This section describes the process of exporting RTL module into the netlist hierarchy using the compiled netlist information. In the following example, the netlist module (`Dut`) is imported

© 2021 All Rights Reserved.

into RTL hierarchy (`Top`). The RTL modules `Mem` and `Chk` are compiled using the IXCOM compiler to export into the netlist hierarchy. The back-end compiler (`xeCompile`) binds the exported module into the netlist hierarchy in the final elaborated design. For the purpose of documentation, the direct instantiation is used to bind `Mem` into the netlist hierarchy. SystemVerilog bind is used to bind `Chk` into netlist hierarchy. You can choose either or both of the approaches to export RTL modules.

The use model examples use the following configuration files:

// File: nl-config-compile.spec

```
NL_FILE dut-nl.v  blk-nl.v
NL_COMPILE all
```

// File: nl-config-reflib.spec

```
NL_REFLIB NL_work
```

1. To import `Dut` as the top-level DUT module, use the following command:

   ```
   vlan chk.sv mem.sv bind.sv
   vlan -netlist -work NL_work dut-nl.v blk-nl.v
   ixcom -top Dut –ua +dut+Dut -externalNetlists nl-config-reflib.spec
   ```

   Note: This use model requires separate analysis using `vlan` and elaboration using `ixcom`.

2. To import `Dut` as the top-level DUT module instantiated in the testbench (`Top`), use the following command:

   ```
   ixcom top.sv chk.sv mem.sv bind.sv -ua +dut+Dut \
         -externalNetlists nl-config-compile.spec
   ```

   Note: In the above command, design units are analyzed and elaborated using IXCOM. The netlist files are compiled using IXCOM.

   Or,

   ```
   vlan top.sv chk.sv mem.sv bind.sv
   ixcom -top Top –ua +dut+Dut -externalNetlists nl-config-compile.spec
   ```

   Note: In the above command, design units are separately analyzed (using `vlan`) and elaborated (using `ixcom`). The netlist files are compiled using IXCOM.

   Or,

   ```
   vlan top.sv chk.sv mem.sv bind.sv
   vlan -netlist -work NL_work dut-nl.v blk-nl.v
   ixcom -top Top –ua +dut+Dut -externalNetlists nl-config-reflib.spec
   ```

   Note: In the above command, design units are separately analyzed (using `vlan`) and elaborated (using `ixcom`). The netlist files are compiled using IXCOM.

3. To import `Dut` as a non-top DUT module within the DUT hierarchy (`Tb`), use the following command:

© 2021 All Rights Reserved.

```
ixcom top.sv chk.sv mem.sv bind.sv -ua +dut+Dut \
       -externalNetlists nl-config-compile.spec
```

Note: In the above command, design units are analyzed and elaborated using IXCOM. The netlist files are compiled using IXCOM.

Or,

```
vlan top.sv chk.sv mem.sv bind.sv
ixcom -top Top -ua +dut+Top -externalNetlists nl-config-compile.spec
```

Note: In the above command, design units are separately analyzed (using `vlan`) and elaborated (using `ixcom`). The netlist files are compiled using IXCOM.

Or,

```
vlan top.sv chk.sv mem.sv bind.sv
vlan -netlist -work NL_work dut-nl.v  blk-nl.v
ixcom -top Top -ua +dut+Top -externalNetlists nl-config-reflib.spec
```

Note: In the above command, design units are separately analyzed (using `vlan`) and elaborated (using `ixcom`). The netlist files are compiled using IXCOM.

For this use model, the following design compile requirements apply:

- All design compile requirements mentioned in Importing Netlist Modules using Compiled Netlist Information.
- RTL module must have HDL instance within the netlist hierarchy for direct instantiation. The direct instantiation method does not support parameterized instances.
- RTL module file(s) must be analyzed and elaborated using IXCOM compiler.

  Note: RTL modules can have any level of instance hierarchy, design construct and language as supported by the IXCOM compiler.
- In addition to SystemVerilog language rules, the following additional restrictions apply to explicit binding using SystemVerilog bind directive:
  - Must be used only to bind exported RTL module(s) in netlist hierarchy
  - Module instantiation and port connection syntax is limited to Verilog syntax. SystemVerilog feature like dot-start (`.*`) port connection is not allowed. This restriction is applicable only for `NL_BIND`.
  - RTL module instances can have parameterization specified.

Exporting RTL modules with Explicit Information

You can export RTL modules into the netlist hierarchy by specifying explicit information without compiling the netlist files.

© 2021 All Rights Reserved.

This use model is provided for back-ward compatibility and can also be used to avoid netlist compile overhead. However, because in this approach IXCOM does not compile the netlist files, you must provide the information required to export the RTL modules into the netlist hierarchy. With this approach, you also need to explicitly specify references to netlist objects from the command line or RTL. Therefore, this approach is not recommended to bring up the design.

For this use model, you must specify the following information in the configuration file:

- Netlist module(s) using the `NL_IMPORT` keyword.
- RTL module(s) using the `NL_EXPORT` keyword.

  You can also specify a comma-separated list of hierarchical paths to module instance with the `NL_EXPORT` keyword. This information is optional, but it is necessary for the module(s) that generate primary I/O signals. The IXCOM compiler generates primary I/O signals if the exported RTL module needs to interact with the software partition (or the testbench).
- Netlist file(s) using the `NL_FILE`, `NL_FILE_HWONLY`, or `NL_FILE_SWONLY` keywords.
- `NL_BIND` to specify files containing SystemVerilog bind statements

  Note: The IXCOM compiler does not process SystemVerilog bind statements specified in the file. The bind statements in the file are processed during the back-end compile process.

This section describes the process of exporting RTL module into the netlist hierarchy using the compiled netlist information. In the following example, the netlist module (`Dut`) is imported into RTL hierarchy (`Top`). The RTL modules `Mem` and `Chk` are compiled using the IXCOM compiler to export into the netlist hierarchy. The back-end compiler (`xeCompile`) binds the exported module into the netlist hierarchy in the final elaborated design. For the purpose of documentation, the direct instantiation is used to bind `Mem` into the netlist hierarchy. `NL_BIND` specification is used to bind `Chk` into netlist hierarchy. You can choose either or both of the approaches to export RTL modules.

The use model examples use the following configuration file:

// File: nl-config.spec

```
NL_FILE dut-nl.v  blk-nl.v
NL_IMPORT Dut
NL_EXPORT Chk : Tb.dut.blk.chk      // exported using direct instantiation
NL_EXPORT Mem : Tb.dut.mem          // exported using NL_BIND  statement
NL_BIND bind.sv                     // SV bind statements
```

1. To import Dut as the top-level DUT module, use the following command:

© 2021 All Rights Reserved.

```
vlan dut-shell.v chk.sv mem.sv
ixcom -top Dut -ua +dut+Dut -externalNetlists nl-config.spec
```

2. To import `Dut` as the top-level DUT module instantiated in the testbench (`Top`), use the following command:

```
ixcom top.sv dut-shell.v chk.sv mem.sv -ua +dut+Dut \
      -externalNetlists nl-config.spec
```

Or,

```
vlan top.sv dut-shell.v chk.sv
ixcom -top Top -ua +dut+Dut -externalNetlists nl-config.spec
```

3. To import `Dut` as a non-top DUT module within the DUT hierarchy (`Tb`), use the following command:

```
ixcom top.sv dut-shell.v chk.sv mem.sv -ua +dut+Dut \
      -externalNetlists nl-config.spec
```

Or,

```
vlan top.sv dut-shell.v chk.sv mem.sv
ixcom -top Top -ua +dut+Top -externalNetlists nl-config.spec
```

For this use model, the following design compile requirements apply:

- All design compile requirements mentioned in Importing netlist modules using compiled netlist information.
- RTL module must have HDL instance within the net-list hierarchy for direct instantiation. The direct instantiation method does not support parameterized instances.
- RTL module file(s) must be analyzed and elaborated using IXCOM compiler.

  Note: RTL modules can have any level of instance hierarchy, design construct and language as supported by the IXCOM compiler.

- In addition to SystemVerilog language rules, the following additional restrictions apply to explicit binding using `NL_BIND` directive:
  - Must be used only to bind exported RTL module(s) in netlist hierarchy
  - Module instantiation and port connection syntax is limited to Verilog syntax. SV feature like dot-start (.*) port connection is not allowed
  - Only instance based binding is allowed
  - RTL module instances can have parameterization specified

  The back-end compiler applies the following rules for binding instances:

  - RTL module instance(s) specified in bind directive will be inserted into target hierarchy if no instances with the same name are found in the target hierarchy.

© 2021 All Rights Reserved.

- RTL module instance(s) specified in bind directive will overwrite existing instance(s) with the same name in the target hierarchy. Compiler will report a warning message for this case.

  Note: This is a non-standard tool behavior and cannot be used if hot-swap is required for the design. You must manually resolve bind instance name collision.

- RTL module instance name collision with the other (non-instance) objects in the target hierarchy results in error- or tool-dependent behavior.

Importing Netlists Using the XEL Scripts for ICE-Mode Compilation

If you have previously written compilation scripts for ICE mode, you can easily edit these scripts to import the netlists generated by the IXCOM compiler.

For this, use the `-target` option of the `ixcom` command to compile the design in stages:

```
ixcom [ options ] -target hdlice
```

The above `ixcom` command ensures that the IXCOM compiler stops after the HDL-ICE Compiler stage and generates an import script in the compilation directory:

```
xc_work/nl/import.qel
```

This compile script only includes the step to import the netlist files specified using the `NL_FILE` keyword together with the `xc_va.ver file` generated during the IXCOM compilation. In the `xcva_top` file, the IXCOM compiler includes the top-level DUT modules specified with the `NL_IMPORT` keyword.

To enable the import of the IXCOM-genenerated netlist, the IXCOM compiler writes the following commands in the compilation script for ICE mode:

- A list of reference libraries:

  ```
  refLib qtref generic
  ```

- Netlist import options:

  ```
  importOption { library hdldb }
  ```

- A list of net-list file (HDL-ICE or third-party net-list files), using:

  ```
  netlistFile {...}
  ```

Source the above-generated import script:

```
source xc_work/nl/import.qel
```

Import the design using the following command:

```
designImport
```


© 2021 All Rights Reserved.

Specify the top-level design name as follows:

```
design hdldb xcva_top
```

Issue the remaining precompile and compile commands as required for the ICE compilation.

Following is a sample ICE compilation script:

```
# ICE compilation script
# Reference libraries
# NOTE: The following reference libraries must be specified in addition
# to the reference libraries required to import other netlist file(s)
refLib qtref generic
#...
# Import options
# NOTE: The following library option must be specified
importOption {library hdldb}
#...
# Specify netlist files using netlistFile
# netlistFile {...}
# Import the netlist file(s) generated by the IXCOM compiler
source xc_work/nl/import.qel
designImport
# Specify the top-level design module
# NOTE: IXCOM uses "xcva_top" as top level design unit name.
design hdldb xcva_top
```

Continue and complete the back-end compile stage (`xeCompile`) using the next stage with the `ixcom` command:

```
ixcom -target xecompile
```

For the above command, you need to specify the ICE-compilation script as required by the IXCOM compiler by using one of the following ways:

- Include `./compile.qel` in the compilation directory
- Use the `ixcom -xeCompile compile=<file>` command

  It is highly recommended to use the `ixcom` command above to invoke `xeCompile`.

If you have invoked xeCompile directly from the ICE-compilation script, you need to specify the IXCOM options listed below explicitly by using the native XEL user data or commands:

```
+1xua
+keepregs
-lps <file>
-xeCompile <arg>
-xecompileargs <string>
```

The IXCOM compiler also updates the design-partition information after the back-end compilation stage. So, you need to invoke the compiler from the following update stage prior to running the design on hardware:

```
ixcom -target postXeCompile
```


© 2021 All Rights Reserved.

To continue and complete the software-model compilation using Xcelium, use the following command:

```
ixcom -target swmodel
```

With the netlist configuration file, you cannot specify additional libraries for the third-party netlist file(s). You must specify the reference in the ICE-compilation script (See the refLib command description in the *VXE Command Reference Manual* for details on specifying reference libraries).

Generating Skeleton Design Netlists for IXCOM

A new -parallelNL option has been added to ixcom command to generate xc_make file that helps HDL-ICE Compiler create skeleton design netlist. This option also generates ixcom.qel script for xeCompile, so it can import skeleton design netlists.

Using Hot Swap with Imported Netlists

The hot-swap feature provides the ability to move the run-time database from hardware (run using emulator) to software (run using Xcelium) and vice-versa during the SA run.

By default, the hot-swap capability is disabled for the netlist hierarchy. To enable it, include the following specification in the configuration file:

```
NL_HOTSWAP on
```

If hot-swap is enabled, the IXCOM compiler presents the netlist file(s) (specified using the NL_FILE and NL_FILESWONLY keywords) to Xcelium with the required binding specification and HDL-ICE Compiler reference file.

If you enable hot-swap capability for the netlist, the IXCOM compiler needs to pass the netlist model to Xcelium and create a simulation congruent software model. This requires the netlist and all of its components (ASIC technology libraries) to be available in a simulation-compatible form. IXCOM passes the netlist and other hotswap simulation files to Xcelium or xmvlog for compilation. If the netlist is generated by HDL-ICE Compiler, IXCOM automatically includes the QTREF simulation library.

The directive NL_FILE_SWONLY configuration keyword identifies the files that are to be used only for the simulation hot swap model. The NL_XMVLOG_OPT configuration keyword is used with the NL_HOTSWAP ON configuration keyword only to pass the compile argument to xmvlog for the compilation of the simulation hot swap model.

Note: Hot-swap capability is available only when a netlist is imported into an RTL module, which means that the netlist should not be the top-level design unit, nor the top level in the emulator.

© 2021 All Rights Reserved.

Using the hot-swap feature with netlist import has the following limitations:

- The netlist file(s) presented to the IXCOM compiler must have simulation-compatible model as supported by Xcelium tools.

  The RTL-module-instance binding using the SystemVerilog bind feature that requires *instance overwrite* is not allowed. So, you need to manually edit the netlist file and make it compliant with the SystemVerilog language requirements for Xcelium compilation.

- The `generate` construct within the exported RTL hierarchy is not supported with the hot-swap feature. This limitation is because of the incompatibilities between the scope names in the hardware and software partitions.

- The current version of the HDL-Compiler reference file provided in the installation directory (`$<IXCOM_INSTALL_HOME>/share/vxe/etc/ixcom/ixcom_hdlice_ref.vp`) does not match the Palladium Z1 execution model, and contains cells that are not fully congruent with the simulation requirements.

  For hot-swap, you can use the simulation-congruent reference file (`$<IXCOM_INSTALL_HOME)/share/vxe/etc/ixcom/hdlice_ref.sv`) using the following specification:

  ```
  NL_HDLICE_REF init_zero
  ```

  This specification initializes state devices in the reference file to zero to make cells simulation congruent.

With the netlist configuration file, you cannot specify additional libraries for the third-party netlist file(s). You must specify the reference libraries and files for the Xcelium compilation by using the `XMVLOGOPTS` and `XMELABOPTS` options in the `./uxc_hdl.var` file.

To specify Xcelium (`xmvlog`) options for netlist compilation, use the following specification in the external netlist configuration file:

```
NL_XMVLOG_OPT <option> ...
```

The IXCOM compiler collects the list of options and makes it available for the xmvlog invocation for the netlist file compilation. This keyword is ignored if the hot-swap feature has not been enabled.

Accessing Netlist Objects from RTL Hierarchy

If the netlist files have been compiled, you do not need to provide netlist object access using `NL_REF`. The compiler extracts the required information from the compiled netlist files for all hierarchical references into netlist objects from the RTL hierarchy.

© 2021 All Rights Reserved.

The IXCOM compiler recognizes the hierarchical references to objects in external netlist hierarchy as scalar nets. For such references, you do not need to provide any additional information in the configuration file.

For non-scalar nets, when netlist files have not been compiled, you must specify both hierarchical reference and object data-type information by using the `NL_REF` keyword:

- The `wire` keyword must be specified for hierarchical reference to *net* (or *signal*) in the netlist hierarchy.
- Packed dimension must be specified for hierarchical reference to `bit vector` net in the net-list hierarchy.
- Both packed and unpacked dimension must be specified for hierarchical reference to `memory` in the netlist hierarchy.
- Both the packed and unpacked dimension specifications must exactly match the original declaration for the netlist object.
- The `reg` keyword must be specified only in reference to the `memory` object as mentioned above. The hierarchical reference to `variable` in the netlist hierarchy is not supported.
- Support for hierarchical references is limited to `net` and `memory` objects. References to other objects are not supported.
- The IXCOM compiler issues an error message for hierarchical references that do not refer to netlist objects.
- The IXCOM compiler issues a warning if hierarchical references to a `net` object are detected in the design unit partitioned to the software side.

The information in this section is applicable only to the netlist that has not been compiled or analyzed by using either the `NL_COMPILE` configuration keyword or the `vlan -netlist` command. The NL_REF keyword is used to build a database of hierarchical references from the RTL and Testbench into those portions of the design about which IXCOM does not have any knowledge.

The following example shows the configuration specification for the non-scalar reference to external netlist objects:

```
// File: top.sv
   module Top;
      //...
      Dut dut(clk, din, dout); Chk chk(clk, dout);
         always @(*)
            dut.r = dut.chk.n[1];
   endmodule // Top
```


© 2021 All Rights Reserved.

```
// File: dut-nl.v
   module Dut(clk, din, dout)
      wire [3:0] a;
      reg [7:0] mem[1023:0];
   endmodule // Dut
// File: chk.sv
   module Chk(clk, dout);
      wire [3:0] n[32];
   endmodule // Chk
// File: nl-config.spec
   NL_FILE dut-nl.v
   NL_IMPORT Dut
   NL_EXPORT Chk
// Specify reference to external netlist objects
   NL_REF wire [3:0] Top.dut.a
   NL_REF reg [7:0] Top.dut.mem[1023:0];
   NL_REF wire [3:0] Top.dut.chk.n [32];
```

For designs compiled with the `NL_REF` keyword references, the references have the following functional behavior:

- The reference to the netlist is always functional (holds accurate value) when the design is compiled with the hot-swap feature enabled.
- The reference to the netlist is only functional in the hardware run when the hot-swap feature is not enabled.

Generating Clocks and Binding Clock signals in the Netlist Hierarchy

The RTL hierarchy (including clock generation module) can refer to clocks in the external netlist using hierarchical reference.

For the example shown in Importing Netlist Modules section, the `Top.dut.blk.clkSig` clock signal can be driven using clock module (generated manually or using the `ixclkgen` utility) as follows:

```
// File: clk.sv
   module Clk;
      //...
      // Generate logic for clock sources
      IXCclkgen  #(1) ixcg_0 (clk_0 );
      // Bind clock sources to generated clock signal
      ixc_cakebind ixcb_0 (Top.dut.blk.clkSig, clk_0);
      // ...
endmodule // Clk
```

You can also have hierarchical references to clock signals within the exported hierarchy. The clock module above must be compiled as an RTL file. You can use compile steps as mentioned in the Importing Netlist Modules and Exporting RTL Modules into the Netlist Hierarchy sections.

© 2021 All Rights Reserved.

To drive a clock signal from an external module into the top-level design unit imported as a netlist, you need to specify the `+targetTop+<du>` option for disconnecting port signals marked as input or inout ports.

Resolving Design Unit Redefinition

A design unit with the same name can exist in both netlist and RTL files. If netlist files are compiled using `NL_COMPILE` or `NL_REFLIB`, the IXCOM compiler uses netlist definition for the instances found in the netlist hiearchy and RTL defnitions are used for instances in the RTL hierarchy. To resolve design-unit re-definitions, you must analyze RTL and netlist files in a specific order to keep the required definition.

Note: The `vlan` tool keeps the last definition if mulitple definitions of the same cell or module are analyzed into the same logical library.

If netlist files are not compiled, you can keep definition from netlist or RTL using the `NL_DEF_KEEP` configuration specification.

The following sections provide details to resolve design unit redefinitions using `NL_DEF_KEEP`.

Keeping Netlist Definition

For design units defined in both domains, you can specify the following configuration to keep the definition from the netlist domain:

```
NL_DEF_KEEP external
```

This is the default behavior in the absence of the above configuration specification.

In some cases, the RTL definition of the module might have a different hierarchy than the netlist definition for the same cell or module. Considering the RTL definition, the IXCOM compiler might result in PI/PO signals to support features like SVA, Coverage, `$export_tasks` and so on. Keeping the netlist definition in such cases can lead to compile errors while mapping the design objects to the database. For such redefinitions, you must reorder the analyze invocations and overwrite the RTL definition with the netlist definition as follows:

```
vlan  -f <RTL_source_files>
vlan -netlist <Netlist_source_files>
```

Note: To implement this overriding of the order, both RTL and Netlist source files must be analyzed into the same logical library (default or user specified).

© 2021 All Rights Reserved.

Keeping RTL Definition

For the design unit defined in both netlist and RTL, you can specify the following configuration to keep the definition from the RTL domain:

```
NL_DEF_KEEP ixcom
```

Resolving Design Unit -Redefinition within Netlist and RTL

The configuration specifications mentioned in the earlier sections enable definitions to be chosen from netlist or RTL. Both the netlist and RTL files can have the design units re-defined as well.

For such a re-defined design unit:

- In both the IXCOM and Xcelium compilations, the last definition that is found is kept
- The back-end compiler (`xeCompile`) keeps the first non-empty definition

  You can modify this behavior to match the Xcelium compilation by using the following command:

  ```
  importOption {keep last}
  ```

Keeping Netlist Logical Libraries Separate in the Back-end Compiler

If netlist files are analyzed into separate logical libraries, such separation can be maintained in the back-end (xeCompile) compilation stage by using the following specification in the configuration file:

```
NL_IMPORT_LIB separate
```

Note: The above configuration is for legacy-ICE compatibility only and is not available for designs that require hot-swap. All of the legacy-ICE limitations apply for importing netlist files into separate logical library.

When importing netlist into separate logical libraries, all instances in the netlist must have a definition. This also means that direct instantiation of RTL within such netlists is not allowed. However, you can choose to insert such RTL instances into netlists by using the SystemVerilog bind construct as explained in the previous sections.

Generating 32-bit Simulation Snapshot for Xcelium Simulator

By default, all IXCOM, Xcelium, and Palladium tools, like xeCompile and xeDebug, are invoked in 64-bit, and all databases and files are generated in 64-bit. However, for certain designs, 32-bit compilation might be required. For example, for designs that have 32-bit

© 2021 All Rights Reserved.

SystemC shared libraries, which are needed during elaboration, 32-bit mode compilation would be necessary for the SystemC portion. Rest of the compilation can be done using 64-bit IXCOM and xeCompile tools. For this kind of compilation, use the `-ixcg32` option of the `ixcom` command.

1. Invoke IXCOM in 64-bit mode with `-ixcg32` option of the `ixcom` command:

```
ixcom -ixcg32 -ua +sv +dut+loopcnt tbench.v loopcnt.v +ignoreSimVerCheck
```

2. Invoke ***xrun*** in 32-bit mode with `-ixcg32` option of the `xrun` command:

```
xrun -32 -c -hw -ixcg32 -ua +sv +dut+loopcnt tbench.v loopcnt.v
     +ignoreSimVerCheck
```

If you want to compile the design only in 32-bit mode, invoke ***xrun*** and IXCOM in 32-bit mode by using the `-32` options of the `xrun` and `ixcom` commands, respectively.

1. Invoke IXCOM in 32-bit mode with `-32` option of the `ixcom` command:

```
ixcom -32 -ua +sv +dut+loopcnt tbench.v loopcnt.v +ignoreSimVerCheck
```

2. Invoke ***xrun*** in 32-bit mode with `-32` option of the `xrun` command:

```
ENV32=env LD_LIBRARY_PATH=$(<XE_INSTALL>)/tools.lnx86/lib/32bit
$(ENV32) xrun -32 -R -tcl
```

The 64-bit mode, which is default for Palladium Z1 is supported with all IXCOM and Xcelium tools in 64-bit, and all result databases and files are generated in 64-bit.

The following table summarizes the usage of these options and specifies the bit-mode in which each tool is invoked and the bit-mode of each generated database with respect to the `-32`, `-ixcg32`, and `-64` options.

Table 16-4 Bit-Mode Supported for Invoking Tools and Generating Databases

Option	IXCOM	xrun	xeCompile and xeDebug	Generated Xcelium Database (sss)	Generated Palladium Database (PDB)
`-ixcg32`	64-bit mode	32-bit mode	64-bit mode	32-bit mode	64-bit mode
`-64`	64-bit mode	64-bit mode	64-bit mode	64-bit mode	64-bit mode
`-32`	32-bit mode	32-bit mode	64-bit mode	32-bit mode	64-bit mode

© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

17

Using $ixc_ctrl Pragma

The `$ixc_ctrl` pragma should be put into an `initial` block. These are pragmas for corresponding code transformations during IXCOM compilation. In general, an `$ixc_ctrl` pragma falls into one of the following categories:

- IXC_CTRL Pragmas for Elaboration and Mapping
- IXC_CTRL Pragmas for Interface Compilation
- IXC_CTRL Pragmas for Behavioral Compilation

IXC_CTRL Pragmas for Elaboration and Mapping

- The following pragma makes the instances of the module unique during elaboration:

  ```
  $ixc_ctrl("unique_instance");
  ```

- The following pragma, for `'qtrefcell` module, dumps `var_decl` in the module, which has been compiled by HDL-ICE Compiler, and contains this directive:

  ```
  $ixc_ctrl("hdlice", var_decl);
  ```

- The following pragma forces the specified two-dimensional array variables to be bit-blasted into a register vector:

  ```
  $ixc_ctrl("map_to_reg", var[, ...]);
  ```

  You can also specify this pragma for OOMRs to variables in a hierarchy.

  Note: In place of `map_to_reg`, you can use the `no_hardmacro` memory-synthesis directive as follows:

  ```
  // quickturn no_hardmacro <local_var>
  ```

  Or

  ```
  -- quickturn no_hardmacro <local_var>
  ```

- The following pragma forces the specified two-dimensional array variables to be mapped as compact RTL memory:

  ```
  $ixc_ctrl("map_to_mem", local_var[, ...]);
  ```


© 2021 All Rights Reserved.

Note: In place of `map_to_mem`, you can use the `use_hardmacro` memory-synthesis directive as follows:

```
// quickturn use_hardmacro <local_var>
```

Or,

```
-- quickturn use_hardmacro <local_var>
```

If required, you can add the `map_to_reg` and `map_to_mem` pragma anywhere in the design and specify these pragma as instance-based XMRs. Specify the complete hierarchical path of the variables. When the module is split in case of multiple instances, the directive is applied on the variable of the particular instance specified in the XMR:

```
$ixc_ctrl("map_to_reg", <Complete_hierarchical_path_to_variable>);
$ixc_ctrl("map_to_mem", <Complete_hierarchical_path_to_variable>);
```

For example, consider the following module:

```
module dut(input clk, input [31:0] data);
    reg [31:0] mem [1023:0];
    ....
endmodule

module tb();
    ....
    dut d1(clk, wdata);
endmodule

module separateTop;                                // : Separately added top-
level module
       initial
       $ixc_ctrl("map_to_mem", tb.d1.mem);
endmodule
```

Note: Setting `map_to_reg` or `map_to_mem` pragma using XMR for VHDL memories is not supported.

- By default, packed arrays without unpacked dimensions are bit-blasted by IXCOM because of the limitation in the way Xcelium can get array values for hot-swap. A packed array can have many bits resulting in gate-count explosion. You can use the following compiler directive to enable support for large packed arrays mapped to emulator memories:

  ```
  // quickturn use_hardmacro  <array_name>
  ```

  The same syntax is used for arrays with unpacked dimensions. The above directive can also be applied to arrays without unpacked dimensions, such as the following:

  ```
  reg [4095:0] a1;            // quickturn use_hardmacro a1
  bit [127:0][31:0] a2;       // quickturn use_hardmacro a2
  ```

  These are packed arrays. This directive informs the HDL-ICE Compiler that the array can be mapped to an emulator memory instance.

  Packed arrays without the `quickturn use_hardmacro` directive need to be bit-blasted because IXCOM requires strict timing and out-of-module references. So, you

© 2021 All Rights Reserved.

need to identify this as a big array and specify the `use_hardmacro` directive if the array is intended to be mapped to a memory.

At run time, the data from Xcelium packed value format is automatically converted to emulator memory format for hot-swap.

- The following pragma is used to set `keep_net` synthesis directive to module local variables.

  ```
  $ixc_ctrl("keep_net", var[, ...]);
  ```

  Note that only module local vars are allowed and OOMRs to variables in a hierarchy are not supported.

  The `keep_net` pragma is primarily intended for internal use. Instead of `keep_net`, use the `keep_net` directive as follows:

  ```
  // quickturn keep_net <local_var>
  ```

  Or,

  ```
  -- quickturn keep_net <local_var>
  ```

IXC_CTRL Pragmas for Interface Compilation

- The following pragma turns on delay transformation for the module:

  ```
  $ixc_ctrl("map_delays");
  ```

  The above pragma is equivalent to specifying the `+iscDelay+<mod_name>` option with the `ixcom` command.

- The following pragma declares an import task or function to run in the blocking mode. By default, it uses memory to transfer input and output. With `tb_pio`, it uses primary I/O resources to do I/O transfer.

  ```
  $ixc_ctrl("tb_import", "<task_name>"[, "tb_pio"])
  ```

- The following pragma declares a task or function, called by the testbench to run in software.

  ```
  $ixc_ctrl("tb_task", "<task_name>")
  ```

- The following pragma declares an export task or function to run in the blocking mode. By default, it uses memory resource to transfer input and output. With `tb_pio`, it uses primary I/O resources to do I/O transfer.

  ```
  $ixc_ctrl("tb_export", <task_name>[, "tb_pio"])
  ```

- The following pragma marks the module in DUT as an XMR task and function inlining candidate. When applicable, tasks and functions in this module would be inlined to the module that is calling them. Refer to the VXE Command Reference Manual for the

© 2021 All Rights Reserved.

description of the `+xmrTfInlineOn` and `+xmrTfInlineOff` options to enable and disable this optimization.

```
$ixc_ctrl("xmrTfInlineOn");
```

IXC_CTRL Pragmas for Behavioral Compilation

- The following pragma turns on behavioral transformation for the module, and is equivalent to `+bc+<mod_name>`.

  ```
  $ixc_ctrl("map_bc");
  ```

- The following pragma unrolls the loops with static constant loop bound in the module when behavioral compilation is enabled.

  ```
  $ixc_ctrl("bc_unroll_loop");
  ```

- The following pragma marks the specified local vars to be behaviorally driven vars, to be used to override default behavior of behavioral compiler.

  ```
  $ixc_ctrl("beh_var", <local_var_reference>[, ...]);
  ```

- The following pragma marks the specified local variables to be pure RTL-driven variables, to be used to override default behavior of the behavioral compiler.

  ```
  $ixc_ctrl("rtl_var", <local_var_reference>[, ...]);
  ```

- The following pragma marks the specified local variables to be RTL clock; this enables behavioral compiler to apply certain performance optimization.

  ```
  $ixc_ctrl("rtl_clk", <local_var_reference>[, ...]);
  ```

- The following pragmas export named events when IXCOM compiles only the DUT, and cannot see the testbench:

  - One pragma is required for exporting the named-event variable for the testbench-sensitive event (`@dutinst.ev`):

    ```
    $ixc_ctrl("export_named_event_sens", <ev>);
    ```

  - One pragma is required for exporting the named-event variable for the testbench trigger (`->dutinst.ev`):

    ```
    $ixc_ctrl("export_named_event_trigger", <ev>);
    ```

  Note that these pragmas are unnecessary if the testbench is compiled together with the DUT with IXCOM.

- The following pragma forces `always` procedures that drive the specified local variables to be serialized.

  ```
  $ixc_ctrl("bcs", <local_var_reference>[, ...]);
  ```


© 2021 All Rights Reserved.

- The following pragma transforms a packed array to an unpacked array, and an unpacked array to a packed array.

  ```
  $ixc_ctrl("mdaReshape", <variableName>, <unpackedDimension>);
  ```

  Here, <*variableName*> is the name of the local variable and <*unpackedDimension*> is the final unpacked-array dimension number, which must be greater than or equal to zero. If this number is `0`, all the unpacked dimensions are moved to packed. If this number is `1`, and the current unpacked array dimension is `2`, one dimension is converted to packed, and so on.

  The `$ixc_ctrl mdaReshape` directive is module based and only applies to the array variables of synthesizable types:

  - Arrays of bit, reg, logic, packed structure or union
  - Byte, shortint, int, integer, longint, time

  Specify this directive in the same scope in which the variable definition exists. The conversion from packed to unpacked or unpacked to packed does not apply to the arrays whose types are defined using typedef(s).

  Example 1:

  ```
  bit [127:0][1023:0][4:0]  xm;
  initial
       $ixc_ctrl("mdaReshape", xm, 2);
  ```

  Output:

  ```
  bit [4:0] xm [127:0][1023:0];
  ```

  Example 2:

  ```
  int  xm [0:3];
  initial
       $ixc_ctrl("mdaReshape", xm, 0);
  ```

  Output:

  ```
  bit [0:3][31:0] xm;
  ```

  Example 3:

  ```
  int  xm [0:3];
  initial
       $ixc_ctrl("mdaReshape", xm, 2);
  ```

  Output:

  ```
  bit xm [0:3][31:0];
  ```

  You can also include this `$ixc_ctrl` directive in the side file that is specified by using the `-directiveFile` option of the `ixcom` command in the following format:

  ```
  M <library_name> <module_name> <variable_name> mdaReshape \
    <unpackedDim> [fatal]
  ```


© 2021 All Rights Reserved.

Here:

- <*library_name*> is the name of the library in which the module is compiled.
- <*module_name*> is the name of the module containing the variable to be reshaped.
- <*variable_name*> is the name of the variable to be reshaped.
- `mdaReshape` is the directive name.
- <*unpackedDim*> is the final unpacked dimension of the multidimensional array.
- [`fatal`] reports an error or warning if the variable, module, or library does not exist.

You need to manually handle the errors generated as a result of the reshaping of multi-dimensional array variables.

Note: Reshaped multi-dimensional array variables cannot be used in a port list.

Refer to the Support for HDL-ICE Compiler Pragmas in Side File section for details of the usage of the side file.

© 2021 All Rights Reserved.

18

Compiling Designs with SystemVerilog Constructs in SA Mode

This release natively supports mapping HDL written in IEEE 1800-2005 SystemVerilog to the hardware. The constructs of SystemVerilog are divided into the following categories for the purposes of compilation and hardware mapping:

- SystemVerilog Design Constructs
- SystemVerilog Testbench Constructs (SVTB)

The constructs that are not supported are listed in Support for SystemVerilog Constructs on page 959.

Important

The content of this chapter pertains only to SA mode.

SystemVerilog Design Constructs

IXCOM partitions the design into two partitions, one that can be run on the emulator, and the other that cannot be mapped to hardware. Any constructs that cannot be mapped to the emulator are partitioned on the software side with the testbench. During acceleration, the design modules in the testbench, execute on *xrun*, while the modules mapped to the emulator are run on emulator. For support of the SystemVerilog Design constructs by IXCOM, use the `+sv` option of the `ixcom` command. Alternatively, use the `vlan` executable to analyze the SystemVerilog design constructs, which can be, then, handled by IXCOM.

The SystemVerilog design needs to be compiled using the following steps.

Compiling a Pure SystemVerilog Design

Analyze and compile the SystemVerilog Design constructs using the following command:

```
ixcom +sv <module1> <module2>... +dut+<dut_module> -ua
```


© 2021 All Rights Reserved.

Alternatively, use the following command sequence to analyze and compile the design:

a. Use `vlan` to analyze the design containing SystemVerilog constructs:

```
vlan +sv <SystemVerilog_files> [vlan_options]
```

b. Compile the SystemVerilog Design constructs using the `ixcom` command:

```
ixcom -top <top_modulename> +dut+<dut_module> -ua
```

Compiling a Mixed-Language Design

1. Compile the design written in the various languages, such as VHDL, SystemVerilog, and Verilog using the following language compilers:

   a. For VHDL designs, use the following `vhan` command:

   ```
   vhan <VHDL_primary_or_secondary_unit> [vhan_options]
   ```

   b. For SystemVerilog designs, use the following `vlan` command:

   ```
   vlan +sv <SystemVerilog_files> [vlan_options]
   ```

2. Compile the SystemVerilog design constructs using the `ixcom` command:

   ```
   ixcom -top <top_module> +dut+<dut_module> -ua
   ```

SystemVerilog Testbench Constructs (SVTB)

Certain SystemVerilog constructs, which are typically a part of the testbench, need to be separately linked by using ***xrun***, and these cannot be directly handled by IXCOM. If these constructs are passed directly to the IXCOM compiler, it generates an error message to indicate that these constructs are unsupported. Most of these constructs belong to the testbench, such as SystemVerilog Classes, randomization and so on. The SVTB constructs are specified in Appendix 19, “Support for SystemVerilog Constructs”.

Compile the testbench modules that contain these SVTB constructs with ***xrun***, and compile the DUT (that is, the RTL) with the `+xcDesignTop` option of the `ixcom` command. For more information on the `+xcDesignTop` option, refer to Linking DUT with Multiple Testbenches on page 649.

Compiling SVTB Constructs Supported by IXCOM

The SVTB constructs need to be compiled using the following process:

© 2021 All Rights Reserved.

Compiling a Pure SystemVerilog Design with SVTB Constructs

Analyze and compile the SystemVerilog SVTB constructs using the following command:

```
ixcom +sv <module1> <module2> +dut+<dut_module> \
   +xcDesignTop+<hier_ref_to_DUT_inst_in_testbench>=<DUT_module_name> \
   -ua
```

Alternatively, use the following command sequence to analyze and compile the design:

a. Use `vlan` to analyze the design containing SystemVerilog constructs:

```
vlan +sv <SystemVerilog_files> [vlan_options]
```

b. Compile the SystemVerilog Design constructs using the `ixcom` command:

```
ixcom -top <top_modulename> +dut+<dut_module>\
      +xcDesignTop+<hier_ref_to_DUT_inst_in_testbench>=<DUT_module>\
      -ua
```

Compiling a Mixed-Language Design with SVTB Constructs Using the `+xcDesignTop` Option

1. For mixed-language designs, compile the design written in the various languages, such as VHDL, SystemVerilog, and Verilog by using the language compiler for each language

   a. Compile VHDL primary and secondary units using the following command:

   ```
   vhan <vhdl_primary_or_secondary_unit> [vhan_options]
   ```

   b. Compile SystemVerilog and Verilog modules using the following command:

   ```
   vlan +sv <SystemVerilog_files> [vlan_options]
   ```

2. Compile using IXCOM to generate the compilation databases.

```
ixcom -top <top_level_module> -top <another_top_level_module> \
      +xcDesignTop+<hier_ref_to_DUT_inst_in_tb>=<DUT_mod_name> ...
```

Support for Covergroups and Mailbox Constructs

IXCOM supports the following SystemVerilog testbench constructs:

- Support for Covergroups
- Support for Mailbox

Support for Covergroups

Covergroups are supported in the testbench and DUT. Refer to Coverage Support on page 997 for more information.

© 2021 All Rights Reserved.

Support for Mailbox

Mailbox is implemented as a template class in SystemVerilog package implementation. The standard package is imported into a design when the `+sv` option is used.

Following datatypes are supported for mailbox:

- `shortint`, `int`, `longint`, `byte`, `integer`, `bit`, `logic`, `reg`
- class handles
- packed arrays of types `bit`, `logic`, and `reg`
- `typedefs` of the above datatypes

Compile the modules that contain these SVTB constructs by compiling DUT and testbench separately with the SVTB compilation method. For more information, refer to Linking DUT with Multiple Testbenches on page 649.

For a list of supported SVTB constructs, refer to Appendix 19, “Support for SystemVerilog Constructs”.

Support for SystemVerilog Interface Constructs

IXCOM supports SystemVerilog interfaces as SVTB constructs (in the software partition) and design constructs (in the hardware partition or the DUT). For a list of supported SystemVerilog interface constructs, refer to Appendix 19, “Support for SystemVerilog Constructs”.

When a SystemVerilog interface is used as a design construct, reference to an interface object through a port connection is limited to:

- net or variable of integral datatype or packed one-dimensional array data type
- task or function
- interface instance

When the SystemVerilog interfaces are used as design constructs, unsupported references to interface objects or port connections are checked during RTL synthesis (controlled by the `-rtlchk` option). For more information on the design partitioning using RTL synthesis checks, refer to Compiling for Hardware Using IXCOM on page 634.

© 2021 All Rights Reserved.

Caution

The current implementation does not warn or issue error for the following interface object references, and can cause simulation mismatch in some cases:

- Hierarchical reference to interface objects
- Multiple writes to an interface object from different modules

For steps to compile SystemVerilog interfaces as SVTB constructs, refer to Compiling SVTB Constructs Supported by IXCOM. For steps to compile SystemVerilog interfaces as design constructs, refer to SystemVerilog Design Constructs.

Interface Declaration

IXCOM supports the interface declaration in both the hardware and software partitions. For items or contents that are common to the module and interface declaration, IXCOM supports these in the interface declaration the same as that in the module declaration (in both the hardware and software partitions).

The following sections describe the support for items and contents that are unique to the interface declaration.

Support for Modports

IXCOM supports the modport declaration in both the hardware and software partitions, with the following limitations:

- Modport expressions are not supported.
- Use of modports to specify the direction of the clocking blocks declared within an interface is not supported.
- Use of modports to import and export tasks and functions is not supported.
- Hierarchical access to modport ports is not supported.

During elaboration, the IXCOM compiler retains the modport names and modports written under interfaces. Therefore, the Palladium generated SHM files also have the modport names. When Indago loads the LWD database and SHM waveform files, you can perform all modport related debug operations, such as sending modports from the Design browser to waveform display, driver tracing, and so on.

© 2021 All Rights Reserved.

Support for Virtual Interfaces

The virtual interface construct is supported only in the software partition. It is not supported in the hardware partition. When using virtual interface constructs, ensure that the modules containing the virtual interface construct are separately compiled for the software partition, and linked with the DUT, which is compiled for the hardware partition. For this separate compilation and linking and compiling multiple top-level modules, use the `+xcDesignTop` or `+xcEmbeddedTb` options of the `ixcom` command as required. Refer to Compiling Testbenches Separately section for details about this use model.

Compiling Virtual Interfaces

When the `+xcDesignTop` option is used, the DUT is compiled by IXCOM and the testbench is compiled with Xcelium. The interface gets transformed by IXCOM in order to insert required instrumentation so that it can be executed on hardware. This creates problem when we assign a virtual interface on a TB side to an instance of interface on the DUT side.

IXCOM compiler transforms interface and generates the interface definitions with the transformed name in the `xc_work/v/` directory based on the following reasons:

- Parameterization
- Enabling or disabling of features like coverage/assertions in particular instance
- Instance hierarchy
- Port specifications

To associate the virtual interface variable written in the testbench to its appropriate definition automatically, specify the `+genIfcSideFile` option. This option directs IXCOM to process the interfaces instantiated in the DUT and generate a mapping file. This mapping file contains the information that helps in binding the virtual interface with the right instance of the interface, which was split into separate definitions. The mapping file is dumped in the `AxisWork/` directory with the name `ixc_ies_ifc`.

Important

If you want to run the design in pure Xcelium environment in the same directory in which the IXCOM work directory exists, it is recommended that you delete the following file before running the design in pure Xcelium environment:

```
AxisWork/ixc_ies_ifc
```

The example of compiling design with virtual interface is shown below. The example illustrates both the `+xcDesignTop` option as well as multiple-top-level-modules methodology.

© 2021 All Rights Reserved.

Example of Compiling Parameterized Interfaces with the +xcDesignTop and +genIfcSidefile Options

Consider the example in which the DUT is written in `dut.v`, the interface is defined in `ifc.v` whereas the testbench is in `tb.v`.

To compile the design with `+xcDesigntop` option, execute the following command with the necessary options:

```
ixcom -clean dut.v ifc.v +dut+dut -ua +xcDesignTop+tb.DUT=dut \
      +genIfcSideFile +sv
```

To compile the design with two top-level modules, execute the following command with the necessary options:

```
ixcom -clean dut.v ifc.v +dut+dut -ua +genIfcSideFile +sv
```

If the `ifc` instance is present in the testbench along with the instance(s) on DUT side, pass the `ifc.sv` file to the `xrun -elaborate` step as well, as shown below:

```
xrun -elaborate -f xc_work/xrun.f -top tb tb.v -sv ifc.v
```

If the `ifc` instance is not present in testbench:

```
xrun -elaborate -f xc_work/xrun.f -top tb tb.v -sv
```

Limitations in the Support for Virtual Interfaces

Following are the limitations in the support for virtual interfaces:

- The `scope -describe | -list` command shows the transformed interface names instead of the original interface name `ifc` for virtual interface variable.
- Using `defparam` from testbenches in the instances of interfaces declared in DUT are not supported.
- Scenarios related to SystemVerilog-interface usage inside the DUT that IXCOM does not handle are not supported.
- XMRs that either resolve to a virtual interface declaration or have a path that resolves through a virtual interface are not supported in IXCOM.
- IXCOM does not preserve the lexical order between a virtual interface declaration and an interface instantiation. IXCOM always dumps (in the output that it produces for consumption by the Xcelium or Palladium compiler) all variable declarations before all instance instantiations (in a given scope). Therefore, a scenario where there is a virtual interface declaration (which is a variable declaration) with an interface instantiation as the initial value, will lead to semantically incorrect HDL (due to incorrect lexical order) output by IXCOM.

© 2021 All Rights Reserved.

- For all `RTL_UNSUPPORTED_*` messages that are issued for synthesis violations found within an interface, the interface declaration is inaccurately referenced as a module.
- Declaration of virtual interface with initialization expression is not supported.
- If there is any instance of interface in the testbench, the interface definition is required to be passed to the `xrun -elaborate` step. At present, ***xrun*** silently binds the interface instance on the testbench side with the dumped interface. This might lead to elaboration errors related to OOMR resolution or simulation mismatch at run time. ***xrun*** does not generate error messages for such specific situations to indicate that an interface definition to the `xrun -elab` step is required.
- For interface instance arrays or Forgen statements, use the `-rtlNameForGenerate` option. Else, instance arrays of interface cannot be accessed using bracketed notation.
- When the `+genIfcSideFile` option is used, IXCOM ensures that all the interface instances are considered unique. The following scenario is not supported with the `+genIfcSideFile` option or with any case in which the interface was considered unique.

Compile Script:

```
ixcom +sv  +xcDesignTop+TOP.DUT_TOP=dut_top -ua +dut+dut_top +genIfcSideFile
dut_top.sv  ifc.sv dut.sv  -clean
```

```
xrun -elaborate -f xc_work/xrun.f -sv -top TOP  -timescale 1ns/1ns top.sv
tb_top.sv
```

Testbench:

```
//Verification TOP, top.sv

module TOP;
       logic clk=0;
       always #1 clk =~clk;
       dut_top DUT_TOP(clk); //xcDesignTop

// Testbench top-level module with interface port results in error when \
   compiled with the commands mentioned in the compile script.

// An error CUINDN is generated to indicate that the interface port \
   should be connected to an interface of the same name.

tb_top TB_TOP(DUT_TOP.i1);
endmodule

//Test bench top, tb_top.sv
module tb_top(ifc.tb i2);
       virtual ifc #(1) vif;
       initial vif=TOP.DUT_TOP.i1;
endmodule

//DUT top , dut_top.sv

module dut_top(input clk);
     ifc #(1) i1(clk);
     dut #(2'b00,2'b01,2'b10,2'b11) d1(i1);
endmodule

//dut, dut.sv
```


© 2021 All Rights Reserved.

```
parameter type t1 = reg[1:0];
module dut #(t1 S0=2'b00,t1 S1=2'b01,t1 S2=2'b10, t1 S3=2'b11) (ifc.dut i1);
     reg[1:0] state, next_state;
     always@(posedge i1.clk, negedge i1.rst)
     begin
       if(~i1.rst) state <= S0;
       else state <= next_state;
     end
     always@(state,i1.x_in)
     begin
       case(state)
            S0,S1,S2: i1.y_out = 0;
            S3: if(i1.x_in) i1.y_out = 1; else i1.y_out = 0;
       endcase
     end
endmodule

//Interface between tb_top and dut_top , ifc.sv
interface ifc #(parameter p1=2) (input clk);
     logic y_out;
     logic x_in,rst;
     modport dut(output y_out, input x_in,clk,rst);
     modport tb(input y_out, output x_in,clk,rst);
endinterface
```

Interface Ports

Interface port declarations are supported in both the hardware and software partitions. References to both Interface instances (both local and hierarchical) and other interface ports (both local and hierarchical) as actuals of an interface port are allowed.

Modport specification is supported in the interface port declaration as well as interface port actuals.

Accessing Objects Through Interface Ports in the Hardware Partition

Interface contents are allowed in the hardware partition. However, only the following can be accessed through the interface ports in the hardware partition:

- Nets and variables of the following datatypes:
  - reg, logic, bit.
  - 1-dimensional packed arrays of {reg, logic, bit}.
  - byte, shortint, int, integer longint.
  - Multi-dimensional packed arrays.
  - enum datatype whose base type is a supported datatype.
  - packed struct, all of whose members have supported datatypes.

© 2021 All Rights Reserved.

  - packed union, all of whose members have supported datatypes.
- Memories defined as 1-dimensional unpacked arrays of 1-dimensional packed arrays.
- Built-in method calls on enum variables referred through interface ports are not supported in the hardware partition. Built-in method calls (first, last, next, prev, num) on enum variables are otherwise supported in the hardware partition.

XMR References to Interface objects

In addition to accessing (interface) objects through interface ports, (interface) objects can also be accessed using a traditional XMR. XMR access to interface objects is supported in the same manner as XMR access to module objects with the following caveats:

- XMR write access to module objects is not supported in the hardware partition. This restriction does not apply to XMR write accesses to interface objects.
- Access to (interface) objects through XMR references to interface ports is not supported.

The following example illustrates the types of XMR access that are permitted.

```
// ixcom -clean -sv ex1.v
// ex1.v

interface ifc;

  reg [3:0] counter;
endinterface

module dut;
  reg [3:0] counter;
  ifc I();
  sub S(I);
  initial counter = I.counter;
  // regular XMR to interface object, supported
  always I.counter = 0;
  // regular XMR write to interface object, will skip XMRWRITE error
endmodule

module sub(ifc ip);
  reg [3:0] counter;
  initial counter = ip.counter;
  // access to object via interface port, supported
  always ip.counter = 0;
  // write access to object via interface port, will skip XMRWRITE error
  foo F();
endmodule

module foo;
  reg [3:0] counter1, counter2;
  initial counter1 = dut.S.ip.counter;
  // access to object via XMR to interface port, not supported
  always dut.S.ip.counter = 0;
  // write access to object via XMR to interface port, not supported
  initial counter2 = dut.I.counter;
  // regular XMR to interface object, supported
  always dut.I.counter = 0;
```


© 2021 All Rights Reserved.

```
  // regular XMR write to interface object, will skip XMRWRITE error
endmodule
```

Limitations In Interface Support

All aspects of the SystemVerilog interface construct that are defined in the LRM are not supported. The limitations in the support for the modport construct, virtual interface construct, interface port, XMR access to interface are listed in the respective sections. This section lists all the other limitations in the support for the SystemVerilog interface constructs.

In general, for constructs common to both the module declaration and the interface declaration, the following rules apply:

- The constructs that are supported for modules are also supported for interfaces.

  For example, `generate` blocks are common to modules and interfaces, and supported in both cases.

- Limitations in the support of a construct for modules also apply for interfaces.

  For example, the class declaration is neither supported for modules nor for interfaces.

- The top-level DUT module - the module specified with the `+dut+<module_name>` option of the `ixcom` command should not have an interface port.

- The interface-port-specific enhancement for `specify` blocks (Section 25.6 of the LRM) is not supported.

- The support for access to interface objects through `$nc_mirror` is the same as the support for access to module objects through `$nc_mirror`, with one exception.

  The path to the interface object (in the argument to `$nc_mirror`) cannot have a reference to an interface port within it.

- Refer to Limitations in the Support for Virtual Interfaces section.

Support for Interface Port Arrays

Interface port array constructs are supported for:

- Declaration of interface port arrays (one dimensional only).

- References (to interface contents) by selecting elements of an interface port array.

- Use of the ranged instantiation identifier (local and hierarchical, slice select and element select), as an actual in a component instantiation, where the formal is a compatible interface port (array).

© 2021 All Rights Reserved.

- Use of the interface port array identifier (local and hierarchical, slice select and element select) as an actual in a component instantiation, where the formal is an interface port (array).

The following examples illustrate the features mentioned above.

```
// ixcom -clean -sv ex2.v
// ex2.v

module test;
  ifc I1 [3:0] ();
  sub1 S11(I1);
  sub2 S12(I1[3:2]);
  sub3 S13(I1[1]);
endmodule
interface ifc;
endinterface

module sub1(ifc ip[3:0]);
endmodule

module sub2(ifc ip[1:0]);
endmodule

module sub3(ifc ip);
endmodule

// ixcom -clean -sv ex3.v
// ex3.v

module test;
  ifc I1 [3:0] ();
  hier H();
endmodule

module hier;
  sub1 S11(test.I1);
  sub2 S12(test.I1[3:2]);
  sub3 S13(test.I1[1]);
endmodule

interface ifc;
endinterface
module sub1(ifc ip[3:0]);
endmodule

module sub2(ifc ip[1:0]);
endmodule

module sub3(ifc ip);
endmodule

// ixcom -clean -sv ex4.v
// ex4.v

module test;
  ifc I1 [3:0] ();
  sub S(I1);
endmodule

module sub (ifc ip[7:4]);
  sub1 S11(ip);
  sub2 S12(ip[7:6]);
  sub3 S13(ip[5]);
endmodule
```


© 2021 All Rights Reserved.

```
interface ifc;
  reg v;
endinterface

module sub1(ifc ip[3:0]);
endmodule

module sub2(ifc ip[1:0]);
endmodule

module sub3(ifc ip);
endmodule

// ixcom -clean -sv ex5.v
// ex5.v

module test;
  ifc I1 [3:0] ();
  sub S(I1);
  sub1 S11(S.ip);
  sub2 S12(S.ip[7:6]);
  sub3 S13(S.ip[5]);
endmodule

module sub (ifc ip[7:4]);
endmodule

interface ifc;
endinterface

module sub1(ifc ip[3:0]);
endmodule

module sub2(ifc ip[1:0]);
endmodule

module sub3(ifc ip);
endmodule

// ixcom -clean -sv ex6.v
// ex6.v

module test;
  ifc I1 [2:0] ();
  sub1 S1(I1);
endmodule

interface ifc;
  reg [1:0] v;
  initial
    #1 $display("%m: v = %h", v);
endinterface

module sub1(ifc ip[2:0]);
  initial ip[2].v = 2;
  sub2 S(ip[1:0]);
endmodule

module sub2(ifc ip[1:0]);
  initial ip[1].v = 1;
  sub3 S(ip[0]);
endmodule

module sub3(ifc ip);
  initial ip.v = 0;
endmodule
```


© 2021 All Rights Reserved.

Limitations in SystemVerilog Interface Array Support

- Only one dimensional interface port arrays are supported.
- For ranged interface instances that are used as actuals (or connected to) a given interface port array, all of those instances should be of the same unique module after elaboration. Else, the IXCOM compiler issues an error. Note that this restriction is not imposed by ***xrun***.

```
// ixcom -clean -sv ex7.v
// ex7.v

module test;
  ifc I1 [1:0][1:0] ();
  sub S1(I1);
endmodule

interface ifc;
endinterface

module sub(ifc ip[1:0][1:0]);
endmodule
```

- Virtual interface arrays declaration is not supported. References to objects through virtual interface are also not supported.

```
// ixcom -clean -sv ex8.v
// ex8.v

module test;
  virtual interface ifc vi[1:0];
  ifc I[1:0]();
  initial
  begin
       vi[0] = I[0];
       vi[1] = I[1];
  end
  sub S(I);
endmodule
interface ifc;
   int counter;
endinterface
module sub(ifc ip[1:0]);
endmodule
```

Support for SystemVerilog Packages in SA Mode

SystemVerilog packages are declared using the `package` keyword. An example of a SystemVerilog `package` declaration is given below.

```
package myPkg;
    typedef int myType;
    myType myVar;
endpackage
```

Objects declared within a SystemVerilog package can be referenced using one of the following ways:

© 2021 All Rights Reserved.

- Explicitly by using the `<pkg_name>::<object_name>` syntax.
- Implicitly by using the `<object_name>` variable, after the `<object_name>` variable is made visible in a given scope through a package `import` declaration.

  ```
  import <pkg_name>::<object_name>;
  ```

  Or,

  ```
  import <pkg_name>::*;
  ```

Refer to the SystemVerilog LRM for details about these declaration methods.

The SystemVerilog package name has a global namespace, which is different from the namespace for the name of a module, interface, or program declaration. This means that two SystemVerilog packages should not have the same name, but a SystemVerilog package can have the same name as a module (or interface or program) declaration.

IXCOM supports use of packages with the same name when these are defined in different logical libraries as explained in the Support for Packages with Same Names in Multiple Logical Libraries section.

Note: As per the LRM, the following references are not allowed for SystemVerilog packages:

- Forward references to SystemVerilog package objects (that is, referring to the package before the SystemVerilog package declaration)
- References to Compilation Unit Scope objects from within SystemVerilog packages.

The IXCOM compiler supports the SystemVerilog package declaration as well as all the forms of references to SystemVerilog package objects. It also supports package import statements written as a part of the module header as per Section 23.2 of the 1800-2009 SystemVerilog LRM.

Use the `+showSvPkgAccess` option of the `ixcom` (when using single-step compilation) or `vlan` (when using multi-step compilation) command to enable display of compilation-time information messages to show the package objects being accessed.

The following INFO message is displayed for every access to SystemVerilog package object:

```
Info in file test.sv at line 9           [FOUND_SV_PKG_ACCESS]
Access to a SystemVerilog package object found
(pkg: pkg1, obj: pkg1Type, file: test.sv, line: 5)
```

The IXCOM compiler supports references to a SystemVerilog package that was compiled in a previous analyzer session (each `vlan` command is a separate analyzer session). If the SystemVerilog package is compiled in the same analyzer session, then, the definition of the package should appear before the reference. Refer to the Support for Packages Referenced Across Multiple Logical Libraries section.

© 2021 All Rights Reserved.

Important

A bidirectional signal declared in a SystemVerilog package having both testbench and DUT drivers is not supported.

Refer to the Support for Packages with Same Names in Multiple Logical Libraries section for details about searching packages with same names in different libraries by using the `-pkgSearch` and `-enablePkgSearch` options.

Accessing SystemVerilog Package Objects from Library Modules

Access to SystemVerilog packages declared in the source HDL is allowed from modules compiled from library files (using the `-v` option of the `vlan` or `ixcom` command). However, the SystemVerilog package declarations contained within library files are ignored, and, therefore, these cannot be accessed from modules compiled from library files.

Consider the following example:

```
// file1.sv
package p1;
     typedef int myInt1;
endpackage
// Note: Compiling the file2.sv as a -v library file as shown below in the ixcom
// command causes a problem.
// file2.sv
package p2;
     typedef int myInt2;
endpackage
module top;
      dut DUT();
endmodule
module dut;
      p1::myInt1 v1;        // OK
      p2::myInt2 v2;        // Not OK : reference to myInt2
endmodule
```

The above example can be compiled as follows:

```
ixcom -clean file1.sv -v file2.sv
```

In the above example, the module `dut` is picked from the library file `file2.sv`. The package `p2` is declared in a library file and, hence, it is not seen by the IXCOM compiler. Subsequently, any reference to package `p2` will result in an error. References to the packages declared in source files (that is, package `p1`) are allowed from modules declared in library files (that is, module `dut`).

© 2021 All Rights Reserved.

Support for Packages Referenced Across Multiple Logical Libraries

The IXCOM compiler supports module references to packages that were analyzed in a logical library different from the one in which the modules were analyzed.

Consider the following example.

```
//pkg.v
package pkg
    typedef int MYINT;
endpackage
//top.v
module top;
    reg clk,a;
    dut dutInst(clk,a);
endmodule
//dut.v
module dut (input reg clk, a);
    import pkg::MYINT;
    MYINT b;
endmodule
```

The above package and module can be analyzed and compiled into multiple logical libraries as follows:

```
vlan +sv -work L1 pkg.v top.v
vlan +sv -work L2 dut.v
```

To resolve the package reference, the `vlan` executable first searches the current logical library for the package referenced in the module. If it does not find the package, it searches other logical libraries specified in the libmaprc file(s).

If multiple packages are found, an error message is issued because the LRM states that the packages must have unique names.

If the referenced package cannot be found, an error message is issued because according to the LRM a package needs to be compiled before being referenced.

Support for Package References Across Libraries

IXCOM does not support referencing of a package defined in a library from a design unit defined in another library, by default. You can use the `-enableCrossLibPkgReference` option of the `ixcom` command to enable support for cross-library package references.

Consider the following example:

```
vlan -sv source/l1/pkg.v -WORK LIB1 # package pkg
vlan -sv source/l2/pkg1.v -WORK LIB2 #package pkg1
vlan -sv m2.v -WORK LIB3 # module m2 reference pkg.v and pkg1.v
vlan -sv m1.v -WORK LIB2 # module m1 reference pkg.v and pkg1.v
```


© 2021 All Rights Reserved.

```
vlan -sv test.v -WORK LIB1
ixcom -64 -top test -ua +dut+test -enableCrossLibPkgReference
```

When you specify the `-enableCrossLibPkgReference` option, according to the package reference across library, IXCOM determines the package dependencies between sessions.

The session in which the package is compiled must be analyzed before it is accessed by another session. The sequence of sessions is not important except that packages are compiled first to maintain the dependencies between sessions. Then, other design units are compiled.

IXCOM specifies the `-reflib` option for HDL-ICE Compiler in the correct order for each `vavlog` command for cross-library package reference.

Note: All files must be Verilog or SystemVerilog. Cross-library package references are not supported in mixed-language designs.

Handling of Re-declaration of Packages

You cannot analyze two *different* packages with the same name into the same logical library. The *same* package can, however, be re-analyzed. Two packages of the same name, or the same package) can be analyzed into two different logical libraries. However, the IXCOM compiler generates an error if the design is found to have references to two *different* packages with the same name. Note that the IXCOM compiler allows multiple definitions of the *same* package.

For compiling packages with the same name in multiple logical libraries use the `-enablePkgSearch` option. To enable cross-library access of such packages, use the `-pkgSearch` option and specify the library.

Refer to Definition of Same Package as per the vlan Analyzer and Definition of Same Package as per the IXCOM Compiler for the definition and handling of *same* package.

If your compilation script results in the *same* package being specified in multiple `vlan` sessions, all subsequent compilations of the *same* package (in the same logical library) are ignored, and `vlan` re-uses only the original definition.

© 2021 All Rights Reserved.

Definition of Same Package as per the vlan Analyzer

During analysis, `vlan` uses the following algorithm to identify if the package being analyzed is the same as a previously analyzed package in the same logical library:

Package Definitions Being Compared	Same or Different
If the HDL filename containing the previously analyzed package definition is ***different*** from The HDL filename containing the package definition being currently analyzed	Packages are considered to be ***different*** and an error message is issued.
If the location of the HDL filename containing the previously analyzed package definition is ***different*** from The location of the HDL filename containing the package definition being currently analyzed Note that `vlan` traverses symbolic links to identify if it was the same file.	Packages are considered to be ***different*** and an error message is issued.
If the last-modified time of the HDL file (containing the package definition being currently analyzed) is more recent than The created time of the database file (with the previous definition of the package in the same logical library)	The previous definition is considered ***outdated*** and `vlan` issues an error message.
If the last-modified time of the HDL file (containing the package definition being currently analyzed) Is older than The created time of the database file (with the previous definition of the package in the same logical library), then the two package definitions are now considered *same*.	Packages are considered to be ***same***.

© 2021 All Rights Reserved.

Definition of Same Package as per the IXCOM Compiler

During compilation, the IXCOM compiler uses the following algorithm to identify if the two analyzed definitions of a package with the same name in different logical libraries are for the *same* package:

Package Definitions Being Compared	Same or Different
If the HDL filename stored in the two package analyzed databases is ***different***	Packages are considered to be ***Different*** and an error message is issued.
If the location of the HDL filename stored in the two package analyzed databases is ***different*** Note that IXCOM compiler traverses symbolic links to identify if it was the same file.	Packages are considered to be ***Different*** and an error message is issued.
All other packages are considered to be ***same***.	

Support for Packages with Same Names in Multiple Logical Libraries

Many designs have packages that have the same name, but are compiled in different libraries. This introduces some ambiguity for IXCOM while picking up the correct package definition for such package references in the design.

For the cases when packages with same names are compiled in different libraries, IXCOM facilitates specification of the library with which such packages can be picked up correctly with the `-pkgSearch` and `-enablePkgSearch` options.

Note: This facility is similar to the Xcelium `-pkgSearch` option with the `xmvlog` command to enable specification of the correct package reference so that the intended package is picked up. Also, while generating `xmvlog` sessions in `xc_make`, IXCOM considers these library dependencies for identifying the dependent package library as defined with the `-pkgSearch` option.

Use the following options to enable search of packages definition from the specified library while looking up the package references in a SystemVerilog design:

- `-pkgSearch` <*`library_name`*> option of the `vlan` command: This option ensures that while parsing package references, the package definitions are looked up only in the library specified as <*`library_name`*>. If multiple `-pkgSearch` options are provided on the command line, the package definitions are looked up in the same order as these

© 2021 All Rights Reserved.

are specified on the command line. If this option is not specified, the default behavior is applied.

Default behavior means that the first current library is looked up and if the package reference is not found, rest of the libraries are looked up in the order as these appear in the `libmaprc` file. If multiple definitions are found, an error is reported.

- `-enablePkgSearch` option of the `vhelab` or `ixcom` command: This option ensures that IXCOM allows packages of the same name and generates the `xmvlog` commands in the `xc_make` file with appropriate `-pkgSearch` options along with the dependent package libraries.

The `-pkgSearch` option is needed for cross-library package references, to ensure intended package access by specifying the corresponding library. When all the packages that are referred in a particular `vlan` session are available in the current logical library, the `-enablePkgSearch` option allows usage of packages with the same name in multiple logical libraries.

Examples of Supporting Package Searches in Specified Libraries

Following are some examples of using the `-pkgSearch` <*library_name*> and `-enablePkgSearch` options:

Example 1:

```
// pkg1.sv
   package pkg;
      typedef logic mytype;
   endpackage
// pkg2.sv
   package pkg;
      typedef logic[1:0] mytype;
   endpackage
// top.sv
   module top;
      pkg::mytype X;
      assign X = '1;
   endmodule
```

Here, `top` refers to the `pkg` package, which is compiled in the `lib1` and `lib2` libraries.

To use the package from `lib1`, use the following commands:

```
vlan -sv pkg1.sv -work lib1
vlan -sv pkg2.sv -work lib2
vlan -sv top.sv -work work -pkgSearch lib1
ixcom -top top -enablePkgSearch
```

Example 2:

© 2021 All Rights Reserved.

```
//mid.sv
module mid;
   pkg::mytype X;
endmodule

//top.sv
module top;
   pkg::mytype X;
   assign X = `1;
   mid mid();
endmodule
```

To allow packages with the same name in the `lib1` and `lib2` libraries, use the following commands:

```
vlan -sv pkg2.sv mid.sv -work lib2
vlan -sv pkg1.sv top.sv -work lib1
ixcom -top top -enablePkgSearch
```

Considerations for Searching Packages with the Same Name in Specific Libraries

Avoid the following scenarios while specifying packages with the same name to be searched in specific libraries because the results might be unexpected:

- Avoid packages with the same name that have functions, variables, and are hierarchically referenced anywhere.
- When package search order among the multiple sessions of the same logical library is important, `vavlog` might fail. Multiple `vlan` sessions of the same logical library are clubbed into a single session, and, therefore, the ordering of the search library using the `-reflib` option cannot be done accurately.
- A bind construct having a package reference is compiled in one library, and the target module is compiled in a different library. If the packages with the same name as specified in the corresponding reference of the bind construct exist in both the libraries, there can be unexpected results.

  Consider the following example:

  ```
  File1.sv:

     package pkg;
        localparam l = 1;
     endpackage
     module top;
        bind mid leaf #( pkg :: l ) I();
        mid I();
     endmodule

  File2.sv

     package pkg;
        localparam l = 2;
     endpackage
     module mid # ( parameter p = 0 );
     endmodule
  ```


© 2021 All Rights Reserved.

```
module leaf # ( parameter p = 0 );
   initial
      $display("leaf::p = %d", p);
endmodule
```

The following commands are used to compile and run the above example:

```
vlan -sv top.sv -work lib1
vlan -sv mid.sv -work lib2
ixcom -top top -enablePkgSearch
```

`'leaf'` will have a value from `pkg` of `lib1`, that is `1`. But, ideally, it should have the value from `lib2`.

- If the datatype has a package reference, like `#(pkg::dt)`, and the target module is compiled in different libraries that have packages with the same name as in the datatype reference, there can be unexpected results, and a warning is generated at the time of elaboration.

List of Supported SystemVerilog package Constructs

The following table lists the SystemVerilog package constructs supported by the Xcelium compiler, the Palladium HDL-ICE Compiler, and the IXCOM compiler according to the following legend:

- **Yes**: Yes, this feature is supported.
- **No**: No, this feature is not supported.

Note: The constructs that are not supported by the IXCOM compiler, but are supported by the Xcelium compiler, can be separately compiled as a part of the testbench by using the Xcelium compiler. For this, use the SVTB compilation method to compile the DUT with the IXCOM compiler and testbench with the Xcelium compiler. Refer to SystemVerilog Testbench Constructs (SVTB) on page 932 for details about separate compilation of testbench.

Table 18-1 Support for SystemVerilog Package Constructs by Various Compilers

Package Constructs	Xcelium Compiler	Palladium HDL-ICE Compiler	IXCOM Compiler
net_declaration	Yes	Yes	Yes
data_declaration	Yes	Yes	Yes
task_declaration	Yes	Yes	Yes

© 2021 All Rights Reserved.

Package Constructs	Xcelium Compiler	Palladium HDL-ICE Compiler	IXCOM Compiler
function_declaration	Yes	Yes	Yes
checker_declaration	No	No	No
dpi_import	Yes	No	Yes
dpi_export	Yes	No	Yes, but only in testbench
extern_constraint_declaration	No	No	No
class_declaration	Yes	No	No
class_constructor_declaration	Yes	No	No
local_parameter_declaration	Yes	Yes	Yes
parameter_declaration	Yes	Yes	Yes
covergroup_declaration	No	No	No
overload_declaration	No	No	No
property_declaration	Yes	No	Yes
sequence_declaration	Yes	No	Yes
let_declaration	Yes, but does not support: Typed arguments Usage of let in bind statements	Yes, but does not support: Typed arguments Usage of let in bind statements	Yes, but does not support: Explicit package reference, such as `pack::item` inside bind statement is not supported for `let`.
anonymous_program	No	No	No
package_export_declaration	No	No	No
timeunits_declaration	Yes	Yes	Yes

Support for SVA and PSL Assertions with SystemVerilog Packages

Property and Sequence declarations in SystemVerilog packages and references to these by modules in the hardware partition is supported.

© 2021 All Rights Reserved.

Support for Coverage Constructs with SystemVerilog Packages

Code coverage and functional coverage constructs are not supported with SystemVerilog packages.

Support for DPI Declarations with SystemVerilog Packages

DPI import and DPI export declarations contained within a SystemVerilog package are supported by the IXCOM compiler. However, only DPI import declarations within a SystemVerilog package can be called by the modules in the hardware partition.

DPI import declarations contained within a SystemVerilog package are supported in the hardware partition. Therefore, DPI import tasks and functions declared within a SystemVerilog package can be called from modules within the hardware partition.

DPI export declarations contained within a SystemVerilog package are not supported in the hardware partition.So, DPI export tasks and functions declared within a SystemVerilog package are marked using the `translate_on` and `translate_off` pragmas, and those modules in the hardware partition that call the DPI export tasks or functions declared in the SystemVerilog package, are partitioned to the software side.

Handling of Stale Package References

The IXCOM compiler ensures that the module that is referencing a package is analyzed *after* the package is analyzed. This ensures database consistency. If it is found that the package has been re-analyzed at a later time, and the referencing module has not been re-analyzed, the IXCOM compiler issues an error message.

Partitioning of SystemVerilog Packages by the IXCOM Compiler

The IXCOM compiler determines the partition (hardware or software) for the SystemVerilog packages. By default, all SystemVerilog packages get partitioned to the software partition. Then, the IXCOM compiler moves the following types of SystemVerilog packages to the hardware partition:

- SystemVerilog packages referenced by modules in the hardware partition
- SystemVerilog packages referenced by other SystemVerilog packages in the hardware partition

All non-synthesizeable constructs contained within the SystemVerilog packages partitioned to the hardware, are marked with the `translate_on` and `translate_off` pragma. If a

© 2021 All Rights Reserved.

module (in the hardware partition) references a non-sythesizeable SystemVerilog package construct, it is partitioned to the software side.

Support for Compilation Unit Scopes in SA Mode

SystemVerilog allows objects to be declared outside of the module, interface, program, or package declaration. The scope for such objects is called the ***Compilation Unit Scope***. In accordance with the LRM, the IXCOM compiler decides the boundary of the *Compilation Unit Scope*. It also mirrors the Xcelium behavior.

Use the `+showSvPkgAccess` option of the `ixcom` (when using single-step compilation) or `vlan` (when using multi-step compilation) command to enable display of information messages when compilation unit scope objects are referenced.

In single-step compilation (only the `ixcom` command), IXCOM creates just one *Compilation Unit Scope*. In multi-step compilation (`vlan`, `vhan`, and `ixcom` commands), IXCOM creates a separate *Compilation Unit Scope* for each `vlan` session that you invoke. IXCOM denotes the beginning of the first file passed to the `vlan` session as the starting point and the end of the last file passed to the same `vlan` session as the ending point of the *Compilation Unit Scope* corresponding to that `vlan` session.

This means that a design can have multiple *Compilation Unit Scopes* - one for each `vlan` session. The IXCOM mirrors the Xcelium implementation in which a separate Compilation Unit Scope is created for each `xmvlog` session.

Objects declared within a *Compilation Unit Scope* are visible to all module, interface, program, and/or package declarations contained within that *Compilation Unit Scope*. These objects can be referenced directly by using <*object_name*> or explicitly by using $*unit*::<*object_name*>. The explicit form of reference is needed when an object is declared in the local scope with the same name as the *Compilation Unit Scope* object, thus hiding the visibility of the *Compilation Unit Scope* object.

Consider the following example:

```
// file test.sv
parameter int myVar = 32;

module top;
    int myVar;
    initial
      $display("The value of parameter myVar is %h", $unit::myVar);
endmodule
```

Objects declared within a *Compilation Unit Scope* are **NOT** visible outside of it. This means that the *Compilation Unit Scope* objects declared in one `vlan` session cannot be syntactically accessed in another `vlan` session.

© 2021 All Rights Reserved.

The Palladium HDL-ICE Compiler supports a single *Compilation Unit Scope* as starting at the beginning of the first file in the `vavlog` session and ending at the end of the last file in the same `vavlog` session. Only one `vavlog` session can have objects declared at the *Compilation Unit Scopes* level.

Each `vlan` session has its own distinct *Compilation Unit Scope*. Therefore, there is no notion of a *Compilation Unit Scope* being *re-declared*.

Supported Constructs

The list of supported constructs in the *Compilation Unit Scope* is the same as the list of supported constructs in the SystemVerilog package. Refer to the Support for SystemVerilog Package Constructs by Various Compilers table.

Assertions (SVA and PSL)

Just like with SystemVerilog packages, Property and Sequence declarations in the *Compilation Unit Scope* and references to them by modules in the hardware partition are supported.

Coverage

Code Coverage and Functional Coverage features are not supported with *Compilation Unit Scopes*.

DPI Declarations

Just like SystemVerilog packages, DPI import and DPI export declarations contained within a *Compilation Unit Scope* are supported by the IXCOM compiler. However, just like with SystemVerilog packages, only DPI import declarations within a *Compilation Unit Scope* can be called by the modules in the hardware partition.

During partitioning, the *Compilation Unit Scopes* are treated in the same manner as SystemVerilog packages.

© 2021 All Rights Reserved.

Support for SystemVerilog bind Directive for Instantiations in VHDL Designs

The SystemVerilog bind directive can be used to create an instance in a VHDL design in elaboration phase. The bind directive follows the SystemVerilog syntax like the use of positional connectivity, named connection, implicit named connection, and dot-star (.*) connection to instantiate within a VHDL design.

Consider the following example of the bind statement that creates an instance of a module (named bot) in a VHDL module that is bound to an instance (named mid1) in the top-level Verilog module (named top):

```
bind top.mid1 bot #(4,5,6) bot_1 (.in1(in1), .in2(in2), .o1(o1));
```

Following are the kind of connections that are supported:

- Use of dot-start connection
- Mix of dot-star and implicit named connection
- Mix of named connection and dot-star

Following are the kind of connections that are not supported:

- Escaped name or extended name
- Range instance

© 2021 All Rights Reserved.

19

Support for SystemVerilog Constructs

Important

The content of this chapter pertains only to SA mode.

The following table provides information about the support for the SystemVerilog 1800 - 2009 constructs by the IXCOM compiler. The following table lists all the constructs that are either partially supported or not supported. All the constructs that do not appear in the following table are fully supported in both software and hardware partitions.

- Partially Supported: Indicates that only some of the constructs in a section are supported.
- Not Supported: Indicates that the modules with these constructs should be compiled separately using ***xrun***. For more information, see the Compiling Testbenches Separately section. Details about the ***xrun*** tool are provided in the *xrun User Guide* available in the Xcelium documentation set.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
5. Lexical Conventions		
5.5 Operators	See section 11 Operators and expressions.	
5.6 Identifiers, keywords, and system names		
5.6.3 System tasks and system functions	See sections 20. Utility system tasks and system functions and 21. I/O system tasks and system functions	
5.6.4 Compiler directives	See section 22. Compiler directives.	

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
5.12 Attributes	Not Supported	Not Supported
5.13 Built-in Methods	See section 6. Data Types.	
6. Data Types		
6.6 Net types	Partially Supported Not Supported- 6.6.2 Unresolved nets	Partially Supported Not Supported- 6.6.2 Unresolved nets
6.11 Integer data types	Partially Supported - shortint, int, longint, byte, integer Not Supported- time	Partially Supported - shortint, int, longint, byte, integer Not Supported- time
6.12 Real, shortreal and realtime data types	Not Supported	Partially Supported Not Supported - shortreal
6.15 Class	Not Supported	Not Supported
6.16 String data type	Supported	Not Supported
6.18 User-defined types	Partially Supported Not Supported - Interface typedef scoping	Supported
6.19 Enumerated type ranges	Supported	Supported
6.20 Constants	Partially Supported Not supported - 6.20.2.1 $ as parameter value 6.20.1 is supported except inside classes and program blocks	Partially Supported Not supported - 6.20.2.1 $ as parameter value 6.20.1 is supported except inside classes and program blocks

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions

6.23 Type operator	Partially Supported:
	In instantiation and type parameter assignment to get the type of:
	■ A typedef datatype:
	`typedef enum logic [3:0]`
	`{`
	`  ADD = 'h0,`
	`  SUB? = 'h1,`
	`  DIVIDE = 'h2,`
	`  MUL = 'h3`
	`} Operation_e;`
	`module flop_stage #(parameter type DTYPE = logic) (output DTYPE q, input DTYPE d);`
	`....`
	`endmodule`
	`module test();`
	`  Operation_e op1;`
	`  Operation_e op2;`
	`  flop_stage #(type(Operation_e)) ff1 (op1, op2);`
	`endmodule`
	■ A parameter or variable whose type is a typedef
	`typedef logic node;`
	`module test(input node in1);`
	`  node clk, rst, n1;`
	`  localparam type  t_n1 = type(n1);`
	`  t_n1 temp_d_n1;`
	`endmodule`	

6.24 Casting		
6.24.1 Cast operator	Partially Supported Not Supported - `constant_cast` `$cast`	Partially Supported Not Supported - `constant_cast`
6.24.2 $cast dynamic casting	Not Supported	Supported
6.24.3 Bit stream casting	Not Supported	Not Supported

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
6.28 Parameterized data types (1800-2012 section)	Partially Supported Not Supported - ■ Non-integral parameters ■ Class in generate blocks ■ Dependency on static variables, tasks, or functions ■ Type parameters with unpacked values ■ Parameterized nested classes	Partially Supported Not Supported - ■ Non-integral parameters ■ Class in generate blocks ■ Dependency on static variables, tasks, or functions ■ Type parameters with unpacked values ■ Parameterized nested classes
7. Aggregate data types		
7.2 Structures	Partially Supported Not Supported - ■ Structure member initial values ■ Unpacked structure data type (OOMRs to members)	Partially Supported Not Supported - Structure member initial values
7.3 Unions	Partially Supported Not Supported - ■ 7.3.2 Tagged unions ■ Unpacked Union data type	Partially Supported Not Supported - 7.3.2 Tagged unions

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
7.4 Packed and unpacked arrays	Partially Supported Not Supported - ■ Hierarchical references to Queues, Dynamic, and Associative Arrays (QDAs) ■ 7.4.2 Unpacked arrays	Partially Supported Not Supported - ■ Hierarchical references to QDAs ■ 7.4.2 Unpacked arrays
7.5 Dynamic arrays	Not Supported	Supported
7.8 Associative arrays	Not Supported	Supported
7.9 Associative array methods	Not Supported	Supported
7.10 Queues	Partially Supported Refer to the Support for Bounded Queues section for details of the support and restrictions in the support for bounded queues in hardware.	Partially Supported Not Supported- 7.10.4 Updating a queue using assignment and unpacked array concatenation
7.12 Array manipulation methods	Not Supported	Not Supported
8. Classes	Not Supported	Not Supported
9. Processes		
9.3 Block Statements	Partially Supported Not Supported - ■ join_none (disable) ■ join_none (wait on automatic variables with wait or event controls)	Supported
9.4 Procedural timing controls		

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
9.4.1 Delay control	Partially Supported Not Supported - `wait_order` and sequence in non-assertion context	Partially Supported Not Supported - `wait_order` and sequence in non-assertion context
9.4.2 Event control	Partially Supported Not Supported - `wait fork`	Partially Supported Not Supported - `wait fork`
9.4.4 Level-sensitive sequence controls	Not supported	Not Supported
10. Assignment statements		
10.9 Assignment patterns	Partially Supported Not supported - ■ left hand side assignment ■ associations by type ■ simple 'type qualification for assignment to OOMRs or port connection expressions)	Partially Supported Not supported - ■ left hand side assignment ■ associations by type ■ simple 'type qualification for assignment to OOMRs or port connection expressions)
10.10 Array compatibility	Supported	Supported
10.11 Net aliasing	Not Supported	Not Supported
11 Operators and expressions		
11.2.1 Constant function as struct member initialization	Supported	Supported
11.4 Operator descriptions		
11.4.7 Logical operators	Partially Supported (only in context of operands having no side effects)	Partially Supported (only in context of operands having no side effects)

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
11.4.11 Conditional operator	Partially Supported Not Supported - `&&&` and `matches`	Partially Supported Not Supported - `&&&` and `matches`
11.4.13 Set membership operator	Not Supported	Not Supported
11.4.14 Streaming operators (pack/unpack)	Partially Supported Not Supported - 11.4.14.4 Streaming dynamically sized data	Partially Supported Not Supported - 11.4.14.4 Streaming dynamically sized data
11.9 Tagged union expressions and member access	Not Supported	Not Supported
11.11 Operator overloading	Not Supported	Not Supported
11.13 Let construct	Supported	Supported
12. Procedural programming statements		
	Partially Supported Not Supported - ■ `inside`, and `matches` ■ `unique0` supported only when the `-convertUnique0ToUnique` option is used ■ 12.6 Pattern matching conditional statements	Partially Supported Not Supported - ■ `inside`, and `matches` ■ `unique0` supported only when the `-convertUnique0ToUnique` option is used
13. Tasks and functions (subroutines)		

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
	Partially Supported Not Supported - ■ Function return - events ■ `const ref`	Supported
13.8 Parameterized tasks and functions (1800-2012 section)	Partially Supported Not Supported - ■ Non-integral parameters ■ Class in generate blocks ■ Dependency on static variables, tasks, or functions	Partially Supported Not Supported - ■ Non-integral parameters ■ Class in generate blocks ■ Dependency on static variables, tasks, or functions
14. Clocking blocks		
	Not Supported	Partially Supported Not Supported - 14.14 Global clocking
15. Interprocess synchronization and communication		
15.3 Semaphores	Not Supported	Supported
15.4 Mailboxes	Supported	Supported
15.5 Named events	Partially Supported Not Supported - 15.5.4 Event sequencing: wait_order()	Partially Supported Not Supported - 15.5.4 Event sequencing: wait_order()
16. Assertions		

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
16.3 Immediate assertions	Partially Supported Not Supported - ■ `assume` ■ `cover`	Partially Supported Not Supported - ■ `assume` ■ `cover`
16.4 Deferred assertions	Not Supported	Not Supported
16.5 Concurrent assertions overview	Partially Supported Not Supported - Global clocking	Partially Supported Not Supported - Global clocking
16.8 Declaring sequences	Partially Supported Not Supported with the following: ■ A checker ■ A generate block ■ local variable formal arguments ■ untyped as formal parameter types	Partially Supported Not Supported with the following: ■ A checker ■ A generate block ■ local variable formal arguments ■ untyped as formal parameter types
16.9 Sequence operations	Partially Supported Not Supported - 16.9.4 Global clocking past and future sampled value functions	Partially Supported Not Supported - 16.9.4 Global clocking past and future sampled value functions

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
16.13 Declaring properties	Partially Supported Not Supported - ■ Declaring properties in a clocking block ■ Using operators AND/OR ■ 16.13.17 Recursive properties ■ `case - endcase` in `property` statement ■ 16.13.7 Implies and iff properties ■ 16.13.14 Abort properties ■ 16.13.15 Weak and strong operators	Partially Supported Not Supported - ■ Declaring properties in a clocking block ■ Using operators AND/OR ■ 16.13.17 Recursive properties ■ `case - endcase` in `property` statement ■ 16.13.7 Implies and iff properties ■ 16.13.14 Abort properties ■ 16.13.15 Weak and strong operators
16.14 Multiclock support		
16.14.2 Multiclocked properties	Partially Supported Not Supported - Multiclocked property definition and Non-overlapping implication	Partially Supported Not Supported - Multiclocked property definition and Non-overlapping implication

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
16.15 Concurrent assertions	Partially Supported Not Supported - ■ 16.15.4 Restrict statement ■ Disabling of procedural concurrent assertions at run time ■ 16.15.7 Inferred value functions	Partially Supported Not Supported - ■ 16.15.4 Restrict statement ■ Disabling of procedural concurrent assertions at run time ■ 16.15.7 Inferred value functions
17. Checkers	Not Supported	Not Supported
18. Constrained random value generation		
18.3 Concepts and usage	Partially Supported Not Supported - `with` clause	Partially Supported Not Supported - `with` clause
18.4 Random variables	Not Supported	Partially Supported Not Supported - Usage of `rand` and `randc` modifiers for `struct` and `union` members
18.5 Constraint blocks	Not Supported	Partially Supported Not Supported: ■ 18.5.2 Constraint inheritance - pure constraint ■ 18.5.7.2 Array reduction iterative constraints
18.6 Randomization methods	Not Supported	Supported
18.7 Foreach inside `with` clause	Not Supported	Not Supported

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
18.8 Disabling random variables with rand_mode()	Not Supported - rand_mode() - members of unpacked structuresrand_mode() - members of unpacked arrays	Supported
18.11 In-line random variable control	Not Supported - 18.11.1 In-line constraint checker is not supported	Supported
18.12 Randomization of scope variables—std::randomize()	Not Supported The following are not supported - ■ Class members in constraints or arguments ■ Multiple constraint expressions after an if or else	Supported
18.13 Random number system functions and methods	Not Supported - The following are not supported - ■ 18.13.4 get_randstate() ■ 18.13.5 set_randstate()	Supported
18.17 Random sequence generation—randsequence	Not Supported	Not Supported
19. Functional coverage		
19.4 Using covergroup in classes	Not Supported	Not Supported

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
19.5 Defining coverage points	Partially Supported Ignore transition bins and illegal transition bins are not supported	Partially Supported Ignore transition bins and illegal transition bins are not supported
19.7 Specifying coverage options	Partially Supported Not Supported - Setting options within always blocks	Supported
19.8 Predefined coverage methods	Partially Supported Not Supported - 19.8.1 Overriding the built-in sample method	Partially Supported Not Supported - 19.8.1 Overriding the built-in sample method
19.9 Predefined coverage system tasks and system functions	Not Supported	Not Supported
20. Utility system tasks and system functions		
20.2 Simulation control system tasks	Not Supported	Supported
20.3 Simulation time system functions	Not Supported	Supported
20.4 Timescale system tasks	Not Supported	Supported
20.6 Data query functions		
20.6.3 Range system function (`$isunbounded`)	Not Supported	Not Supported

© 2021
All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
20.7 Array querying functions ■ `$right` ■ `$low` ■ `$high` ■ `$increment` ■ `$size` ■ `$dimensions` ■ `$unpacked_dimensions`	Not Supported	Supported
20.8 Math functions	Not Supported	Supported
20.11 Assertion control system tasks	Not Supported.	Partially Supported The following tasks are not supported for immediate assertions: ■ `$asserton` ■ `$assertoff` ■ `$assertkill`
20.12 Assertion action control system tasks ■ `$assertpasson` ■ `$assertpassoff` ■ `$assertfailon` ■ `$assertfailoff` ■ `$assertnonvacuouson` ■ `$assertvacuousoff`	Not Supported	Not Supported

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
20.13 Assertion system functions	Not Supported	Partially Supported Not Supported: ■ `$past_gclk` ■ `$rose_gclk` ■ `$fell_gclk` ■ `$stable_gclk` ■ `$changed_gclk` ■ `$future_gclk` ■ `$rising_gclk` ■ `$falling_gclk` ■ `$steady_gclk` ■ `$changing_gclk`
20.14 Coverage system functions	Not Supported	Partially Supported: ■ `$coverage_control` ■ `$coverage_save` (See $coverage_control and $coverage_save) Not Supported: ■ `$coverage_get_max` ■ `$coverage_get` ■ `$coverage_merge`
20.15 Probabilistic distribution functions	Not Supported	Supported
20.16 Stochastic analysis tasks and functions	Not Supported	Supported

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
20.18 Miscellaneous tasks and functions - $system	Supported	Supported
21. I/O system tasks and system functions		
21.2.1 The display and write tasks		
21.2.1.2 Format specifications	Partially Supported Not Supported - Format specifiers:%p, %u, %z [supported with $fwrite $fread, unformatted output not shown with $display]	Partially Supported Not Supported - Format specifier: %p
21.3 File input-output system tasks and system functions		
21.3.4 Reading data from a file	Partially Supported Not Supported - $fread extensions [optional argument (third) of “count” is not supported]	Supported
21.4 Loading memory array data from a file	Partially Supported Not supported for associative arrays: ■ `$readmemb` ■ `$readmemh`	Supported
21.4.3 File format considerations for multidimensional unpacked arrays	Not Supported	Supported

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
21.5 Writing memory array data to a file	Partially Supported Not supported for associative arrays and dynamic arrays: ■ `$writememb` ■ `$writememh`	Supported
22. Compiler directives		
22.5 `define, `undef and `undefineall		
22.5.1 `define	Supported	Supported
22.6 `ifdef, `ifndef and `elsif		
Allow usage of SystemVerilog keywords as identifiers	Partially Supported Not Supported - Disallow usage of directive names as identifiers No macro name	Supported
22.9 `unconnected_drive, `nounconnected_drive		
	Partially Supported Not Supported - ■ Handling these constructs within design elements ■ Usage of any other strengths except `pull0` and `pull1`	Partially Supported Not Supported - ■ Handling these constructs within design elements ■ Usage of any other strengths except `pull0` and `pull1`
23. Modules and hierarchy		
23.2 Module definitions		

© 2021
All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
23.2.2 Port declarations	Partially Supported Not Supported - Event ports, union ports, and ref ports are not supported	Partially Supported Not Supported - Event ports, union ports, and ref ports are not supported
23.3 Module instances (hierarchy)		
23.3.2 Module instantiation syntax	Partially Supported Not Supported - 23.3.2.4 Connecting module instances using wildcard named port connections (.*)	Partially Supported Not Supported - 23.3.2.4 Connecting module instances using wildcard named port connections (.*)
23.3.3.5 Formal Port Datatype	Partially Supported Not Supported - Range of instance having packed formal and unpacked actual and vice versa	Partially Supported Not Supported - Range of instance having packed formal and unpacked actual and vice versa
23.4 Nested modules	Not Supported	Not Supported
23.5 Extern modules	Not Supported	Not Supported
23.11 Binding auxiliary code to scopes or instances	Supported	Supported
24. Programs		
	Partially Supported Not Supported - Anonymous programs in compilation units and packages	Partially Supported Not Supported - Anonymous programs in compilation units and packages
25. Interfaces		

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
25.3 Interface syntax Interface_nonansi_header [timeunits_declaration]	Partially Supported Not Supported: ■ Nested interfaces ■ Interfaces in generates ■ Interfaces connected to ports of interface	Supported
25.5 Modports	Partially Supported Not Supported: ■ complex expressions in modport ■ genvar ■ 25.5.5 Clocking blocks and modports- `modport_clocking_declaration` is not supported	Partially Supported Not Supported: ■ complex expressions in modport ■ genvar ■ 25.5.5 Clocking blocks and modports- `modport_clocking_declaration` is not supported
25.6 Interfaces and specify blocks	Partially Supported Not Supported - `specify` blocks	Partially Supported Not Supported - `specify` blocks

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
25.7 Tasks and functions in interfaces	Partially Supported Not Supported - ■ `modport_tf_ports_declaration (import)` ■ `modport_tf_ports_declaration` (export) ■ Multiple task export ■ Modports - clocking keyword on Modports ■ Extern fork-join tasks	Partially Supported Not Supported - ■ `modport_tf_ports_declaration (import)` ■ `modport_tf_ports_declaration` (export) ■ Multiple task export
25.8 Parameterized interfaces	Partially Supported Not Supported - Access without ports (static interface)	Supported
25.9 Virtual interfaces	Partially Supported Not Supported - virtual interface member reference	Partially Supported Not Supported - virtual interface member reference
26. Packages		

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
26.2 Package declarations	Partially Supported Not Supported: ■ Static/automatic lifetime specification in package header ■ Anonymous programs in packages ■ Package export declaration ■ Covergroups in packages ■ Overload declaration	Partially Supported Not Supported: ■ Static/automatic lifetime specification in package header ■ Anonymous programs in packages ■ Package export declaration ■ Covergroups in packages
26.6 Exporting imported names from packages	Supported	Supported
27. Generate constructs		
27.4 Loop generate constructs	Partially Supported The name for each `for-generate` block instance is different in Xcelium and IXCOM. For example, the name in Xcelium is `genblk_name[<instance_number>]` and in IXCOM, the name is `genblk_name_<instance_number>_`	Partially Supported The name for each `for-generate` block instance is different in Xcelium and IXCOM. For example, the name in Xcelium is `genblk_name[<instance_number>]` and in IXCOM, the name is `genblk_name_<instance_number>_`
27.6 External names for unnamed generate blocks	Partially Supported The name for each unnamed generate blocks is different in Xcelium and IXCOM.	Partially Supported The name for each unnamed generate blocks is different in Xcelium and IXCOM.

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
28. Gate-level and switch-level modeling		
28.16 Gate and net delays	Not Supported	Supported
30. Specify blocks	Not Supported	Supported
31. Timing checks	Not Supported	Supported
33. Configuring the contents of a design	Not Supported	Not Supported
34 Protected Envelopes	**Note:** Encryption should be done using the `xmprotect` utility.	**Note:** Encryption should be done using the `xmprotect` utility.
35. Direct programming interface (DPI)		
35.5 Imported tasks and functions	Partially Supported Not Supported - Export function/tasks - time consuming	Supported

© 2021 All Rights Reserved.

IEEE 1800-2009 LRM Section and Constructs	Hardware Partition Restrictions	Software Partition Restrictions
35.5.6 Types of formal arguments	Partially Supported Not Supported - ■ Export function return types - string ■ Import/export function/task arguments - packed union of type bit/long ■ Import/export function/task arguments - enums ■ Import/export function/task dynamic array arguments (non-strings) ■ Import/export function/task dynamic array arguments ■ Import/export function return types - shortreal	Supported
37. VPI object model diagrams	Not Supported	Supported
39. Assertion API	Not Supported	Supported
40. Code coverage control and API		
40.2 Overview	Partially Supported Not supported - 40.2.1 SystemVerilog coverage API	Supported
Annex G **Std package**	Not Supported	Supported

© 2021 All Rights Reserved.

Limitations in the Support for SystemVerilog Constructs

This section describes limitations in support for SystemVerilog constructs.

Difference in Behavior in Software and Hardware Runs for Sampled Value Functions Used in Continuous Assignment Statements

When using sampled value functions, such as `$past`, in continuous assignment statements, some mismatches can be observed when the same design is run on the hardware emulator and the software simulator. These mismatches occur because of the way hardware samples values internally. Consider the following examples:

Example 1:

```
module dut;
   assign out1 = $past(in1,,,@(posedge CLK));
   always @(posedge CLK)
      $strobe("in1=%d , out1=%d",in1,out1);
endmodule
```

In the above example, `out1` is expected to have the previous or the past value of `in1`. But the current value, and not the earlier sampled value of `in1`, is displayed as the value of `out1`.

Example 2:

```
module dut (clk,in1,a,b,out1);
    input [3:0] in1;
    input wire a,b;
    input clk;
    output [3:0] out1;
    reg [3:0] out1;
    reg mytaskr=1'b0;
    wire mytaskw;
    assign out1= $past(in1, , , @(posedge clk));
    always @(posedge clk)
      mytaskr <= ~mytaskr;
    ixc_pulse(mytaskw,mytaskr);
    axis_tbcall(mytaskw,"t1");
       task t1;
           $display("in1=%b,out1=%b",in1,out1);
       endtask
endmodule
```

When the above module is run on the hardware emulator; the task `t1` runs on the software simulator and reads a variable, `out1` that has been assigned a value using the `$past` sampled value function. The testbench call region will not display the earlier sampled value of `in1` as the value of `out1`.

You will see incorrect values because the values for sampled value functions like `$past` are captured during the postponed region of a cycle to be used in the next timeslot.

© 2021 All Rights Reserved.

Constructs like `$strobe` and `$monitor` also operate in the postponed region of a timeslot but after the sampling code has executed. The testbench calls occur after the postponed (read-only) regions of the time slot. Hence, the values read by constructs like `$monitor`, `$strobe`, and the testbench calls are the current values and not the previously sampled values.

Handling of Concatenation Operators

A zero or negative multi-concatenation multiplier is not supported in SystemVerilog modules. A replication with a zero replication constant is considered to have a size of zero, and is ignored. Such a replication can appear only within a concatenation in which at least one of the operands of the concatenation has a positive size.

- For SystemVerilog modules:

  `{0{expr}}` within a non-zero size concatenation is ignored with a warning message.

  An error is generated in the following cases:

  - `{0{expr}}` within a zero size concatenation multiple
  - `{0{expr}}` within a zero size concatenation
  - `{0{expr}}` not within a concatenation

- For Verilog 2K modules, each occurrence of `{0{expr}}` is converted to `1'b0` in all the below cases:

  - `{0{expr}}` within a non-zero size concatenation
  - `{0{expr}}` within a zero size concatenation multiple
  - `{0{expr}}` within a zero size concatenation
  - `{0{expr}}` not within a concatenation

Support for Bounded Queues

This section describes the IXCOM support for SystemVerilog bounded queue in the DUT. It specifies the way IXCOM transforms a bounded queue and its operations in DUT for synthesis and provides a simple use model of bounded queue in hardware verification.

Note: The behavior for the supported operations on the queue is SystemVerilog 1800-2012 LRM-compliant unless stated otherwise.

© 2021 All Rights Reserved.

As defined in Section 7.10 of SystemVerilog LRM, 1800-2012, a queue is a variable-size, ordered collection of homogeneous elements. It is analogous to a one-dimensional unpacked array that grows and shrinks automatically. The size of a queue can be limited by specifying a maximum size or it can be unlimited. These two types of queues are respectively referred to as bounded queue and unbounded queue. Due to the dynamic nature of a queue and its operations, RTL synthesis tools, such as HDL-ICE Compiler do not natively support it.

IXCOM transforms the bounded queues. However, its support is limited to bounded queues that are both defined and used in hardware. Unbounded queues or references from the software code to a queue declared in hardware are not supported. Thus, queues that are either defined or used in software are not supported. Also, software code operating on a hardware queue indirectly through a hardware task or function is not supported unless the hardware task (or function) is an `export` function.

Note: In this section, the terms *queue*, *bounded queue*, and *queue in hardware* are used interchangeably and all of these terms refer to a *bounded queue in hardware*.
The terms *method* and *operation* are used interchangeably to refer to any operation that can be performed on a queue. Hence, the use of a method, such as `push_back` on a queue may be referred to as an operation on the queue, or vice-versa.

IXCOM supports only those bounded queues that are declared in the module scope. It does not transform queues declared in any other scope, such as, task-function scope, named or unnamed block scope, process scope, package scope, compilation-unit scope, generate block scope, and so on. Declarations of queue variables within any scope other than a module scope are ignored. Unless stated otherwise, the queue variables discussed in this section refer to queues declared in the module scope in hardware.

Enabling Support for Bounded Queues in Hardware

Use the following options of the `ixcom` command to enable support for bounded queues:

`+boundedQueue`	Enables support for bounded queues in all hardware modules.
`+boundedQueue+<mod1>+<mod2>+...`	Enables support for bounded queues in specific hardware modules, that is <*mod1*> and <*mod2*>.

Following is an example to illustrate the support for bounded queues in hardware.

```
test.sv:
module top;
   bit clkx; always #3 clkx = ~clkx;
   bit clky; always #5 clky = ~clky;
```


© 2021 All Rights Reserved.

```
   bit [31:0] i, j, d1, d2;
   bit [31:0] q1[$:30];
   bit [31:0] q2[$:30];
   always@(posedge clkx) begin
           if(q1.size() < 31 & q2.size() < 31) begin
              i = i+1;
              q1.push_front(i);
              q2.push_back(i);
           end
   end

   always@(posedge clky) begin
           if(q1.size() > 0 && q2.size() > 0) begin
              j = j+1;
              d1 = q1.pop_back();
              d2 = q2.pop_front();
              if(d1 != j || d2 != j)
                 $display($time, " error %d %d %d", d1, d2, j);
           end
   end
endmodule

module tb;
   initial #1000 $finish;
endmodule
```

For the above example, you can use the following command to compile the design with support enabled for bounded queues:

```
ixcom -ua +dut+top +iscDelay+top test.sv +boundedQueue
```

A hardware module for which bounded queue support is enabled is referred to as a bounded queue module. The `+boundedQueue` option specified for a software module is ignored.

Supported Operations in Bounded Queues

Support for bounded queues in hardware is applicable only to the following queue methods and operations:

- `size()`
- `pop_front()`
- `push_front()`
- `pop_back()`
- `push_back()`
- Indexing operation

Any other queue methods, such as, `insert` or `delete`, or queue operations, such as, queue-slice expression or part-select expression, are not supported. Similarly, operations that operate on an entire queue are not supported.

© 2021 All Rights Reserved.

Restrictions in the Support for Operations in Bounded Queues

Following are the restrictions related to the supported operations:

- Indexing operator to read from or write to a queue index can only be used within an assignment statement.

```
int Q1[$ : 6], x, y;
Q1[3] = x;
y = Q1[2] + 5;
```

- Queue references are not supported within the initialization statements, conditional expression, or step statements of a FOR loop, or within the conditional expression of a WHILE, DO_WHILE loop, or in the argument of a FOREACH loop.

```
for (k =0; k < q1[idx]; k++, a = q1[j]) begin // not supported
     q1[idx] = k;
end
```

- Non-blocking assignments that assign queue elements using indexing operator are not supported.

```
int Q1[$:6];
always @(posedge clk) begin
         Q1[0] <= 5;                // not supported
end
```

- Assignment operators are not supported with queues in hardware. So, an initial value cannot be provided to a queue variable in the declaration. Thus, a declared queue variable is initialized to an empty queue.

```
integer q10 [$ : 6] = { 1, 2 }; // not supported
integer q10 [$ : 6];            // supported, q10 initialized to an empty queue
```

- References to the last index of a queue using the `$` symbol are not allowed:

```
int Q1[$ : 6], i;
Q1.push_back(1);
Q1.push_back(2);
i = Q1[$];      // not supported
```

- Recursive tasks and functions containing queue references within bounded queue modules that are called from the hardware scopes are not supported. Also, tasks and functions containing queue references within bounded queue modules that are called from the hardware scopes and that cannot be in-lined are not supported.

```
int Q1[$:6], i;
always @(posedge clk) begin
         i = f1(0);        // not supported. f2 is a recursive TF.
end

function automatic int f1(int x);
         return f2(x);
endfunction

function automatic int f2(int x);
         x = x+1;
         if (x == 5) begin
```


© 2021 All Rights Reserved.

```
            Q1.push_back(x);
            return x;
        end
        else return f2(x);
endfunction
```

- Queue method calls and operations are not supported within the condition expression of a `wait` statement or within a `fork...join` block.

```
int Q1[$ : 6], i;
wait(Q1[5]) a = b;           //not supported
fork                         //not supported
     i = Q1[1];
     Q1.push_back(4);
join
```

- Queue references are not supported within primitives, in input-output signals, in port declarations, or in port expressions for a module or a primitive instance. Also, queue references within tasks and functions that are called from these constructs are not supported.

```
int Q1[$ : 6];
wire w;
assign w = f();       //not supported
function bit f();
         Q1[0] = 5;
         return 1;
endfunction
```

- Operations, such as assignment of a queue variable to another, concatenation of queue variables or of slices of queue variables, comparison of queue variables using equality or inequality operators, or queue operations or methods other than the ones listed as being supported, are not allowed.

  - Slice expression:

    ```
    Q3[ 2 :4 ]       // not supported
    ```

  - Concatenation:

    ```
    bit Q5[$ : 6];
    bit Q6[$ : 2];
    "{Q5, Q6}"     //not supported
    ```

  - Equality or Inequality operation:

    ```
    'Q5 == Q6'       //expression not supported
    'Q5 != Q6'       //expression not supported
    ```

An error is generated for the unsupported use of queues. Errors might not be generated for an unsupported queue variable declaration in a bounded-queue module unless the variable is used in the design.

© 2021 All Rights Reserved.

Support for References

Only local references to queues are supported. Hierarchical references to queue variables are not supported.

Example:

```
module m;
    bit Q[$ : 6], i;
    bit Q1[$];          // unbounded Queue. Not supported
    bit Q2[$ : 14];     // supported
    always @(posedge clk) begin
             bit Q3[$ : 14];      // not supported within process scope
             i = m.Q.size();        // hierarchical reference not supported
             i = mInst.Q.size();    // hierarchical reference not supported
             i = Q.size();          // supported
    end
    task t;
           bit Q4[$ : 14];                 // not supported within TF scope
    endtask
endmodule
```

Support for Datatypes

IXCOM supports only the following types of queue variables:

- A queue variable that stores elements of packed and variable datatypes that are supported in hardware. Queues that store elements of unpacked or dynamic or net datatypes are not supported. So, unpacked data types, such as, unpacked array or unpacked struct, string, or datatypes, which are not supported in hardware, such as class, or net datatypes, such as wire, are not allowed to be the element datatypes stored in a queue.
- Only a queue variable that represents a single queue is supported. Arrays of bounded queues or bounded queues of unpacked arrays or struct are not supported. For example:

```
typedef struct {int addr; byte data;} S;
S q1[$ : 30];             // unpacked struct type unsupported
wire  q2[$: 14];          // wire element type not supported in bounded queue
typedef class C;
C q9[$: 30];              // class element type not supported in hardware
typedef int T[$:7];
T q3[0:3];                // array of bounded queues not supported in hardware
```

Limitations in Support for Bounded Queues in Behavioral Modules

This section describes the restrictions in the support for bounded queues in behavioral modules.

© 2021 All Rights Reserved.

2-state data

The hardware emulation stores data in the 2-state form. DUT queues that store 4-state elements would operate on the respective 2-state forms during emulation.

Thus, the behavior of hardware run for DUT queues that store 4-state elements can be non-compliant with SV-1800 LRM and different than their SW simulation behavior. In SW simulation, the 4-state values may be preserved.

Example:

```
integer q1[$ : 2];
integer i;
q1.push_back(5);
i = q1.pop_front();   // after this statement: i is 5, q1 is empty
i = q1.pop_back();    // value read from a nonexistent array entry of 2-state
integral type is ‘0 unlike ‘X for 4-state integral type. Since emulation stores
2-state data, ‘i’ will become 0.
```

Push_front behavior

If `push_front` is used to add elements to a bounded queue has reached its maximum size, then the behavior of `push_front` method is as follows:

- The queue is unchanged.
- The elements to be added, are discarded.
- A warning is issued.

This behavior is non-compliant with the SV-1800 LRM, which specifies the following to occur when `push_front` is called on a full bounded queue.

- The highest numbered element of the queue before `push_front` was called, is deleted.
- The element to be added, is added to the front.
- A warning is issued.

Following is an example of the behavior of `push_front` that is supported:

```
integer q2[ $ : 2], q1[$ : 2], i;
q2.push_back(5);
q2.push_front(6);
q2.push_front(7);
q2.push_front(8);    // integer 8 will be discarded, q2 remains {7, 6, 5 }
i = q1.pop_back();   // i becomes '0' as pop_back returns '0', \
                     // the default value of a 2-state integral \
                     // type 'int', q1 remains empty
```


© 2021 All Rights Reserved.

Concatenation behavior

The SystemVerilog LRM 1800-2012 does not specify the order of assignment to the variables on the LHS of the concatenation operation. So, behavior of some queue-indexing operations that increase the size on the queue on the LHS of the concatenation might be different than the simulation behavior.

The queue-indexing operations on the LHS of concat are assigned in the order in which these appear in the LHS expression. Following are some examples of queue indexing on LHS of concatenation operation:

Example of supported behavior:

Two elements are inserted at the tail end of the queue. First, `q[2]` is added, which will be assigned the value of `q[0]` and the queue size grows to three. Then, `q[3]` is added and assigned the value of `q[1]`. The final queue size is four.

```
int q[$ : 14];
q.push_back(6);
q.push_back(7); // 'q' after 2 push_back operations is {6, 7} and the size is 2
{q[2], q[3]} = {q[0], q[1]};             // No error. Final queue size is 4.
```

Example of unsupported behavior that generates a run-time error:

The concatenation on the LHS leads to insertion of only one new element to the queue. First, `q[3]` assignment attempt is unsuccessful because the queue size is two and the highest valid index to which the queue index write operation can occur is two. Thus, writing to `q[3]` generates an error message and no element is added to the queue. Then, assignment to `q[2]` adds an element at index two with value `q[1]` and increases the size of the queue to three.

```
int q[$ : 14];
q.push_back(6);
q.push_back(7);              // 'q' after 2 push_back operations is {6, 7} \
                             // and the size is 2.
{q[3], q[2]} = {q[0], q[1]}; // Error generated for write to q[3].\
                             // Final queue size is 3.
```

Support for Synthesis of Bounded Queues

The support for using queues in hardware is limited in terms of the producers and consumers to the ends of queues. An operation that can increase the number of elements in the queue is referred to as a producer because it produces or writes data to the queue. Similarly, an operation that can decrease the number of elements in the queue is referred to as a consumer because it consumes data from the queue.

This limitation depends on whether the module within which a queue is declared and used is an RTL module or a behavioral module, or whether or not the queue is referenced within an

© 2021 All Rights Reserved.

`export` function. These limitations are described in the Limitations in the Support for Synthesis of Bounded Queues section.

For a given queue, the following methods or operations are producers for the queue:

- `push_back`
- `push_front`
- Indexing write operation

  The indexing write operation is a producer of the queue because it can potentially add an element to a queue by writing to the `$+1` index, which would insert an element at the tail-end of the queue.

Producer methods/operations that add data to the front of the queue are called producers to the front or to the front-end of the queue. The only producer operation to the front of a queue that is within the scope of this document is `push_front()`.

Producer methods and operations that add data to the tail end of the queue are called producers to the end or to the tail end of the queue. The indexing write operation to any index of the queue is a producer to the tail-end of the queue because it can append data only to the tail end of the queue. Indexing write to index `0` of an empty queue is referred to as writing to the tail end of the queue. The only producer operations to the tail end of a queue that are supported are `push_back()` and indexing write operation.

The following methods and operations are consumers for a queue:

- `pop_back`
- `pop_front`
- Export function

  An `export` function is a hardware task or function that belongs to any of the following forms.

  - `sfifo` marked SystemVerilog functions
  - `tb_export` marked SystemVerilog functions
  - `sfifo` marked SystemVerilog tasks that IXCOM might transform to `always` processes

- Invalid index

  An invalid index for a read operation on a queue is an index that lies outside the range of 0 (index of the first element) to $ (index of the last element).

© 2021 All Rights Reserved.

```
integer Q1[$ : 14]; // bounded queue, integer elements. Maximum size 15
integer i;
i = Q1[2];     // Q1[2] refers to the 3rd element in the queue. Since Q1
has no elements, i becomes '0', the default value of a 2-state integral
type 'int'.
```

```
integer Q2[$ : 6]; // bounded queue, integer elements. Maximum size 7
integer j;
Q2.push_front(1);
Q2.push_front(2);
j = Q2[1];   // Q2[1] refers to the 2nd element in the queue. j becomes '1'.
```

An invalid index for a write operation is an index that lies outside the range of 0 to $+1, where the expression "$ + 1" refers to the value of the last index of the queue before the write operation plus 1.

```
integer Q3[$ : 14]; // bounded queue, integer elements. Maximum size 15
integer i = 5;
Q3.push_front(1);
Q3.push_front(2);
Q3[4] = 5;        // operation ignored and a warning issued
```

- Select operation

  For queues that store supported packed types, member-select or bit-select operation on queue indexing expression can add additional elements to the queue. This can occur when writing to a queue using select operation at the last valid index for the write operation.

```
bit[0:10][0:5] Q1[$:14];
Q1[0][1][1] = 1;        // Adds an element at index 0 of the queue
```

```
typedef struct packed {
        int a;
  } st;
```

```
st stq [$:62];
st[0].a = 3;          // Adds an 'st' element element at index 0 of the
queue
```

Note: The indexing read operation is not a consumer of the queue because it cannot affect the number of elements in the queue.

Limitations in the Support for Synthesis of Bounded Queues

This section describes the restrictions in the support for bounded queues in hardware. Queues that are declared or used in software are not supported.

- Limitations in Size of Bounded Queues:

  The maximum number of elements (bound) of a bounded queue is specified in its declaration. Our implementation is to use an unpacked fixed array of size equal to the next power of 2 GREATER than the declared queue size. For example, we are using an unpacked array of size 16 to support a bounded queue Q[$:7], and size 8 for Q[$:6]. However, the implementation should not change the semantic of the declared bounded

© 2021 All Rights Reserved.

queue. That is, the out-of-bound check should still be based on the declared queue size. For example, a bounded queue Q[$:7] already has eight elements. Regardless of an unpacked array of size 16 in the implementation, an attempt to add another element by push_back (or push_front) or index operation Q[8] will result in a runtime warning.

- Limitations in Producer-Consumer Processes:

  Following are the limitations on the number and form of producer and consumer processes in which a queue can be referenced:

  When either of the following conditions (a) or (b) is true, then the producer-consumer limitation listed below applies:

  a. The module containing the queue is an RTL module

  b. A reference to a producer or consumer operation of the queue occurs within an export function.

  Only one `always` process that contains producer operations on the queue and at only one `always` process that contains consumer operations on the queue, and if a producer `always` process as well as a consumer `always` process operating on a queue are present, these must add data to the queue only at one end of the queue and consume data from the queue only at the opposite end of the queue.

  In other words, if a queue variable is declared in an RTL module or if a queue variable in any type of module (RTL or behavioral, or any other form) is referenced as a producer or as a consumer within an `export` function, then the references to the queue are limited to only those allowed per the above producer-consumer limitation.

Examples to Illustrate Support for Bounded Queues

Following are examples of queues declared in an RTL module:

- Example for supported push and pop operations from opposite ends of the same queue:

```
int Q[$:14];
int i;
always @(posedge clk) begin
   Q.push_back(i);
end
always  @posedge clk) begin
   i = Q.pop_front();
end
```

- Example for unsupported push and pop operations from the same end of the queue:

```
int Q[$:14];
int i;
always @(posedge clk) begin
         t();
end
```


© 2021 All Rights Reserved.

```
always  @(posedge clk) begin
          i = Q.pop_back();
end

task t();
    Q.push_back(i);
endtask
```

- Example-3: Not supported. 1 producer and 1 consumer at the same end (tail end) of the queue (the first always block is a producer that can append data to the back of Q)

```
int Q[$:14];
int i = 0, j = 0, k;
always @(posedge clkx) begin
   Q[j] = i;   // write operation on Q using indexing op is a producer for Q.
Indexing write places value 'i' at Q[0].

    i++;

end

// consumer always block that consumes from the back of queue Q
always  @posedge clky) begin
    k = Q.pop_back();
end
```

- Example-4: Not supported. Two always producer blocks adding data to the back of the queue (the first always block is a producer that can append data to the back of Q)

```
int Q[$:14];
int i = 0, j = 0, k;
always @(posedge clk) begin
   Q[j] = i;   // write operation on Q using indexing op is a producer to the
tail-end of the Q.

    i++;

end


always  @(posedge clk) begin     // producer to the front of the Q
    Q.push_front(j);
end
```

Following are examples of queues declared in a behavioral module:

- Following is an example of the restrictions in support for a behavioral module with producer or consumer references to a queue in an export function. This limitation applies due to the `export` function `hw_f1`. The example contains a producer reference to the queue in an `export` function and has both producer and consumer processes for the front of the queue. Thus, the module fails the restriction related to producers and consumers of a queue referenced in export functions despite it being a BC module (non-RTL module).

```
int Q[$:14];
int i;
$ixc_ctrl("map_bc");                    // Behavioral module
initial $ixc_ctrl("tb_export", "hw_f1");
always @(posedge clk) begin
         hw_f1();
end
```


© 2021 All Rights Reserved.

```
always @(posedge clk) begin
         i = Q.pop_front();
end

function void hw_f1();
  sz = Q.size();
  $display($time, " Before push: Size of Q: %d", sz);
  Q.push_front(i);
endfunction
```

- Following is an example of a supported behavioral module with multiple producer and consumer references to a queue at the same end of the queue in a regular hardware function within a behavioral module.

```
int Q[$:14];
int i;
$ixc_ctrl("map_bc");        // Behavioral module
always @(posedge clk) begin
         hw_f2();
end

always @(posedge clk) begin
         i = Q.pop_front();
end

function void hw_f2();
    sz = Q.size();
    $display($time, " Before push: Size of Q: %d", sz);
    Q.push_front(i);
endfunction
```

- References to a queue variable from a combinational block in a module are not supported. This limitation applies for all types of modules including behavioral modules. As a result, the following forms of `always` processes, which are combinational blocks, are not supported.

```
always_comb
always_latch
always @ ( * )
int Q[$:14], i;
always @(*) begin
         i = Q.push_back(1);      // not supported
end
```

- Level-triggered `always` blocks are not supported:

```
int Q[$:14], i;
bit a, b;
always @(a or b) begin
      i = Q.push_back(1);     // not supported
end
```

- The `always` blocks containing both level triggers and edge triggers in the sensitivity list are not supported:

```
int Q[$:14], i;
bit a, b;
always @(a or posedge b) begin
         i = Q.push_back(1);         // not supported
end
```


© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

20

Coverage Support

Coverage is defined as the percentage of verification objectives that have been met. It is used as a metric for evaluating the progress of a verification project to reduce the number of simulation cycles spent in verifying a design.

Coverage metrics in verification fall under the following categories:

- Code coverage: those that can be automatically extracted from the design code.

  In this chapter, the code coverage support is detailed in Code Coverage.

- Functional coverage: those that are user-specified to tie the verification environment to the design intent or functionality.

  In this chapter, the functional coverage support is detailed in Functional Coverage Support with Palladium Z1.

The IEEE 1800 - 2009 standard for SystemVerilog provides a detailed description of these terms and their differences.

Important

The content of this chapter is applicable only for the designs compiled using the IXCOM compiler. Details of using the IXCOM compiler are provided in Chapter 16, "Compiling Designs for Simulation Acceleration Using IXCOM."

Xcelium Versions Required for Coverage Features

The coverage features are tightly integrated with the Xcelium simulator. Xcelium does not support merging coverage results that originate from different major versions of Xcelium. Therefore, the version of xmsim used at the time of running the design on the Palladium emulator must be consistent for all runs in order to merge results.

Over the past few releases, Palladium coverage results have been integrated into a single Palladium Unicov database (typically named `uxe*.ucd`.) All of the coverage types are integrated into the Palladium coverage database.

© 2021 All Rights Reserved.

This helps preserve the original HDL when analyzing coverage data and creates one location for coverage results generated by Palladium Z1. The alternative to a Palladium Unicov database is an ICC Unicov database, which is generated by Xcelium, and named `icc*.ucd`.

It is recommended that MDV 17.10 be used to analyze coverage results.

Coverage Databases and Co-simulation

In a co-simulation environment, coverage can be handled by both the simulator and the emulator. Therefore, it is important to know which engine is going to handle which coverage item so that you can use the correct options and commands.

The following diagrams explain the handling of coverage items by the simulator and emulator engines.

The first diagram shows a scenario in which the `+xcDesignTop` option is used to ensure that the testbench is handled completely by the simulator, and the DUT is handled by Palladium emulator. This is the simplest example, and the recommended use model when you need coverage in the testbench and DUT.

Note: For completeness, an imported netlist is shown within the DUT. Imported netlists are, however, not instrumented for coverage. The coverage items handled by the simulator are governed by simulation options and commands. The end target for coverage handled by simulation is an ICC Unicov database.

Coverage within the DUT is written into a UXE Unicov database. Both UXE and ICC Unicov databases can be loaded in IMC and vManager. The coverage items handled by Palladium are subject to Palladium compile options and run-time commands. Some of these are equivalent to Xcelium coverage commands.

The diagram shows the coverage output going to separate databases. However, it is placed within the same coverage test directory. IMC loads both databases when you open this coverage test directory.

© 2021 All Rights Reserved.

Figure 20-1 Coverage Handling in Co-Simulation Environment

The second diagram is the same as the first diagram except for the addition of a testbench shown to be compiled by IXCOM. The following diagram illustrates a mixed testbench environment in which a portion of the testbench is handled by IXCOM, and the other portion is handled by the simulator. The database in which a coverage item is placed determines the options and commands that will govern the item, and the resulting database (ICC Unicov or UXE Unicov) into which the coverage will be written.

Coverage items in the testbench that is compiled by IXCOM are handled as follows:

- The covergroups that can be pushed into the DUT are placed in the emulator. Otherwise, the covergroups are left in the testbench.
- The assertions in the testbench are always written to the Palladium database.
- Code coverage is not supported in the testbench.

© 2021 All Rights Reserved.

Figure 20-2 Coverage Handling with Testbench Compiled by IXCOM

© 2021 All Rights Reserved.

Code Coverage

In code coverage, the information monitored is inferred from the design. The IEEE standard focuses on defining functional coverage items and minimal detail regarding code coverage. Therefore, in the absence of an industry standard for code coverage, the Cadence's Integrated Comprehensive Coverage (ICC) solution is used as a basis. With respect to the ICC definitions, Palladium Z1 supports toggle and block forms of code coverage.

Palladium Z1 supports the following types of code coverage:

- **Toggle Coverage** - Provides information about the change (toggle) of signals and ports, during a simulation run. It measures activity in the design, such as unused signals, signals that remain constant, or signals that have too few value changes. This coverage metric tracks the 0->1 and 1->0 transitions of signals and ports.
- **Block Coverage** - Identifies the lines of code that get executed during a simulation run. It helps you determine if the various testbenches exercise the statements in a block. This coverage metric tracks the HDL blocks that have been executed.
- **Expression coverage -** Factorizes logical expressions and monitors them during simulation run. It provides the metrics to quantify the degree of verification completeness. It measures the thoroughness with which the testbench exercises expressions in assignments and procedural control constructs (if or case conditions). It identifies each input condition that makes the expression true or false and whether that condition happened in simulation. Expression coverage provides finer granularity of coverage metrics than other code coverages, such as block and branch coverage.

The toggle coverage support with Palladium Z1 is described in Toggle Coverage Support with Palladium Z1 on page 1003 and the block coverage support with Palladium Z1 is detailed in Block Coverage Support with Palladium Z1 on page 1013. For details about expression coverage, refer to the Expression Coverage Support with Palladium Z1 section.

For more information on the other types of code coverage, refer to the following chapters of the *ICC User Guide* available in the Xcelium documentation set:

- *"Block, Branch, and Statement Coverage" Chapter* for an overview of Block Coverage in ICC.
- *"Toggle Coverage" Chapter* for an overview of Toggle Coverage in ICC.
- *"Expression Coverage - Control Scoring" Chapter* for an overview of Expression Coverage in ICC.

© 2021 All Rights Reserved.

Code Coverage Limited to DUT on Palladium Z1

The primary purpose of code coverage is to ensure that there are no logically unreachable or untested areas of the design. On Palladium Z1, code coverage applies only to the design units that are targeted for the emulator. The design units that comprise the testbench or extracted hierarchies are not candidates for code coverage.

Note: If required, you can use the `+xcDesignTop` option of the `ixcom` command and compile the testbench to run in a separate simulation environment. Refer to Compiling Top-Level Testbenches Using +xcDesignTop on page 654 for details. Note that only one `+xcDesignTop` specification is supported for a given design.

Differences between the Emulation Coverage Space When Compared to Simulation

The coverage results captured during emulation do not always completely match the data captured from comparable simulation runs. Often, there is not a complete overlap of the coverage spaces. As a result, one database might report coverage items that are not in the other database. This section lists common scenarios where this mismatch can occur. Unless otherwise noted, these reasons are applicable to all coverage types, including functional.

Given the same design unit, following are the reasons for a coverage item to appear in a simulation database, but not in an emulation database:

- The emulator functionality does not match the latest simulation feature set.
- The emulator cannot support a certain construct because it cannot be synthesized.
- The coverage instrumentation does not synthesize well.
- The coverage instrumentation would be too expensive in terms of emulator capacity.
- The functional coverage item has a tool-generated label.
- You have specified a design unit to not be covered.

For the most part, the converse also applies. Given the same design unit, following are the reasons for a coverage item to appear in an emulation database, but not a simulation database:

- The functional coverage item has a tool-generated label.
- The simulator has optimized a coverage item.
- You have specified a design unit or a specific coverage item to not be covered.

© 2021 All Rights Reserved.

Toggle Coverage Support with Palladium Z1

Toggle coverage is used to track the changes in the value of a signal from `0` to `1` or from `1` to `0`.

- For toggle coverage, the following terms have a specific meaning:
  - Track - indicates that HDL variable will be monitored at run time and included in the coverage database and coverage report.
  - Scored - is the act of marking a coverage item as being covered.

Toggle coverage is only tracked for the following:

- Verilog (scalar and vector) nets, registers, and ports

  Note: Aggregate variables, such as unpacked arrays, are not supported and will not be tracked.

- VHDL boolean (and vector versions)
- VHDL `std_ulogic`, `std_logic` (and vector versions), records

  Note: Arrays of VHDL records are not supported and will not be tracked.

- SystemVerilog struct data type

  Structural elements that would not be included in toggle coverage, that is the integer data type are not included in the toggle coverage report for a structure.

  Arrays of packed SystemVerilog structures are tracked if the `+tcovMDA` option is used and the array fits within the required bounds specified by the `+tcovMDA` option.

The following items are not scored during toggle coverage on Palladium Z1, and no messages are issued for these items:

- VHDL bit and vector versions

Note: Xcelium supports toggle coverage of the above two types.

- Variables that are not at the module scope (generates, named blocks, and so on)
- Real numbers or integers
- supply0 and supply1 nets
- SystemVerilog-specific data types that do not directly map into the supported subset mentioned above
- SystemVerilog aggregate types - union etc.

© 2021 All Rights Reserved.

- SystemVerilog package variables
- SystemVerilog interfaces
- Items within `` `qtrefcells `` primitives

For a given simulation run, the number of counter bits is fixed to one for code coverage items. In other words, if a signal toggles from 0 to 1 seven times in a given simulation, the actual count will reflect only one toggle. For toggle coverage, there are three one-bit counters: `rise`, `fall`, and `full`. Note that this is the default behavior of the Xcelium tool. But Xcelium enables users to specify an option to increase the maximum hit count.

Differences in Toggle Coverage Support on Palladium Z1 with Respect to Xcelium

Certain differences exist in coverage because Xcelium, which is an event-based simulator and the Palladium Z1, which is a cycle-based emulator, do not behave in a similar fashion. Following are the scenarios where Palladium Z1 coverage results might differ from Xcelium.

- **Scenario 1: Transitions to X or Z Values can Result in a Toggle on Palladium Z1 Emulator**

  It is a good practice to resolve all the `X` and `Z` values before swapping to the emulator. The `xc` command set enables you to specify how X and Z values should be handled (`xc xt0`, `xc zt0`, `xt1`, `zt1`). The conversion from `X` and `Z` values to a constant `1` or `0` does not cause a toggle at swap time, but it does result in a toggle the next time that the signal changes. Take the following waveform as an example:

  Figure 20-3 Waveform Depicting Toggles as a Result of Transitions to X or Z Values

  In the above waveform, the top `en` signal represents a simulation run where `en` remains un-initialized in a simulation environment. At time 20ns, `en` is driven to a `1`. In a software simulator, the transition at time 20ns does not count as a rising toggle. The bottom `en` signal represents a Palladium Z1 run where the design is swapped into hardware at time 10ns. When the design is swapped to the hardware mode, all `X` values that you specify are converted to `0` on the emulator. As a result, the transition at time 20ns is seen as a

© 2021 All Rights Reserved.

rising toggle on Palladium Z1. To avoid this behavior, try to delay the swap to the hardware (emulator) until all the `X` and `Z` values have been resolved.

This situation might also happen when a bus signal is driven to an `X` or `Z` value in pure simulation due to a multi-driver bus contention. On the emulator, the multi-bus-driver conflict does not resolve the value to `X`.

Note: Avoid using the Xcelium configuration switches set like `set_toggle_includex|z`, to get Xcelium to match Palladium Z1. These switches are not intended to match Xcelium runs with Palladium Z1.

- **Scenario 2: Event-based Glitches Result in a Toggle in Simulation**

  Due to the cyclic nature of the Palladium Z1 hardware, 0-delay glitches do not result in a toggle. In an event-based simulator, these glitches are treated as toggles.

  Note: Some features of Palladium Z1 enable certain logic to run in a more behavioral fashion. When these scenarios occur, glitches are counted as a toggle. Common examples of this include enabling the behavioral compiler, driving the design through DPI calls, and so on. When these features are used, Palladium Z1 does not regard anything as being toggled if it was not toggled in the simulator. In other words, these will not result in false positives.

  Xcelium provides a configuration option called `set_glitch_strobe` that can be used to define a glitch window where toggles should be ignored. If glitching is a problem in your simulation runs, use this option to get a more Palladium-Z1-like behavior. However, due to the different nature of these engines, this option cannot be used to guarantee an exact match with hardware.

Selective-Toggle Coverage

Selective-toggle coverage refers to enabling toggle coverage on a specific set of signal instances within a specific instance. Selective-toggle coverage is supported with all of the versions of Xcelium and MDV that are supported with the current version of VXE release.

Specify the signals to be monitored and tracked for toggle coverage in a file in XML format and specify the name of this file that includes the selective-coverage entries to IXCOM with the `-covSelectItemFile` <*file*> option of the `ixcom` command. Using this option also disables toggle coverage for all of the signals that are not specified in this file. Refer to The Format of the Selective-Toggle Coverage XML File for details of the XML format.

You can specify the `-covSelectItemFile` <*file*> option multiple times with all the entries being concatenated together as if it were one file. If the target file does not exist, an error is issued and the compilation process exits with a non-zero value.

© 2021 All Rights Reserved.

Note: Use the `-featureFile` option to add the module-level groupings of toggles so that both mechanisms can co-exist.

Considerations for the Signals to Be Tracked

To enable tracking of the signals via selective coverage, you do not need to specify other coverage options like `+tcovTypes`, `+tcovMDA`, and `+tcovStructs`, and so on. Selective-toggle coverage helps track toggle coverage on specific signals and enhances performance instead of enabling tracking for the entire design unit.

The restrictions on tracking of toggle coverage also apply to selective-toggle coverage. Selective coverage cannot be used to get IXCOM to track signals that would not be tracked via other means. For example, selective coverage cannot be used to track unpacked data types, nor can it be used to track signals within a netlist region.

IXCOM ignores duplicate entries for a selected signal, even if tracking was enabled by another mechanism. Therefore, if signal `a` in module `m` is to be tracked because of the `-featureFile` option setting, having an instance of `a` in the select item file is considered to be a duplicate because signal `a` will already be tracked.

Note the following considerations to be kept in mind for selective-toggle coverage:

- **VHDL is not supported:** Only Verilog and SystemVerilog signals can be tracked via selective toggle. If there are any references to VHDL signals, IXCOM generates a warning while processing the XML file.
- **Escaped identifiers are not supported:** Hierarchy and signals that have escaped identifiers in their names are not supported. If there are any references to such signals, IXCOM generates a warning while processing the XML file.
- **Compile-Time cost of using the `-covSelectItemFile` option:** Keep the number of signals to be tracked at a minimum. Reserve using selective toggle coverage for scenarios where less than 10% of the design signals are being tracked. To track more signals, use the `-featureFile` option to enable coverage.
- **Capacity implications of using the `-covSelectItemFile` option:** Toggle coverage instrumentation is optimized when large portions of the design are enabled. So, instead of selectively enabling every signal in the design (which is not recommended), use coverage toggle. If you selectively enable 10% of the design signals, the capacity is higher than 10% of the coverage toggle scenario capacity. The effectiveness of the optimizations depends on the signals that are being tracked.
- **Type-based coverage counters are not recorded:** Type-based coverage counters are not recorded for signals being tracked by selective toggle coverage since selective toggle coverage is instance-based. When a type report is generated on a database

© 2021 All Rights Reserved.

containing selected toggles, the signals that were enabled only by selective toggle coverage will not be present. This means that when the selective toggle coverage database is merged with full-design database, the selective toggle coverage will only contribute to the instance-based coverage.

- **Non-tracked signals in the coverage report:** The Unicov database requires that all bit values of a signal should be represented in the database. Starting with Xcelium 19.08.a001 and MDV 19.08.a001, Palladium coverage data treats such non-tracked bits as placeholders that appear in the coverage report but do not have associated counters. Such non-tracked-bit positions also do not contribute to the overall coverage calculation. In the older versions of Xcelium and MDV, the non-tracked bits are reported as zeros and contribute to the overall coverage count. You are strongly encouraged to migrate to the newer Xcelium and MDV releases to work with selective toggle coverage.

Option Precedence while Enabling Selective-Toggle Coverage

Selective-toggle coverage can be used with other options that enable or disable toggle coverage. Use the set of options to ensure tracking of required signals and avoiding the tracking of immaterial signals. The following table describes the precedence order for enabling or disabling toggle signals while compiling the design with IXCOM. The precedence order is important for gaining maximum control over enabling and disabling the tracking of the signals. Some examples are provided in the Examples of the Impact of Precedence on Tracking Signals section.

Option	Associativity	Precedence
`set_libcell_scoring via -covfile`	**None (Union)**	**Highest**
`-covSelectItemFile` <*file*>	**See below**	
`-featureFile` <*file*> `-covdut` <*design_unit*>	**Left**	**to**
`-coverage` <*coverage_type*>	**None (Union)**	**Lowest**

Associativity does not apply to the options marked as **None** because these only add coverage. The `-featureFile` and `-covdut` options are applied from left to right as these are parsed on the IXCOM command line. This complicates the specification of `enable` and `disable` regions for coverage.

The `-covSelectItemFile` option is given a higher precedence than `-featureFile` in the above table, which is true for adding elements. The `-covSelectItemFile` option inserts

© 2021 All Rights Reserved.

an implicit `-featureFile` setting to disable toggle coverage collection across the design. This implicit addition follows the precedence rules of the `-featurefile` option.

Keep the following scenarios in mind:

- When `-covSelectItemFile` is used and `-featureFile` (with toggle enables) is not used, only the signals in the selected item file are tracked.
- When `-covSelectItemFile` is used and the`-featureFile` (with toggle enables) option is used, specify the `-featureFile` option after the first `-covSelectItemFile` option with the `ixcom` command. In this scenario, the `-featureFile` setting cannot override the `-covSelectItemFile` settings because the `-covSelectItemFile` has higher precedence.

Note: The precedence table is related only to enabling and disabling signals. The `-coverage` option does more than just enable the hierarchy for coverage. It also specifies that a toggle-coverage database should be generated at run time. If you do not specify the `-coverage toggle`, `-coverage t`, or `-coverage all` option, then the `-covSelectItemFile`, `-featureFile`, and `-covdut` options do not generate toggle data, and the toggle-related functionality associated with these options is ignored. This is to provide you a single control point to enable or disable coverage generation without having to modify the rest of the options of the `ixcom` command.

Using the `-covSelectItemFile` and `-featureFile` options together allows you to enable toggle coverage on design units (via the `-featureFile` option) and on individual signals (via the `-covSelectItemFile` option) during the same compilation process. It is important to understand the precedence rules to ensure the desired effect. Only feature files with toggle entries after the `-covSelectItemFile` option are honored.

Using set_libcell_scoring Through covfile

The `set_libcell_scoring covfile` command is equivalent to the Xcelium `covfile` command. It is used to enable coverage scoring on cell modules. If this command is not specified, the default behavior is to not track coverage on Verilog modules that:

- Were specified within a file passed with the `-v` option
- Were specified within library directory passed with the `-y` option
- Have the `` `celldefine `` compiler directive

When the `set_libcell_scoring` command is specified, coverage is collected on the above modules. The `set_libcell_scoring` command supports the `-enable_vy` option, which provides coverage on the specified modules with the `-v` or `-y` option, but not with the `` `celldefine `` compiler directive.

© 2021 All Rights Reserved.

For more details on this command, refer to the *Integrated Coverage User Guide* included in the *Xcelium documentation set*.

Examples of the Impact of Precedence on Tracking Signals

This section illustrates examples for explaining the precedence order on tracking signals.

1. The following example does not involve selective toggle and enables toggle coverage on the entire design. The `-coverage` option is of the lowest precedence. However, in this case, it applies to the entire design because no other options are specified.

   ```
   ixcom -coverage t ...
   ```

2. The following example includes the `-featureFile` option is used in addition to the `-coverage t` option. The `-coverage t` option enables toggle coverage processing, and it enables toggle coverage across all of the modules in the design. The `-featureFile` option has higher precedence. So, it can be used to disable (and re-enable) areas of the design.

   ```
   ixcom -coverage t -featureFile disable.ff ...
   ```

   The ordering of these options does not matter. This example is equivalent to the following:

   ```
   ixcom -featureFile disable.ff -coverage t ...
   ```

3. The following example illustrates selective toggle coverage. It includes the `-covSelectItemFile` option in a manner that is similar to the usage of the `-covSelectItemFile` option.

   ```
   ixcom -coverage t -covSelectItemFile togs.xml ...
   ```

   In this example also, the order of the above arguments does not matter. Coverage is enabled across the entire design via the `-coverage t` option. The `-covSelectitemFile` option disables general collection and limits the tracking to the signals specified in the `togs.xml` file.

4. The following example shows how selective toggle and feature file can be used in conjunction.

   ```
   ixcom -coverage t -covSelectItemFile togs.xml -featureFile disable.ff ...
   ```

   In this example, the order of arguments is **important**. Coverage is enabled across the entire design via the `-coverage t` option. The `-covSelectItemFile` option automatically disables toggle coverage across the design and only enables it for the signals listed in the XML file. The `-featureFile` option re-enables toggle coverage in the situation that require module-level control. However, this is lower priority, and the `-featureFile` cannot disable signals that are tracked via the `-covSelectItemFile` option.

© 2021 All Rights Reserved.

Note that the following example is NOT equivalent to the above example:

```
ixcom -coverage t -featureFile disable.ff -covSelectItemFile togs.xml ...
```

For this example, the `-featureFile` option is processed before the `-covSelectItemFile` option. This replaces the toggle-coverage settings of the `-featureFile` option with the settings of the `-covSelectItemFile` option.

The Format of the Selective-Toggle Coverage XML File

The XML format is used to contain the signals to be tracked for toggle coverage. Specify the signals fully so that the entire hierarchy is present. If the `+xcDesignTop` option is used, the hierarchy can include the testbench hierarchy; but it is not required.

XML DTD (Document Type Declaration) is a standard way to describe the structure associated with an XML file. The DTD specifies every element, attribute and format for values in a compact fashion.

```
<?xml version="1.0"?>
<!DOCTYPE coverage-select-root [

<!ELEMENT coverage-select-root (rules)>
<!ELEMENT rules (rule)*>
<!ELEMENT rule (#PCDATA)>

<!ATTLIST rule ccType (inst) #REQUIRED>
<!ATTLIST rule entityName CDATA #REQUIRED>
<!ATTLIST rule name (include) #REQUIRED>
<!ATTLIST rule entityType (toggle) #REQUIRED>
]>
```

Following are the descriptions of the XML attributes listed above. Incorrect XML results in an error message and a non-zero exit status from IXCOM. The XML parsing errors specify the file name and line number.

XML Attribute	Definition
`ccType` XML Attribute	The `ccType` specifies the kind of coverage that the signal entry represents. Here, `inst` indicates that the signal entry is a full instance name. No other values are supported.

© 2021 All Rights Reserved.

XML Attribute	Definition
`entityName` XML Attribute	The `entityName` contains a legal Verilog or SystemVerilog full instance name (also called XMR) of the signal to be tracked for toggle coverage. Following are the warning scenarios associated with the `entityName` attribute that result in the specified XML element being discarded. ■ Unsupported entityName Scenario ■ Hierarchical path could not be resolved ■ Signal reference is not found ■ Reference is to something other than a signal ■ Signal is not Verilog or SystemVerilog ■ Part select such as top.M1.bus[1:4] ■ Escaped identifier (anywhere in reference) ■ Bit select is out of range
`name` XML Attribute	The `name` attribute indicates the action to be performed on the specified element. Here, `include` indicates that the item should be included for toggle coverage. No other values are supported.
`entityType` XML Attribute	The `entityType` attribute indicates the kind of coverage is associated with the specified element. Here, `toggle` indicates toggle coverage. No other values are supported.

XML Example

The following table describes some sample signals to monitor via selective-toggle coverage. Using the complete signal name on aggregate data types results in all bits of that aggregate

© 2021 All Rights Reserved.

type being tracked. Specifying signals via bit select is supported. Specifying signals via partial selections, such as `bus[3:2]`, is not supported and these are ignored.

XMR	Leaf Module Name	Associated Variable Declaration	Comments
`top.DUT.M0.clk`	`middle`	`input wire clk`	Scalar clock pin in `M0`
`top.DUT.M1.dout[1]`	`middle`	`reg [3:0] dout;`	Bit select of a vector in `M1`
`top.DUT.M1.L.bus`	`lower`	reg [3:0] bus;	Monitor complete vector
`top.DUT.M1.myStruct.opcode`	`dut`	`typedef struct packed { bit [1:0] opcode; bit [7:0] addr; } instruction; instruction myStruct;`	Equivalent to monitoring: `top.DUT.M1.myStruct.opcode[1]` `top.DUT.M1.myStruct.opcode[0]`
`top.DUT.M0.myStruct[8]`	`dut`	`typedef struct packed { bit [1:0] opcode; bit [7:0] addr; } instruction; instruction myStruct;`	Equivalent to monitoring: `top.DUT.M0.myStruct.opcode[0]`
`top.DUT.M0.myMDA[1][2]`	`dut`	`reg [1:0][3:2] myMDA`	Multidimensional bit select
`top.DUT.M1.myMDA[1]`	`dut`	`reg [1:0][3:2] myMDA`	Equivalent to monitoring: `top.DUT.M1.myMDA[1][3]` `top.DUT.M1.myMDA[1][2]`

© 2021 All Rights Reserved.

Following is the XML file for the items referenced in the above table.

```
<?xml version="1.0"?>
<coverage-select-root>
<rules>
<rule ccType="inst"
      entityName="top.DUT.M0.clk"
      entityType="toggle"
      name="include"/>
<rule ccType="inst"
      entityName="top.DUT.M1.dout[1]"
      entityType="toggle"
      name="include" />
<rule ccType="inst"
      entityName="top.DUT.M1.L.bus"
      entityType="toggle"
      name="include" />
<rule ccType="inst"
      entityName="top.DUT.M1.myStruct.opcode"
      entityType="toggle"
      name="include"/>
<rule ccType="inst"
      entityName="top.DUT.M0.myStruct[8]"
      entityType="toggle"
      name="include"/>
<rule ccType="inst"
      entityName="top.DUT.M0.myMDA[1][2]"
      entityType="toggle"
      name="include"/>
<rule ccType="inst"
      entityName="top.DUT.M1.myMDA[1]"
      entityType="toggle"
      name="include"/>
</rules>
</coverage-select-root>
```

Block Coverage Support with Palladium Z1

Block coverage is used to track the HDL blocks in the code that have been executed and the ones that have not, and it identifies whether or not the test scenarios exercise the statements in a block. A block is a statement or sequence of statements in Verilog or VHDL that executes with no branches or delays. Either none or all of the statements in a block are executed.

This section describes a mechanism to gather and report block coverage statistics using Palladium Z1.

Block coverage on Palladium Z1 emulator supports only sequential blocks for coverage in both Verilog and VHDL. A block is considered sequential if it falls into one of the following categories:

- A process with a single edge-sensitive clock event.

© 2021 All Rights Reserved.

- A process with multiple clock events, where one event is the true clocking event and all other events are asynchronous reset events.

  Note: All clocking events must be edge sensitive.

The blocks or processes that do not meet these criteria are ignored for block coverage without warning.

Specifically, on Palladium Z1, block coverage is tracked only for the following:

- Verilog `always` blocks (sequential only)
- Verilog `initial` blocks
- VHDL `process` and `generate` statements
- Verilog and VHDL processes within `generate` `if`, `for`, and `case` blocks
- Verilog and VHDL explicit case default

The following items are not scored during block coverage on Palladium Z1, and no messages are issued for these items:

- SystemVerilog final blocks (Same as Xcelium.)
- Non-sequential VHDL processes (Supported by Xcelium.)
- Verilog tasks and functions (Supported by Xcelium.)
- Statements in a TBcall region (Not applicable by Xcelium.)
- VHDL concurrent signal assignments (Same as Xcelium.)
- Functions and procedures in VHDL subprograms inside packages (Supported by Xcelium with certain limitations.)
- Event constructs `@`, `or`, and event (Supported by Xcelium with the `-coverage` block option.)
- Gate-level or user-defined primitives (Same as Xcelium.)
- Protected Verilog and VHDL statements (Same as Xcelium.)
- Verilog/VHDL implicit else (Supported by Xcelium with an option.)
- Verilog implicit case default (Supported by Xcelium with an option.)
- Verilog continuous assignments as blocks (Supported by Xcelium with an option.)
- Verilog ternary assignments and VHDL conditional and selected signal assignments (Supported by Xcelium with an option.)

© 2021 All Rights Reserved.

- Verilog Programs (Supported by Xcelium.)
- SystemVerilog interfaces (Supported by Xcelium.)
- Initial blocks used to specify design clocks (Supported by Xcelium.)

For a given simulation run, the number of counter bits is fixed to one for code coverage items. In other words, if a signal toggles from 0 to 1 seven times in a given simulation, the actual count will reflect only one toggle. For block coverage, each block is represented by a one-bit counter that is set when the block is executed. Note that this is the default behavior of the Xcelium tool. But Xcelium enables you to specify an option to increase the maximum hit count.

Expression Coverage Support with Palladium Z1

Expression coverage is enabled with the `-coverage` option of the `ixcom` command, provided the `-ua` option is specified. Alternatively, you can use the `-featureFile` option for enabling expression coverage by using the `coverage_expression` term.

Following is an example that enables expression coverage only on the `dut` module.

```
coverage_expression disable
coverage_expression enable du=dut
```

All `control_option` and `configuration_option_opt` values are supported with expression coverage. Refer to the Enabling and Disabling Features at Compilation Time section for details on the `-featureFile` option.

The `+covGenFeatureFile` option disables expression coverage logic in addition to toggle and block coverage.

Cadence Xcelium simulator supports three variants of expression coverage:

- FCC - Full Conditional Coverage
  - Not recommended in Xcelium because of the heavy impact on simulation performance.
  - Not supported on Palladium.
- SoP - Sum of Products Coverage
  - Default for Xcelium.
  - Not supported on Palladium. On Palladium, use Control Scoring instead of SoP.
- Control Scoring

© 2021 All Rights Reserved.

- Most widely used and recommended on Xcelium.
- On Palladium, Control Scoring is supported for Verilog, and SystemVerilog, and VHDL.

Note: Control scoring checks if each input has controlled the output value of the expression at some time during the simulation. If the value of input changes, the value of output also changes. Control scoring improves verification accuracy by applying stronger requirements in order to call an expression input covered. The control-scoring mode follows the following rule:
Coverage is scored if and only if an expression term has at least one input (or operand) that controls the result of the expression.
Two or more inputs of an expression might together control the result of an expression.

Tracking of Condition Statements in Verilog and SystemVerilog Designs

As with the simulator, the following condition statements are candidates for being tracked for expression coverage on Palladium:

- The `if`, `if...else`, and `case` within supported processes.

  Note: `casex` and `casez` are not supported.

- Ternary operators within continuous assignments, procedural assignments, and procedural continuous assignment statements (regardless of the use of the `-contassign` or `-procassign` option). The `-contassign` is an option within a file that is passed to`-covfile`.
- Continuous assignment statements (when `-contassign` and `-covfile` options are used).
- Procedural assignment and procedural continuous statements (when the `-procassign -covfile` options are used).
- The `-procassign` option enables scoring of sequential statements in VHDL. The `-contassign` option enables concurrent assignment in VHDL.

Note: The `for`, `repeat`, `do..while`, and `while` looping constructs are not supported. The looping constructs are, typically, non-synthesizable except for the most basic scenarios (which would yield 100% coverage).

By default, only `&&` and `||` operators are tracked. The rules for tracking operators follow the simulation rules defined in the *ICC User's Guide*. Expanding support beyond `&&` and `||` can be done by using the `set_expr_coverable_operators` command in the `covfile`.

© 2021 All Rights Reserved.

Only instance-based coverage is collected. Type-based expression coverage data is not captured.

Refer to the Supported CCF Options for the DUT section for the list of supported options for `set_expr_coverable_statements` command in the file passed with the `-covfile` option.

Supported Processes

The `initial` and synthesizable `always` processes are supported. Conditions within behavioral processes are not tracked for expression coverage. The SystemVerilog `final` processes are not supported.

Conditions within a procedural `assign` statement are tracked if the process meets the overall support criteria.

Unsupported Condition Statements

The following condition statements scenarios are ignored and not tracked:

- Conditions within SystemVerilog tasks and functions
- Conditions within a TBcall region
- Conditions within a SystemVerilog final block
- Conditions within protected HDL regions
- Conditions within SystemVerilog programs
- Conditions within SystemVerilog interfaces

Expression coverage is not tracked if one of the following conditions exists within the condition:

- A function with side effects
- Unary increment operators (`++`, `--`)

Functions with Side Effects

In SystemVerilog, a function is considered to have side effects if the function modifies variables in the design. Such functions are not tracked.

© 2021 All Rights Reserved.

Non-Supported Clocking or Reset Conditions

Expression coverage is not tracked for conditions associated with clock and reset regions, such as the following:

```
always @(posedge clk or posedge rst)
            if(rst == 1'b1)            // Not tracked, reset region
            dout <= 1'b0;
            else                       // Start of RTL region
            if(sync_rst == 1'b1)       // Tracked if '==' enabled
                 dout <= 1'b0;
            else
                 dout <= din;
```

Tracking of Condition Statements in VHDL

The following VHDL condition statements are candidates for being tracked for expression coverage on Palladium:

- Expression of `if` and `elseif` conditions in `if` sequential statements
- Expression of `case` expression of `case` sequential statements

Conditional signal assignments and selected signal assignments are equivalent to sequential `if` and `case` assignments, respectively, as defined by the VHDL language. The expressions in these equivalent statements are also scoring candidates.

Supported Operators in VHDL

Use `set_expr_coverable_operators -all` for all supported operators.

The following operators have been identified for scoring in a given expression:

- Six relational operators:

  `=`, `/=`, `<`, `>`, `<=`, and `>=`

  Use `set_expr_coverable_operators -relational` for the relational operators.

- Six logical or bitwise operators:

  `AND`, `OR`, `NAND`, `NOR`, `XOR`, and `XNOR`

  For expression coverage on these operators, consider the following:

  - `AND`, `OR`, `NAND`, and `NOR` are supported by default.
  - Use `set_expr_coverable_operators -bitwise` for the bit-wise operators: `XOR` and `XNOR`.

© 2021 All Rights Reserved.

By default, Xcelium performs short-circuit evaluation and accordingly generates the control-coverage table of such evaluation. So, some operands values are marked as - (dash) to imply `Don't Care`.

To override this default behavior in Xcelium and generate the control-coverage table without `Don't Care`, use the following `covFile` (CCF) command:

```
set_expr_scoring -control -no_vhdl_shortcircuit
```

For emulators, there is no short-circuit evaluation, the control-coverage table is not generated with `Don't Care`. The presence or absence of the `-no_vhdl_shortcircuit` option in the `set_expr_scoring` CCF command does not make any difference.

Supported Processes in VHDL

Sequential processes and concurrent assignments are supported.

Non-Supported Scenarios in VHDL

The following condition statements scenarios are ignored and not tracked:

- Conditions inside VHDL functions and tasks.
- Conditions within `while` loops and `for` statements.
- Conditions within protected HDL regions
- Conditions within while loops

Only conditions in RTL region of asynchronous reset blocks are covered.

Clock expressions are ignored because:

- The instrumentation needed to track the expressions lies outside the *RTL region* in the case of asynchronous-reset scenarios. So, the generated instrumentation is *non-synthesizable*.
- In clock expressions, such as `clk‘EVENT and (clk = '1')` even if it is not an asynchronous scenario:
  - `clk‘EVENT` cannot be used in creating instrumented expressions.
  - Though `clk = '1'` can be instrumented, it is not needed in hardware and it reflects the correct value because, in hardware, generally, clock paths are tuned for faster transfers than data paths.

  Note: Expressions within the RTL region being driven by the clock expressions are

© 2021 All Rights Reserved.

covered.

Differences between Emulation and Simulation for Expression Coverage

The expression results reported by Palladium are based on the behavior of the emulator. Running the same test in an event-based simulator as compared to a cycle-based emulator can yield different results because the models are executed differently. Following are some common differences between event-based and cycle-based simulators:

- X and Z support in the simulator is not available in emulation.
- Event scheduling in the simulator is different from cycle-based evaluation in the emulator.
- Combinational logic is always evaluated with every evaluation in the emulator. In the simulator, combinational logic is only executed when the inputs change.
- There may be delays associated with combinational logic propagation in the simulator that creates glitches. These glitches can contribute to coverage differences between simulation and emulation.

If you are unsure of coverage results, it is recommended that you compare the Palladium results against a waveform generated from the same Palladium run instead of comparing against a simulation run.

Simulation Options and their Equivalent Palladium Options

The default expression-coverage configuration for the Cadence simulator is different than the default configuration for Palladium. The following CCF options can be used if you want pure simulation runs to generate expression coverage in a similar fashion to Palladium.

```
set_expr_scoring -control
set_expr_scoring -struct
```

By default, the Palladium emulator sets scoring type to control scoring. By default, Xcelium discards any conditions that contain a structural reference. You can enable these in Xcelium by using the `-struct` option.

Merging of IMC Databases

IMC merges expression coverage results when all databases involved with the merge are generated by Palladium. IMC does not support cross-engine merging of expression data. Performing a cross-engine merge on a Palladium database with expression data results in the expression data not surviving the merge.

© 2021 All Rights Reserved.

Expected Capacity Impact

The overhead for control scoring expression coverage in emulation with default operators is an additional 30-40%. This is equivalent to block coverage. The emulation resources are likely to be higher because non-default values are used to expand the expression coverage space. The actual number is based on the coding style used in the design being compiled. This overhead can be reduced by limiting the coverage area using the `-featureFile` option. Enabling additional expression or statement types adds additional capacity.

Using the -covfile <file> Option with Expression Coverage

In previous releases, the file specified with the `-covfile` option was *as-is* passed to the to the simulator. Starting with the VXE 18.1 release, IXCOM parses the Coverage Configuration File (CCF) Commands listed in the file specified with the `-covfile` option.

Beyond basic parsing, the CCF commands passed to IXCOM are partitioned into:

- The simulation subset, which is passed to the simulator for testbench-related coverage (Supported CCF Options for the Testbench)
- The supported DUT subset, which is acted upon by the emulator (Supported CCF Options for the DUT)
- The unsupported emulation subset for which a fatal error message or a warning is issued (Unsupported CCF Options Generating Fatal Error Messages and Unsupported CCF Options Generating Warnings)

Note: If a specified option is supported by both the simulation subset and the emulation subset, that option appears in both subsets.

© 2021 All Rights Reserved.

If you are using abstract IXCOM compilation flows, such as the `+xcDesignTop` option, for which the simulation layer is not visible to IXCOM, you need to manually partition the design and pass the CCF to IXCOM. Typically, you can pass the CCF as-is to IXCOM. However, you must remove those options that are intended for the testbench portion because these are not visible to IXCOM and might cause IXCOM to generate errors.

Specifying `all` and `modules` Arguments for CCF Options

Typically, the CCF options that have the `all` argument are supported. For example, using `set_expr_scoring -all` has no effect on the simulator because expression coverage is not supported in the testbench and only enables expression types supported on the emulator. An informational message is issued when the `all` argument is used and the resulting behavior might not match the pure simulation behavior. This applies to any option where `all` is partially supported.

CCF options that take an optional `modules` argument to specify the module for which the option is applicable are not supported. A fatal error message is issued if a module name is specified in conjunction with a CCF option.

Supported CCF Options for the Testbench

The following CCF options and arguments are supported and passed to the simulator. These are applied to the modules that are controlled by the simulator, that is by the testbench. The **Warning/Notes Generated** column indicates that a warning is issued stating that the option is supported only in the testbench partition.

CCF Option	In Testbench, Applies To...	Warning/Notes Generated
`include_ccf`	All	No
`set_covergroup -optimize_dump`	Covergroup	No warning is generated because this option is always enabled for the DUT partition controlled by IXCOM.
`set_covergroup -show_uninstantiated`	Covergroup	Yes
`set_covergroup -new_instance_reporting`	Covergroup	Yes

© 2021 All Rights Reserved.

CCF Option	In Testbench, Applies To...	Warning/Notes Generated
`set_covergroup -default_cross_num_print_missing`	Covergroup	Yes
`set_covergroup -default_goal`	Covergroup	Yes
`set_covergroup -default_type_option_goal`	Covergroup	Yes
`set_covergroup -merge_instances`	Covergroup	Yes
`set_covergroup -cross_auto_bin_max_default_zero`	Covergroup	Yes
`set_covergroup -per_instance_default_one`	Covergroup	Yes
`set_covergroup -detect_overlap`	Covergroup	Yes
`set_covergroup -bin_merge`	Covergroup	Yes
`set_covergroup -merge_cp_by_name`	Covergroup	Yes
`set_covergroup -svcg_expand_trans`	Covergroup	No warning is generated because this option is always enabled for the DUT partition controlled by IXCOM. Xcelium has deprecated this option and its use might generate errors during Xcelium elaboration stage.
`set_parameterized_module_coverage`	All	Yes When specified on a testbench module, `set_parameterized_module_coverage` derives the information from the instrumented name. Therefore, using this option is not recommended.

© 2021 All Rights Reserved.

Supported CCF Options for the DUT

The following CCF options and arguments are supported and passed to the emulator. These are applied to the modules that are controlled by the emulator, or by the DUT.

CCF Option	In DUT, Applies to...	Notes
`include_ccf`	All	
`set_expr_scoring -all`	expr	
`set_expr_scoring -control`	expr	
`set_expr_scoring -struct`	expr	Option ignored because it is always enabled in IXCOM.
`set_expr_scoring -max_expr_terms`	expr	IXCOM issues a warning message at compilation time if an expression exceeds `max_expr_terms` and is not scored. By default, the number of expression coverage items is set to a maximum of `256`. If required, this setting can be revised to `16`,`32`,`64`,`128`, `512`,`1024`, or `2048`.
`set_expr_coverable_operators -all`	expr	Warning that all excludes `event_or`
`set_expr_coverable_operators -bitwise`	expr	
`set_expr_coverable_operators -relational`	expr	

© 2021 All Rights Reserved.

CCF Option	In DUT, Applies to...	Notes
`set_expr_coverable_operators -conditional`	expr	
`set_expr_coverable_operators -reduction`	expr	
`set_expr_coverable_operators -logical_not`	expr	
`set_expr_coverable_statements -all`		expr
`set_expr_coverable_statements -procassign`		expr
`set_expr_coverable_statements -contassign`		expr
`set_toggle_noports`	toggle	

Unsupported CCF Options Generating Fatal Error Messages

The following CCF options and arguments are not supported and a fatal error message is generated if these are used.

CCF Option	Applies To...
`(de)select_coverage <metric> -module`	all
`(de)select_coverage <metric> -class`	all
`(de)select_coverage <metric> -cg_name`	all
`(de)select_coverage <metric> -instance`	all
`(de)select_coverage <metric> -file`	all
`(de)select_coverage <metric> -filelist`	all
`(de)select_coverage <metric> -sysv_bind_modules`	all
`set_implicit_block_scoring`	block
`set_explicit_block_scoring`	block
`set_assign_scoring`	block
`set_branch_scoring`	block
`set_statement_scoring`	block
`set_subprogram_scoring`	block, expr

© 2021 All Rights Reserved.

CCF Option	Applies To...
`set_expr_scoring -sop`	expr
`set_expr_scoring -vector`	expr
`set_expr_scoring -fcc`	expr
`set_expr_scoring -event`	expr
`set_expr_scoring -no_vhdl_shortcircuit`	expr
`set_expr_scoring -vhdl_noreduce_table`	expr
`set_expr_scoring -vhdl_not_as_operator`	expr
`set_expr_scoring -max_terms_fcc`	expr
`set_expr_scoring -max_terms_sop`	expr
`set_expr_coverable_operators -event_or`	expr
`set_expr_coverable_statements -event_control`	expr
`set_expr_coverable_statements -misc`	expr
`set_fsm_attribute`	FSM
`select_fsm`	FSM
`deselect_fsm`	FSM
`set_fsm_reset_scoring`	FSM
`set_fsm_arc_scoring`	FSM
`set_fsm_arc_termlimit`	FSM
`set_fsm_scoring`	FSM
`set_toggle_scoring -sv_enum`	toggle
`select_functional`	Functional
`set_libcell_scoring`	All
`set_merge_with_libname`	All

© 2021 All Rights Reserved.

Unsupported CCF Options Generating Warnings

The following CCF options are arguments are not supported and a warning is generated if these are used.

covvfile Option	Applies To...
`set_glitch_strobe`	block, expr
`set_hit_count_limit`	block, expr
`set_com`	block, expr, toggle
`set_com_interface`	block, expr, toggle
`set_code_fine_grained_merging`	block, expr
`set_refinement_resilience`	block, expr
`set_toggle_strobe`	toggle
`set_toggle_limit`	toggle
`set_toggle_includex`	toggle
`set_toggle_includez`	toggle
`set_toggle_excludefile`	toggle
`set_optimize -vlog_prune_on`	Assert
`set_optimize -ial_ovl_inst_asrt`	Assert
`set_optimize -top_expr_non_scorable`	Assert
`set_covergroup -show_uninstantiated`	Covergroup
`set_covergroup -new_instance_reporting`	Covergroup
`set_covergroup -default_cross_num_print_missing`	Covergroup
`set_covergroup -default_goal`	Covergroup
`set_covergroup -default_type_option_goal`	Covergroup
`set_covergroup -merge_instances`	Covergroup
`set_covergroup -cross_auto_bin_max_default_zero`	Covergroup
`set_covergroup -per_instance_default_one`	Covergroup
`set_covergroup -detect_overlap`	Covergroup
`set_covergroup -bin_merge`	Covergroup

© 2021 All Rights Reserved.

covvfile Option	Applies To...
`set_covergroup -merge_cp_by_name`	Covergroup
`set_libcell_scoring`	All
`set_parameterized_module_coverage`	All

Verilog and VHDL Process Restrictions in Code Coverage Support

Only a subset of processes is instrumented for coverage in Verilog and VHDL.

A Verilog process is instrumented only if it falls into one of the following categories:

- The process has a single clock event that is edge sensitive.
- The process has multiple clock events, where one event is the true clocking event and all other events are asynchronous reset events. All clocking events must be edge sensitive.
- The process has one or more clocking events that are all not edge sensitive. (ie. a combinational process)
- The process has @(*) as the clocking event

VHDL restrictions are similar to Verilog, except that combinatorial processes are not supported.

Note: Processes that do not meet these criteria are ignored for block coverage without any warning messages.

Differences in Code Coverage Support on Palladium Z1 with Respect to Xcelium

Certain differences exist in coverage support provided by Palladium Z1 and Xcelium because Xcelium, which is an event-based simulator and the Palladium Z1, which is a cycle-based emulator, do not behave in a similar fashion.

Xcelium supports block coverage of the following types:

- Verilog/VHDL implicit else
- Verilog implicit case default
- Verilog continuous assignments as blocks

© 2021 All Rights Reserved.

- Verilog ternary assignments and VHDL conditional and selected signal assignments
- Verilog and VHDL protected statements

Following are the scenarios where Palladium Z1 coverage results might differ from Xcelium.

Scenario 1: Combinational Processes

Combinational processes are evaluated differently in software simulation than they are in hardware. Consider the following example:

```
always @(a, b, sel)
begin           // block 0
if(sel
    d <= a;         // block 1
else
    d <= b;         // block 2
end
```

For block `0`:

- Simulation semantics call for the overall process to only be executed when one of the inputs have changed.

  Therefore, this block is only considered covered if one of the inputs change.

- Hardware semantics call for the overall process to be executed all the time.

  Therefore, this block is always considered covered in hardware.

For block `1`:

- Software semantics call for block `1` to be executed if `a`, `b` or `sel` change and `sel` is true.
- Hardware semantics call for block `1` to be executed whenever `sel` is true.

For block `2`:

The same rules for block `1` apply, but since `sel` defaults to `0` on hardware, the `else` block will be executed in hardware when it would not be executed in software.

Although the results might not match, the coverage from simulation and emulation is correct when compared to the behavior of the underlying engine.

Following are some examples:

- Fewer blocks being covered in emulation than would be in simulation due to glitches on inputs.

© 2021 All Rights Reserved.

- More blocks being covered in emulation than would be in simulation due to hardware always running processes every cycle. (This would be the case if the process never wakes up in software).

This behavior is common for simulation and emulation execution.

Scenario 2: Multiple Asynchronous Clocking Events on Palladium Z1

Logically speaking, the asynchronous clocking events are evaluated in a specific order. However, this order is ignored on Palladium Z1, when evaluating block coverage for asynchronous blocks. Consider the following example:

```
always @(posedge clk or posedge r0 or posedge r1)
begin
     if(r0)
         q <= 0;
     else if(r1)
         q <= 0;
     else
         q <= d;
end
```

If `r0` is set to `1'b1` for all time, the block associated with `r1` should logically never fire. On Palladium Z1, the block coverage for asynchronous regions is tied to the clocking events. Even if `r0 == 1'b1`, a `posedge r1` clocking event causes the block associated with `r1` to be considered as covered.

Scenario 3: Asynchronous Regions not Covered on Palladium Z1

Like the previous scenario, the block coverage for an asynchronous region is tied to the clocking event associated with that region. If `r0`, in the above example, is initialized to `1'b1` as in the following code, and the value remains unchanged, the block associated with the `posedge r0` clocking event is not marked as covered, even if other clocking events wake up the process:

```
reg r0 = 1'b1;
```


© 2021 All Rights Reserved.

Functional Coverage Support with Palladium Z1

Functional coverage requires you to specify the information that needs to be tracked. Palladium Z1 supports aspects of functional coverage through SystemVerilog and PSL. Functional coverage helps identify the non-tested functionality in the design, and provides a view of verification completeness from a functional point of view. Functional coverage can be performed on user-defined coverage points, specified using PSL and SystemVerilog assertions, or SystemVerilog covergroup statements. These coverage points specify scenarios, error cases, corner cases, and protocols to be covered and also specify analysis to be done on different values of a variable. By exercising the important areas of the design with the help of user-defined functional coverage points, you can optimize the testing effort.

The IXCOM compiler supports functional coverage using assertions and covergroups.

This section describes the mechanisms to gather and interact with the functional coverage statistics for a SystemVerilog design by using the covergroup construct of SystemVerilog. Covergroup support is based on the 2009 SystemVerilog LRM. All references to the SystemVerilog LRM in this chapter are specific to the 2009 version. For information on functional coverage support provided by IXCOM using assertions, refer to Functional Coverage Support on page 1215.

The terms related to functional coverage support provided by SystemVerilog are defined below. This chapter uses the following terms:

- A bin is a value or set of values that have a corresponding hit counter. The counter increments when these values are sampled.
- A coverpoint is a collection of bins that relate to a particular expression.
- A *cross* is a collection of bins that represent the cross product of the referred coverpoints.
- A *covergroup* is a collection of coverpoints, crosses, and the options that relate to them.

© 2021 All Rights Reserved.

The following entity-relationship diagram explains how the bin, coverpoint, cross, and covergroup relate to each other. For more information, refer to the *IEEE-1800-2009 SystemVerilog Language Reference Manual.*

Covergroup Support in IXCOM

The SystemVerilog functional coverage support for IXCOM is a subset of the covergroup support provided by the base Xcelium release. IXCOM supports functional coverage for SystemVerilog designs as well as mixed-language designs and references. IXCOM supports the use of covergroups in the LA mode.

The following sections summarize support information working from the inner-most level, the bin, to the outer-most level, the covergroup. Additionally, support information for where and how a covergroup can be instantiated has also been provided.

For more information about creating covergroups in a way to make these Palladium-compatible, refer to Quick Reference Guide for Covergroup Synthesis.

© 2021 All Rights Reserved.

Bin Support

The following table lists the different types of bins and support level provided by IXCOM. See the *Functional coverage* chapters of the *SystemVerilog LRM* and the Cadence document *ICC User Guide* in the Xcelium documentation for more information about bins.

Name	Description	Example	Supported
Scalar	Single bin for a set	`bins a = {5, 6};` `bins trana = (0 => 1);`	Yes
Vector	One bin per value in set	`bins b[] = {[0:9]};` `bins tranb[] = (0 => 1), (2 => 3);`	Yes
Fixed	Distribute set to fixed number of bins	`bins c[2] = {[0:9]};`	Yes
Automatic	All possible values are distributed to `auto_bin_max` number of bins	N/A, but implicitly equivalent to: `bins auto[auto_bin_max] = {[$:$]};`	Yes
Transition	Sequence drives the bin	`bins d = {1 => 2 => 3};`	Yes, see Considerations for Bins
Default	Convenient way to group all non-specified values	`bins e = default;`	Yes, see Considerations for Bins
Default Sequence	Convenient way to group all non-specified sequences	`bins f = default_sequence;`	No
Wildcard	Convenient way to wildcard bit positions of a specified value	`wildcard bins g = { 4'b11?? };` `wildcard bins t = (2'b0? => 2'b1?);`	Yes, see Considerations for Bins
Ignore	Mechanism to ignore values from coverage computation	`ignore_bins h = {5};`	Yes, state bins only.

© 2021 All Rights Reserved.

Name	Description	Example	Supported
Illegal	Mechanism to generate an error message if specified values are sampled during coverage sampling	`illegal_bins i = {5};`	Yes, state bins only.

Considerations for Bins

- The use of `$` as a range value is supported in the upper or lower bound of a range. Additionally, the non-standard usage of `[$:$]` is supported to indicate all valid values.
- Default bins are supported, but their counts are not tracked due to capacity reasons. The default bins, however, do not influence the coverage computation, according to the *SystemVerilog LRM*. The illegal default bins cause a simulation error.

  Note: Xcelium reports default bin counts as a separate item in the IMC report.
- Transition bins are supported except for:
  - Illegal_bins
  - Ignore_bins
  - Default_sequence
  - Wildcards with more than eight wildcard characters
  - When the transition sequences are too large and would use an excessive amount of capacity.

  For the above-mentioned exceptions, a warning is issued and the entire covergroup is discarded.
- Formal argument to specify bin array index is not supported.
- `iff` expressions specified at the bin level are supported for state, transition, and wildcard bins. These are not supported for ignore, illegal and default bins, or default sequences. If an unsupported bin `iff` is encountered, the entire covergroup is discarded. Each line of the bin can have its own guard.

  For example:

```
covergroup cg @ (posedge clk);
      A: coverpoint expr1 {
```


© 2021 All Rights Reserved.

```
        bins scalar1 = { [0:3] } iff (en[0]);
        bins scalar2 = { [2:5] } iff (en[1]);
        bins vector1[] = { [4:7] } iff (en[2]);
        bins vector2[] = { [8:9], 11 } iff (en[3]);
        bins fixed1[2] = { 10, [12:13] } iff (~en[0]);
        bins fixed2[2] = { [12:15] };
        bins trana = (0 => 1), (2 => 3) iff (en[0]);
        bins tranb[] = ( [0:1] => [1:2]) iff (en[1]);
    }
    B: coverpoint expr2 {
    wildcard bins b_01 = {16'b????????????????1?} iff (en[0]);
    wildcard bins b_02 = {16'b???????????????1??} iff (en[1]);
    wildcard bins b_03 = {16'b??????????????1???} iff (en[0]);
    wildcard bins b_04 = {16'b???????????????1????} iff (en[1]);
    wildcard bins b_05 = {16'b????????????1?????} iff (en[0]);
    wildcard bins b_06 = {16'b?????????1??????} iff (en[1]);
    wildcard bins b_07 = {16'b????????1???????} iff (en[0]);
    }
    endgroup // cg
```

- Error messages for illegal bins are not generated in LA mode.
- Wildcard bins are supported except for:
  - Vector or fixed bins partially supported. See Wildcards with Vector, Fixed, and Automatic Bins for details.
  - Transition bins with more than eight wildcard characters
  - The state bins that contain more than ten wildcard characters, and are part of a cross under some conditions.

    See Coverpoints Part of a Cross and Containing Wildcard Bins for more details.

Considerations for Coverpoints

- Coverpoints are supported on integral expressions, in compliance with the *SystemVerilog LRM*.
- You can explicitly state the name associated with a coverpoint. When a name is not explicitly supplied, the *SystemVerilog LRM* rules for naming coverpoints are used. When the LRM naming rules cannot be implemented, the name follows the Xcelium convention, as described in Name Generation Algorithm for Coverpoint and Cross on page 1046.
- `iff` expressions specified at the coverpoint level are supported.

© 2021 All Rights Reserved.

Considerations for Crosses

- Crosses are supported on coverpoints or variable identifiers, in compliance with the *SystemVerilog LRM*.
- It is recommended that each cross has an explicit name for tool portability. When a name is not explicitly supplied, the LRM rules for naming crosses are used. When the LRM naming rules cannot be implemented, the name follows the Xcelium convention as described in Name Generation Algorithm for Coverpoint and Cross.
- `iff` expressions specified at the cross level are supported.
- The `select` expressions and conditions are supported.
- Hierarchical identifiers are supported.

  ```
  covergroup CG @ (posedge clk);
    AxB: cross sub1.a, sub2.b;
  endgroup;
  ```

  Here, `sub1` and `sub2` can be any scope reference, such as a sub-module, a structure, and so on.

Coverpoints Part of a Cross and Containing Wildcard Bins

If a coverpoint is part of a cross and contains wildcard bins, it is supported if one of the following is true:

- The number of state wildcard characters is ten or less or the number of transition wildcard characters is eight or less.
- The number of state wildcard characters is more than ten and:
  - There is at most one such coverpoint in the cross.
  - The number of non-wildcard coverpoints does not exceed 2048 bins.
  - The product of the number of non-wildcard bins and the number of scalar wildcard bins does not exceed 2^16.

For example:

```
covergroup cg @(posedge clk);
    option.auto_bin_max = 8192;
        A : coverpoint expr1 {
            ignore_bins b_toplevel = {4'h5, 4'h9, 4'hA, 4'hD, 4'hE};
        }
        B : coverpoint expr2 {
            wildcard bins b_00 = {16'b?????????????????1};
            wildcard bins b_01 = {16'b??????????????1?};
            wildcard bins b_02 = {16'b???????????????1??};
            wildcard bins b_03 = {16'b????????????1???};
```


© 2021 All Rights Reserved.

```
            wildcard bins b_04 = {16'b??????????1????};
            wildcard bins b_05 = {16'b?????????1?????};
            wildcard bins b_06 = {16'b????????1??????};
            wildcard bins b_07 = {16'b???????1???????};
            wildcard bins b_08 = {16'b??????1????????};
            wildcard bins b_09 = {16'b?????1?????????};
            wildcard bins b_10 = {16'b????1??????????};
            wildcard bins b_11 = {16'b???1???????????};
            wildcard bins b_12 = {16'b??1????????????};
            wildcard bins b_13 = {16'b?1?????????????};
            wildcard bins b_14 = {16'b1??????????????};
            wildcard bins b_15 = {16'b1??????????????};
        }
        AxB : cross A, B;
    endgroup // cg
```

Considerations for Covergroup Declarations

- If the total number of bins within a covergroup instance exceeds `2^16`, a warning is issued and that covergroup instance is discarded.
- Most of the instance-specific coverage options (as listed in *Table 19-1 of the SystemVerilog LRM*) are supported, but must be specified in the covergroup declaration. The `get_inst_coverage` and `detect_overlap` options are not supported, and covergroups with these options are ignored with a warning.
- Covergroup type options (as listed in *Table 19-3 of the SystemVerilog LRM*) are supported, but must be specified in the covergroup declaration. The `goal`, `strobe`, and `merge_instances` type options are not supported, and covergroups with these options are ignored with a warning.
- Formal arguments for covergroup declarations are supported in compliance with the rules specified in the *Sections 13.5.2* and *19.3* of the *SystemVerilog LRM*. For example, consider the following covergroup declaration:

```
covergroup CG (ref logic[3:0] X, input int C, string commentStr,
input int low, input int high) @(posedge clk);
    coverpoint X
    {
    option.at_least = C;
    bins b1[] = {[low:high]};
    }
option.comment = commentStr;
endgroup
```

```
logic[3:0] bus;
CG cg1 = new(bus, 3, "bottom values", 0, 7);
CG cg2 = new(bus, 2, "upper values", 7, 15);
```

  Violations of the LRM rules generate errors at compilation time.

  Additionally, the following types are not supported for the arguments:

  - An element of an unpacked array

© 2021 All Rights Reserved.

- A member of an unpacked structure
- A class property

Violations results in the offending covergroup instances to be discarded with a warning. Other supported instances of the same covergroup are, however, compiled.

For example, when compiling the following code within a design the covergroup instance, `cg3` is discarded with a warning:

```
logic[3:0] cnt[3:0];
CG cg3 = new(cnt[0], 3, "bottom values", 0, 7);
CG cg4 = new(bus, 3, "bottom values", 0, 7);
```

Coverage information for `cg4` is still created. Thus, type coverage summary contains data for only the supported instances.

- Overriding the built-in `sample` method is not supported. Covergroups that specify a built-in `sample` method are ignored with a warning.
- Block event expressions are not supported as a sampling mechanism. Use of block event expressions generates an illegal-use error, and the compilation process terminates.
- Sampling is supported only by specifying an explicit clocking event in the covergroup declaration. Certain scenarios, such as the following, are not supported and their use results in a warning message and the covergroup is discarded:
  - No clocking event
  - A clocking event made up of multiple clocks containing `iff` expressions.
  - A level-sensitive clocking event with an `iff` expression.

Considerations for Covergroup Instances

A covergroup can be instantiated in an `initial` block provided the following conditions are met:

- The covergroup is defined at the module level.
- The covergroup variable is defined at the module level.
- The covergroup variables are instantiated only once. For example, instances that are determined statically to be exclusive via if-case branching are supported; however, instances within looping constructs are not supported.

If any of the above conditions is not met, the covergroup is discarded with a warning at compilation time.

© 2021 All Rights Reserved.

Multiple instances of the same covergroup are supported; however, the same covergroup variable cannot be instantiated more than once. For example:

```
CG1 cg1 = new;
initial
begin
    #10 cg1 = new;  // same covergroup variable instantiated again
end
```

All the instances of such a covergroup variable are discarded with a warning during compilation.

In the hardware partition, covergroup instances are only supported within modules,generate blocks, or interfaces. The covergroup instances are not supported within `always` blocks, `final` blocks, , function calls, or task calls. These unsupported instances are ignored with a warning.

Declaring a covergroup variable from within a process block is not supported, and such declarations are ignored during compilation.

For example, all instances of a covergroup variable such as the following are discarded with a warning:

```
initial CG cg1 = new;
```

Covergroup instances are only supported in variable declarations and in `initial` blocks as shown below:

```
CG cg1 = new();
CG cg2;
initial begin
    cg2 = new;
end
```

Support for Covergroups in Generate Instances

You can declare and instantiate covergroups inside generate scopes. Various use models are supported, which vary depending on the scope (module or generate) of covergroup declaration, the scope of covergroup instance, the scope of the signals used as expressions or actual argument of covergroup instance and the scope of the new statement associated with instantiation. The following table list out few of the supported use models:

	Declaration Scope	Instance Scope	Expression Scope	Initialization Scope
1	Module	Generate	Module	Generate

© 2021 All Rights Reserved.

	Declaration Scope	Instance Scope	Expression Scope	Initialization Scope

	Example	`module M;`
		`  bit [3:0]m_bus`
		`  covergroup MCG (input int val) @ (posedge clk)`
		`  coverpoint m_bus {}`
		`  endgroup`
		`  generate`
		`  if (PARM) begin: IFG`
		`  MCG cg1 = new (6);`
		`  MCG cg2;`
		`  initial cg2 = new (8);`
		`  end`
		`  endgenerate`
		`endmodule`		

2	Generate	Generate	Module	Generate

	Example	`module M;`
		`  generate`
		`  if (PARM) begin: IFG`
		`  covergroup GCG (input int val) @ (posedge clk)`
		`  coverpoint bus {}`
		`  endgroup`
		`  GCG cg3 = new (6);`
		`  GCG cg4;`
		`  initial cg4 = new (8);`
		`  endgenerate`
		`endmodule`		

3	Generate	Generate	Generate	Generate

	Example	`module M;`
		`  generate`
		`  if (PARM) begin: IFG`
		`  bit [3:0] local_bus`
		`  covergroup GCG (input int val) @ (posedge clk)`
		`  coverpoint local_bus {}`
		`  endgroup`
		`  GCG cg5 = new (6);`
		`  GCG cg6;`
		`  initial cg6 = new (8);`
		`  endgenerate`
		`endmodule`		

4	Generate	Generate	Generate	Generate

	Example	`module M;`
		`  generate`
		`  if (PARM) begin: IFG`
		`  bit [3:0] local_bus`
		`  covergroup GCG (input int val) @ (posedge clk)`
		`  coverpoint local_bus {}`
		`  endgroup`
		`  GCG cg7 = new (6);`
		`  initial cg7.set_inst_name("renamed_cg7");`
		`  endgenerate`
		`endmodule`		

© 2021 All Rights Reserved.

	Declaration Scope	Instance Scope	Expression Scope	Initialization Scope
5	Module	Module and Generate	Module	Module and Generate

	Example	`module M;`
		`  covergroup MCG (input int val) @ (posedge clk);`
		`  coverpoint bus {}`
		`  endgroup`
		`  MCG cg8 = new (3);`
		`  genvar ii;`
		`  generate`
		`  for (ii = 6; ii < 9; ii += 2) begin: FG`
		`  MCG cg9 = new (ii);`
		`  end`
		`  endgenerate`
		`endmodule`		

Covergroup in Vunit or Property Files

Support of covergroups inside property or vunit files provides the flexibility and convenience to separate the verification files from core design files. The covergroups are supported inside the vunit files bound to module or module instance.

Note: The covergroup inside the vunit files creates a generate scope inside the module to the bounded module.

```
vunit V1 (moduleName( {
    covergroup cg1 @ (posedge clk);
      coverpoint state {}
      cross state, address;
    endgroup
    cg1 cg_inst1 = new ();
}
vunit V2 (top. InstanceName) {
    covergroup cg2 @ (posedge clk);
      coverpoint state {}
      cross state, address;
    endgroup
    cg2 cg_inst2 = new ();
}
```

Type Coverage Reporting Scope

The scope of type coverage depends on the initialization of the covergroup instance. For example, a covergroup can have multiple instances and some instances might be initialized

© 2021 All Rights Reserved.

in the module scope and others in the generate scope. The following table specifies the scope provided in the type coverage report.

	Declaration Scope	Initialization Scope	Reporting Scope
1	Module	Generate	Generate
2	Generate	Generate	Generate
3	Module	Module	Module
4	Module	Module and Generate	Module

Accelerating Testbench Covergroups in the Hardware

Covergroups in the software partition can be accelerated in the VXE hardware under certain conditions without user intervention. IXCOM automatically identifies the covergroups that can be accelerated and processes these in the hardware. It scans all the covergroups in the software partition and checks all the signals accessed inside the covergroup.

- When the signals covered in a covergroup reside in the hardware partition, that covergroup is automatically moved to the hardware partition and processed by the Palladium Z1 emulator. IXCOM still preserves the covergroup in the original hierarchy.

  Note: IXCOM does not analyze the covergroups embedded in classes. Embedded covergroups remain in the software partition.

- When the signals covered in a covergroup reside in the software partition, that covergroup remains in the software partition and is processed by Xcelium.

Refer to Criteria for Partitioning Covergroups to the Hardware for details and examples of covergroups moved to hardware partition or retained in software partition. Therefore, a software module can have a mixture of covergroups processed by Xcelium software and Palladium hardware. When a mixture of Xcelium and Palladium covergroups exist, all covergroup statistics are captured in different database files that reside in the same coverage working directory. When you open this directory in IMC, you will see both Xcelium-processed-coverage items and Palladium-coverage items. The Xcelium-processed-coverage items display with instrumented file and line information. To avoid this restriction, partition the testbench by using the `+xcDesignTop` to enable the simulator to directly process the raw coverage construct.

There is no difference in the coverage reports when a software covergroup is accelerated. IXCOM preserves the covergroup labels that you have defined and the coverage results are displayed in the original module in the software partition.

© 2021 All Rights Reserved.

Note: Most of the features supported by Xcelium can be used in the software partition except when a coverage method, call, or reference operates on the design level or refers to the DUT from the software partition.

Hierarchical references to covergroup instances are not supported. When present, an error is issued and IXCOM exits non-zero.

Criteria for Partitioning Covergroups to the Hardware

The following criteria must be satisfied for a covergroup in the software partition to be accelerated, that is to be moved to hardware:

- ❑ Usage must be within the feature subset supported by IXCOM.
- ❑ Sampling clock must be a signal declared in the hardware partition.
- ❑ All coverpoint variables and guard conditions must be signals declared in the hardware partition.
- ❑ Covergroup options and type options assigned in the procedural context are ignored with a warning.
- ❑ Covergroup methods such as `start()`, `stop()`, and `sample()` are not allowed.
- ❑ Coverpoint variable data types are limited to packed and integer data types. Enum types are not allowed.

IXCOM will issue an informational message to you when a covergroup is successfully moved to the hardware partition.

Following is an example of the covergroups moved into the hardware:

Sampling clock, coverpoint variable, and guard condition are in the hardware partition.

```
module tb_monitor_1();
       // sampling clock, coverpoint variable, guard condition
       // are inside DUT
       covergroup CG1 @(top.<DUT_scope>.clk);
           cp1: coverpoint top.<DUT_scope>.bus iff(top.<DUT_scope>.en);
       endgroup
       CG1 cg_inst = new();
endmodule
```

Following are examples of the covergroups NOT moved into the hardware:

- Sampling clock is in the hardware partition, and a mixture of signals is in the hardware partition and on the testbench - DUT interface.

```
module tb_monitor_3();
       // sampling clock and the coverpoint variable are inside DUT
```


© 2021 All Rights Reserved.

```
        // guard condition is TB signal and is on the
        // TB<->DUT interface
        covergroup CG3 @(top.<DUT_scope>.clk);
             cp1: coverpoint top.<DUT_scope>.bus iff(top.en);
        endgroup
        CG3 cg_inst = new();
endmodule
```

- Sampling clock is in the software partition, and the coverpoint variable is in the hardware partition:

```
module tb_monitor_2();
        covergroup CG2 @(top.clk);
            cp1: coverpoint top.<DUT_scope>.bus;
        endgroup
        CG2 cg_inst = new();
endmodule
```

- Sampling clock is in the hardware partition, and a mixture of signals is in the hardware and software partitions:

```
module tb_monitor_4(en);
        covergroup CG3 @(top.<DUT_scope>.clk);
                cp1: coverpoint top.<DUT_scope>.bus iff(en);
        endgroup
        CG3 cg_inst = new();
endmodule

module top();
        bit en;
        dut DUT();
        tb_monitor_4 CHK(en);
endmodule
```

- Covergroup method calls are in the procedural code:

```
module tb_monitor_5();
        covergroup CG5 @(top.<DUT_scope>.clk);
                cp1: coverpoint top.<DUT_scope>.bus;
        endgroup

        CG5 cg_inst = new();
        always @(posedge _start) begin
                cg_inst.start();
        end

        always @(posedge _done) begin
                cg_inst.stop();
        end

endmodule
```


© 2021 All Rights Reserved.

Support for Coverage Methods

The following table lists the predefined coverage methods and the support level provided by IXCOM. See *Section 19.8 of the 2009 SystemVerilog LRM* for more information about the predefined coverage methods.

Method Name	**Support Level**
`sample()`	Not supported Calls to `sample()` are ignored.
`get_coverage()`	Not supported When used, an error is generated and IXCOM exits non-zero.
`get_inst_coverage()`	Not supported When used, an error is generated and IXCOM exits non-zero.

`set_inst_name()`	Supported
	The `set_inst_name()` method is supported only from within an `initial` block. When used outside an `initial` block, a warning is displayed and the name is ignored. The covergroup is not ignored.
	`CG cg1;`
	`initial`
	`begin`
	`cg1 = new;`
	`cg1.set_inst_name(“inst1”);`
	`...`
	`set_inst_name()` can also be called for covergroups instantiated via variable declarations.
	IXCOM allows `$sformatf` to be used as an argument to the `set_inst_name` method. So, you can dynamically change the instance name of the covergroup at run-time in a testbench-call or `initial` block.
	Example:
	`initial`
	`begin`
	`fc2 = new();`
	`fc2.set_inst_name($sformatf("%m.fc2_inst"));`
	`end`

© 2021 All Rights Reserved.

Method Name	Support Level
`start()` and `stop()`	Supported **Note:** If called in the same time slot in which the covergroup is sampled, a race can occur between the method and the evaluation of the covergroup. This should be avoided, because it can cause different behavior when swapped to hardware versus swapped to software versus regular simulation.

Name Generation Algorithm for Coverpoint and Cross

The following points describe how names are generated for coverpoints and crosses.

- If a coverpoint or a cross has a label, that label is taken as the name of corresponding coverpoint or cross. For example:

```
CP1 : coverpoint a;  // Name is "CP1"
CROSS1 : cross a, b; // Name is "CROSS1"
```

- If a coverpoint has no label and the coverpoint is declared on a single variable, the name of the variable is taken as the name of the coverpoint. For example:

```
coverpoint a; // Name is "a"
```

- If a coverpoint has no label and the coverpoint is declared on an expression, the name is internally generated. For example:

```
coverpoint a+b; // Name is "__coverpoint_0"
coverpoint c&b; // Name is "__coverpoint_1"
```

  The integer is reset to `0` for each covergroup as the names are generated.

- If a coverpoint has no label and it uses a reference argument variable, the name of the variable is used as the name of the coverpoint. For example:

```
covergroup cg (ref logic [3:0] f1) @(posedge clk);
    coverpoint f1; // Name is "f1"
endgroup
```

- If a cross has no label and coverpoints are simple variables, the names of variables are used in generating the name of cross. For example:

```
cross a , b; // Name is "__a_X_b"
```

- If a cross has no label and coverpoint labels are used in cross, the coverpoint label names are used for creating the generated name of cross. For example:

```
A: coverpoint a;
B: coverpoint b;
cross A, B; // Name is "__A_X_B"
```


© 2021 All Rights Reserved.

- If a coverpoint has no label and contains an OOMR variable reference, all names in the OOMR are used in forming the name of the coverpoint. For example:

  ```
  coverpoint i1.xx; // Name is "i1_xx"
  ```

- If a tool-generated cross label conflicts with another cross label, a numeric extension is added. For example, if a cross named `_a_X_b` already exists, then a second cross with the same name will have `_1` appended to the name.

- If a coverpoint has no label and contains an OOMR reference that ends with a part select or bit select, an internally generated name is used.

  ```
  coverpoint i1.xx.val[7:0];  // Name is _coverpoint_0
  ```

- If a coverpoint has no label and contains a OOMR reference with a bit select (not at the leaf level), the bit select is removed from the name. Duplicate names have an index attached to them.

  ```
  coverpoint i1.xx[4].val;   // Name is i1_xx_val
  coverpoint i1.xx[5].val;   // Name is i1_xx_val_1
  ```

Wildcards with Vector, Fixed, and Automatic Bins

The following table specifies the support for wildcards with vector, fixed, and automatic bins.

Bin Type or Value Type	State Bins	Transition Bins
Scalar	Full support except for certain Cross bins	Support for <= 8 wildcard characters
Vector	Support for <= 10 wildcard characters	Support for <=8 wildcard characters
Fixed	Support for <= 10 wildcard characters	Not LRM Compliant
Automatic	Support for <= 10 wildcard characters in any illegal or ignore bins	Not LRM compliant

The following example is supported:

```
covergroup CG @(posedge clk);
      A: coverpoint cnt {
         wildcard bins a[2] = {4'b0?00};
         wildcard bins b[] = {4'b1?00};
      }
      B: coverpoint cnt {
         wildcard ignore_bins ign = {4'b000? };
      }
      C: coverpoint cnt {
```


© 2021 All Rights Reserved.

```
         wildcard bins a[] = (4'b???0 => 4'b???1);
      }
   endgroup
```

The following is not supported:

```
covergroup CG @(posedge clk);
      A: coverpoint cnt {
            wildcard bins a[2] = {16'b?0?00};
            wildcard bins b[] = {16'b?1?00};
      }
      B: coverpoint cnt {
            wildcard ignore_bins ign = {16'b?000? };
      }
      C: coverpoint cnt {
            wildcard bins a[] = (16'b???0 => 16'b???1);
      }
endgroup
```

Support for Covergroups in Dynamic Target Mode

Following are the limitations with covergroups in the Dynamic Target mode:

- Coverpoint illegal bin messages are not printed.
- Large covergroups (streaming mode) might lose data if these fill up.

The following kinds of messages are printed for covergroups in the Dynamic Target mode:

- Large covergroups are present in the design. These might show reduced data collection if run in uncontrolled clock domains.
- Full data collection resumed for large covergroups.

For covergroups with the streaming model and/or illegal bins, messages are generated in the following conditions:

- At time 0, if there are known uncontrolled clocks in the design.
- Whenever the design run enters and exits dynamic target mode.

Functional coverage data using covergroups might result in incomplete results when these covergroups are a part of an uncontrolled domain. Covergroups are in an uncontrolled domain in the following scenarios:

- The covergroup trigger clock is an uncontrolled clock created with the `ixc_hwclksrc` primitive.
- The Palladium Z1 emulator is put into dynamic target mode, making all controlled clocks uncontrolled.

© 2021 All Rights Reserved.

When covergroups require a large memory to record the coverage events, a streaming model is used. This requires periodic stopping of the simulation and unloading of the data. If the covergroups is in an uncontrolled domain, this process is unreliable because the emulator can generate data faster than it can be unloaded. This causes data loss resulting in an incomplete view including items marked uncovered when these items really were covered or possibly decreased cover counts per item. But it never results in items getting marked as covered when they are not.

When covergroups contain illegal coverpoint bins, a message is issued when an illegal value is reached. This requires stopping the simulation to unload the data and print the message. If the covergroup is in an uncontrolled domain, this process is unreliable because the emulator might change the actual values of the coverpoint expression while the message is being printed.

For accurate information, ensure that the covergroups are in a controlled clock domain.

Following is an example of a covergroup that prompts these messages in the Dynamic Target mode:

```
reg [7:0] in1;
reg [7:0] cyclecnt;

covergroup cg @ (posedge clk);
      A: coverpoint in1
      {
            illegal_bins ign = { 14 }; // Causes illegal bins message
      }
      B: coverpoint cyclecnt
            {
               bins a[] = { [$:$] };
            }
      AxB : cross A, B; // causes streaming model message.
endgroup; // cg
```

Support for SystemVerilog Functional Coverage Constructs

The following table contains the complete support information for the SystemVerilog covergroups with respect to the syntax specified in the *IEEE-1800-2009 SystemVerilog Language Reference Manual*. The table lists the following information:

- SystemVerilog LRM Reference, with reference numbers quoted inside the square brackets [ ] (*Outer* column)
- Construct description (*Mid-level* column)
- Scope unit (*Inner* column)
- Support in VXE (*VXE* column)

© 2021 All Rights Reserved.

- Any limitations or points to be noted (*Remarks* column)

Outer	Mid-Level	Inner	VXE	Remarks
LRM [19.3]	covergroup instance in	package	No	
LRM [19.3]	covergroup decl in	package	No	
LRM [19.3]	covergroup instance in	compilation unit scope	No	
LRM [19.3]	covergroup decl in	compilation unit scope	No	
LRM [19.3]	covergroup decl in	module	Yes	
LRM [19.3]	covergroup instance in	module - var decl	Yes	Covergroups in DUT only are accelerated.
LRM [19.3]	covergroup instance in	module - initial statement	Partial	An instance can be assigned only once.
LRM [19.3]	covergroup instance in	module - always statement	No	
LRM [19.3]	covergroup instance in	module - final statement	No	
LRM [19.3]	covergroup instance in	module - generate	No	
LRM [19.3]	covergroup decl in	module - generate	No	
LRM [19.3]	covergroup instance in	module - function call	No	
LRM [19.3]	covergroup decl in	module - function call	No	
LRM [19.3]	covergroup instance in	module - task call	No	
LRM [19.3]	covergroup decl in	module - task call	No	
LRM [19.3]	covergroup instance in	program	N/A	Cannot exist in the DUT
LRM [19.3]	covergroup decl in	program	N/A	Cannot exist in the DUT
LRM [19.3]	covergroup instance in	interface	Yes	

© 2021 All Rights Reserved.

Outer	Mid-Level	Inner	VXE	Remarks
LRM [19.3]	covergroup decl in	interface	Yes	
LRM [19.3]	covergroup instance in	class	N/A	Cannot exist in the DUT
LRM [19.3]	covergroup decl in	class	N/A	Cannot exist in the DUT
LRM [19.3]	covergroup instance in	checker	No	
LRM [19.3]	covergroup decl in	checker	No	
BNF Syntax 19-1	covergroup_declaration	arguments	Yes	
BNF Syntax 19-1	covergroup_declaration	Trailing: covergroup_identifier	Yes	
BNF Syntax 19-1	covergroup_declaration	coverage_event	Partial	See event items below for detailed support information.
BNF Syntax 19-1	covergroup_declaration	coverage_spec_or_option	Yes	
BNF Syntax 19-1	coverage_spec_or_option	attributes - coverage_spec	Yes	
BNF Syntax 19-1	coverage_spec_or_option	attributes - coverage_option	Yes	
BNF Syntax 19-1	coverage_option	option	Yes	
BNF Syntax 19-1	coverage_option	type_option	Partial	See individual type_option cells further below in this table for detailed information about this support.

© 2021 All Rights Reserved.

Outer	Mid-Level	Inner	VXE	Remarks
BNF Syntax 19-1	coverage_spec	cover_point	Partial	Limit to 2^16 bins
BNF Syntax 19-1	coverage_spec	cover_cross	Yes	
BNF Syntax 19-1	coverage_event	clocking event	Partial	■ Event variables not supported. ■ Sequences not supported. ■ `iff` not supported.
BNF Syntax 19-1	coverage_event	@@(block_event_expression)	No	
BNF Syntax 19-1	block_event_expression	or operator	No	
BNF Syntax 19-1	block_event_expression	begin hierarchical_btf_identifier	No	
BNF Syntax 19-1	block_event_expression	end hierarchical_btf_identifier	No	
BNF Syntax 19-1	hierarchical_btf_identifier	hierarchical_tf_identifier	No	
BNF Syntax 19-1	hierarchical_btf_identifier	hierarchical_block_identifier	No	
BNF Syntax 19-1	hierarchical_btf_identifier	hierarchical_identifier	No	

© 2021 All Rights Reserved.

Outer	Mid-Level	Inner	VXE	Remarks
BNF Syntax 19-1	variable_decl_assignment	new operation	Yes	
BNF Syntax 19-1	variable_decl_assignment	Optional arguments	Yes	
Semantics Syntax 19-1	class	omitted covergroup_variable_identifier	N/A	Cannot exist in the DUT
LRM [19.3]	new operator	ref arguments	Partial	Only simple variables are allowed.
BNF Syntax 19-2	cover_point	identifier	Yes	When the LRM naming rules cannot be implemented, Xcelium naming conventions are used for unnamed bins.
BNF Syntax 19-2	cover_point	expression	Yes	
BNF Syntax 19-2	cover_point	iff (expression)	Yes	
BNF Syntax 19-2	cover_point	bins_or_empty	Yes	
BNF Syntax 19-2	bins_or_empty	attribute_instance	N/A	Attribute instances are not supported by IXCOM.
BNF Syntax 19-2	bins_or_empty	bins_or_options	Yes	
BNF Syntax 19-2	bins_or_options	coverage_option	Yes	

© 2021 All Rights Reserved.

Outer	Mid-Level	Inner	VXE	Remarks
BNF Syntax 19-2	bins_or_options	wildcard	Partial	Fixed, vector, automatic bins have limited support. See Wildcards with Vector, Fixed, and Automatic Bins section for details.
BNF Syntax 19-2	bins_or_options	bins_keyword	Yes	
BNF Syntax 19-2	bins_or_options	bin_identifier	Yes	When the LRM naming rules cannot be implemented, Xcelium naming conventions are used for unnamed bins.
BNF Syntax 19-2	bins_or_options	bin automatic array	Yes	Illegal and ignore bins are not supported when present in an automatic coverpoint where the number of values is greater than `auto_bin_max`.
BNF Syntax 19-2	bins_or_options	bin vector array	Yes	
BNF Syntax 19-2	bins_or_options	bin fixed array	Yes	Illegal and ignore bins are not supported when a fixed array is present in the coverpoint.

© 2021 All Rights Reserved.

Outer	Mid-Level	Inner	VXE	Remarks
BNF Syntax 19-2	bins_or_options	open_range_list	Yes	
BNF Syntax 19-2	bins_or_options	iff (expression)	Yes	See Considerations for Bins for limitations.
BNF Syntax 19-2	bins_or_options	trans_list	Yes	See Considerations for Bins for limitations.
BNF Syntax 19-2	bins_or_options	default	Yes	Default bins are not tracked.
BNF Syntax 19-2	bins_or_options	default sequence	No	
BNF Syntax 19-2	bins_keyword	bins	Yes	
BNF Syntax 19-2	bins_keyword	illegal_bins	Yes	Illegal bins do not generate a non-zero exit status at run time.
BNF Syntax 19-2	bins_keyword	ignore_bins	Yes	
BNF Syntax 19-2	open_range_list	value_range	Yes	
LRM [19.5]	clocking blocks	coverpoint sampling	No	Clocking block signals are not supported.
LRM [19.5]	open_value_range	[$:expression]	Yes	Additionally, the Xcelium proprietary [$:$] syntax is supported.

© 2021 All Rights Reserved.

Outer	Mid-Level	Inner	VXE	Remarks
LRM [19.5]	open_value_range	[expression:$]	Yes	Additionally, the Xcelium proprietary [$:$] syntax is supported.
BNF Syntax 19-3	trans_list	(trans_set)	Yes	See Considerations for Bins for limitations.
BNF Syntax 19-3	trans_set	trans_range_list	Yes	See Considerations for Bins for limitations.
BNF Syntax 19-3	trans_range_list	trans_item	Yes	See Considerations for Bins for limitations.
BNF Syntax 19-3	trans_range_list	trans_item[* repeat_range]	Yes	See Considerations for Bins for limitations.
BNF Syntax 19-3	trans_range_list	trans_item[-> repeat_range]	No	Goto repeats are not supported.
BNF Syntax 19-3	trans_range_list	trans_item[= repeat_range]	No	Non-consecutive repeats are not supported.
BNF Syntax 19-3	repeat_range	“constant or constant range”	Yes	See Considerations for Bins for limitations.
BNF Syntax 19-4	cover_cross	cross_identifier	Yes	
BNF Syntax 19-4	cover_cross	list_of_coverpoints	Yes	
BNF Syntax 19-4	cover_cross	iff (expression)	Yes	
BNF Syntax 19-4	cover_cross	select_bins_or_empty	Yes	

© 2021 All Rights Reserved.

Outer	Mid-Level	Inner	VXE	Remarks
BNF Syntax 19-4	list_of_coverpoints	“cross_item,{cross_item}”	Yes	
BNF Syntax 19-4	cross_item	cover_point_identifier	Yes	
BNF Syntax 19-4	cross_item	variable_identifier	Yes	
BNF Syntax 19-4	select_bins_or_empty	{bins_selection_or_options}	Yes	
BNF Syntax 19-4	bins_selection_or_options	attribute_instance	N/A	Attribute instances are not supported by IXCOM.
BNF Syntax 19-4	bins_selection_or_options	coverage_option	Yes	See option and type_option section for specific support set.
BNF Syntax 19-4	bins_selection_or_options	bins_selection	Yes	
BNF Syntax 19-4	bins_selection	bins_keyword	Yes	
BNF Syntax 19-4	bins_selection	bins_identifier	Yes	
BNF Syntax 19-4	bins_selection	select_expression	Yes	
BNF Syntax 19-4	bins_selection	iff (expression)	Yes	
BNF Syntax 19-4	select_expression	select_condition	Yes	
BNF Syntax 19-4	select_expression	! select_condition	Yes	

© 2021 All Rights Reserved.

Outer	Mid-Level	Inner	VXE	Remarks
BNF Syntax 19-4	select_expression	select_expression && select_expression	Yes	
BNF Syntax 19-4	select_expression	select_expression , select_expression	Yes	
BNF Syntax 19-4	select_expression	(select_expression)	Yes	
BNF Syntax 19-4	select_condition	binsof(bins_expression)	Yes	
BNF Syntax 19-4	select_condition	intersect {open_range_list}	Yes	
BNF Syntax 19-4	bins_expression	variable_identifier	Yes	
BNF Syntax 19-4	bins_expression	cover_point_identifier	Yes	
BNF Syntax 19-4	bins_expression	cover_point_identifier.bins_identifier	Yes	
LRM [Table 19-1]	options	weight	Yes	Options allowed within declaration only.
LRM [Table 19-1]	options	goal	Yes	Options allowed within declaration only.
LRM [Table 19-1]	options	name	Yes	Options allowed within declaration only.
LRM [Table 19-1]	options	comment	Yes	Options allowed within declaration only.

© 2021 All Rights Reserved.

Outer	Mid-Level	Inner	VXE	Remarks
LRM [Table 19-1]	options	at_least	Yes	Options allowed within declaration only.
LRM [Table 19-1]	options	detect_overlap	No	
LRM [Table 19-1]	options	auto_bin_max	Yes	Number of bins must be divisible by value.
LRM [Table 19-1]	options	cross_num_print_missing	Yes	Options only allowed within declaration
LRM [Table 19-1]	options	per_instance	Yes	
LRM [Table 19-1]	options	get_inst_coverage	No	
LRM [Table 19-2]	type-options	weight	Yes	Options only allowed within declaration
LRM [Table 19-2]	type-options	goal	No	
LRM [Table 19-2]	type-options	comment	Yes	Options only allowed within declaration

© 2021 All Rights Reserved.

Outer	Mid-Level	Inner	VXE	Remarks
LRM [Table 19-2]	type-options	strobe	No	The covergroup strobe type_option is designed to reduce the sampling effects associated with event-based simulators so that excessive events within a timeslot do not cause unwanted samples. The covergroup type_option.strobe is supported in the IXCOM-processed testbench. When present in the hardware partition, it is ignored with a warning.
LRM [Table 19-5]	coverage methods	sample	No	Calling `sample` explicitly is not supported.
LRM [Table 19-5]	coverage methods	get_coverage	No	Getting coverage mid-simulation is not supported.
LRM [Table 19-5]	coverage methods	“get_coverage(ref int, ref int)”	No	
LRM [Table 19-5]	coverage methods	get_inst_coverage	No	Getting coverage mid-simulation is not supported.
LRM [Table 19-5]	coverage methods	“get_inst_coverage(Ref int, ref int)”	No	
LRM [Table 19-5]	coverage methods	set_inst_name(string)	Yes	This is supported only in `initial` blocks.

© 2021 All Rights Reserved.

Outer	Mid-Level	Inner	VXE	Remarks
LRM [Table 19-5]	coverage methods	start()	Yes	
LRM [Table 19-5]	coverage methods	stop()	Yes	
LRM [19-9]	Task/Function	$set_coverage_db_name(name)	No	
LRM [19-9]	Task/Function	$load_coverage_db(name)	No	
LRM [19-9]	Task/Function	$get_coverage()	No	

IXCOM does not support clocking blocks or use of clocking-block signals in covergroup constructs.

Reporting for Functional Coverage Constructs

VXE covergroup reporting includes separate type-based and instance-based reports. Keep in mind the following considerations for these reports:

- The instance-based report does not display type-based report.
- By default, the default value for `option.per_instance` is set to zero (`0`), not one (`1`) as per the LRM.
- In VXE, by default, covergroup instance names are auto-generated if not explicitly specified. Specify the instance name by using the `set_inst_name()` method or `option.name` setting. The instance-based report displays the user-specified names.

Error Handling for Functional Coverage Constructs

The following table describes how errors are categorized and handled when using SystemVerilog coverage constructs with IXCOM.

Category	Result
Illegal HDL	IXCOM issues error message, and exits non-zero.

© 2021 All Rights Reserved.

Category	Result
Not supported in VXE	IXCOM issues warning, and covergroup might or might not be ignored. See below.
Not supported in Xcelium	IXCOM issues warning, and covergroup is ignored.
Covergroup exceeds capacity limitations	IXCOM issues warning, and covergroup instance is ignored.
Unsupported run-time commands	`xeDebug` issues a warning.
Illegal bin hit	An error message is displayed.

Illegal Bins

The *SystemVerilog LRM* permits specification of illegal bins. However, in VXE, when an illegal bin is encountered, an error similar to the following is displayed:

```
ERROR: (File: ill001.v Line: 31):(Time: 55) Sampled value (3) for
       coverpoint (top.DUT.cg1.cnt) is an illegal value.
```

When a cross-illegal bin is encountered, an error message similar to the following is displayed:

```
ERROR: (File: illCross022.v Line: 50):(Time: 285) Illegal cross tuple
       (< 5, 5 >) occurred corresponding to illegal cross bin (err) of
       cross (top.inst1.cg1.axb).
```

Support for SystemVerilog Control Functions

This section describes the partial support for control tasks that can be used to control coverage collection via testbench control logic.

Note: None of these control functions are supported by Xcelium.

$coverage_control

The `$coverage_control` function is specified in the SystemVerilog LRM section 40.3.1. It is used to reset coverage counters. With IXCOM compiler, this function only resets coverage counters on coverage elements that have been transformed by IXCOM. This includes all code coverage counters in the DUT, all covergroup counters in the DUT, and all assertion counters in the IXCOM-visible testbench and the DUT. You can use `$coverage_control` only in the software testbench or in a `translate_off` region. Xcelium does not support `$coverage_control`. So, the control call affects coverage only within the DUT.

© 2021 All Rights Reserved.

The SystemVerilog LRM specifies the following syntax for `$coverage_control`:

```
$coverage_control(control_constant,
                  coverage_type,
                  scope_def,
                  modules_or_instances)
```

The only supported control_constant is `` `SV_COV_RESET ``. Specifying other values results in an error during IXCOM compilation.

The `coverage_type` field limits the action to a particular type:

- `` `SV_COV_ASSERTION ``
- `` `SV_COV_FSM_STATE ``
- `` `SV_COV_STATEMENT ``
- `` `SV_COV_TOGGLE ``

IXCOM, does not support the above-mentioned level of control for `coverage_type`. It, instead, supports the proprietary types that allow control at a more abstract level. The following table lists the IXCOM support for the values for `coverage_type`:

`coverage_type` **Value**	**Specified in 1800-2012 LRM 40.3.1**	**Supported/Unsupported by IXCOM**
`` `SV_COV_ASSERTION ``	Yes	Not supported.
`` `SV_COV_FSM_STATE ``	Yes	Not supported. FSM coverage not supported.
`` `SV_COV_STATEMENT ``	Yes	Not supported. Ambiguous with reference to Xcelium statement coverage types.
`` `SV_COV_TOGGLE ``	Yes	Not supported. All code coverage counters must be reset at once.

© 2021 All Rights Reserved.

`` `SV_COV_TYPE_CODE ``	No	Partially Support The only supported value for `scope_def` is `` `SV_COV_HIER ``. The only supported value for `modules_or_instances` is `"$root"`. Note that the double quotes enclosing `$root` are required.
`` `SV_COV_TYPE_FUNCTIONAL ``	No	
`` `SV_COV_TYPE_ALL ``	No	

$coverage_save

You can use `$coverage_save` only in the software testbench or in a `translate_off` region. Xcelium does not support `$coverage_save`. So, the control call affects coverage only within the DUT.

The LRM specifies the syntax for `$coverage_save` as follows:

```
$coverage_save(constant_type,"name")
```

The only supported constant_type is `` `SV_COV_TYPE_ALL ``. Usage of other constant types listed in the IEEE-1800-2012 LRM for this function result in a run-time warning. indicating that the argument value is not supported, and all coverage will be dumped. Usage of an invalid constant type results in a compile-time failure.

© 2021 All Rights Reserved.

General Flow for Using Coverage

To use the coverage feature with Palladium Z1, follow these steps:

1. Enable coverage at compile time by using the `ixcom` or `xrun -hw` command. Refer to the compile-time options listed in Table 20-1, “Compile-Time Options for Enabling Coverage,” on page 1066.
2. Generate the coverage information at run time using the `xrun`, `xeDebug`, or `xrun -xeDebug` command. Refer to Generating a Coverage Database at Run Time on page 1076.
3. View the coverage information by using the IMC tool. For details, refer to Using Integrated Metrics Center for Analyzing Coverage Data.

To enable support for coverage at compile time, you need to use the following options with either the `ixcom` or `xrun -hw` command:

© 2021 All Rights Reserved.

Table 20-1 Compile-Time Options for Enabling Coverage

Option	Xcelium Equivalent	Description
`-coverage [`**`b`**`lock | `**`t`**`oggle | f`**`u`**`nctional | `**`e`**`xpr | `**`a`**`ll]`	`-coverage <`*`target`*`>`	Enables coverage. ■ **`b`**`lock` - enables block coverage ■ **`t`**`oggle` - enables toggle coverage ■ `f`**`u`**`nctional` - enables functional coverage ■ **`e`**`xpr` - enables expression coverage ■ **`a`**`ll` - enables all supported types of coverage You can use shorthand notation specified in **bold** font. For example, you can use **`t`** for `toggle` and **`u`** for `functional`. You can specify multiple options in the same line using colon (**`:`**) as the separator, such as `-coverage b:t`. By default, all modules compiled for hardware are included under code coverage. To specify particular design portions to be covered, use the `-featureFile` and `-covdut` options.
`-covdut <`*`dut_module`*`>`	`-covdut <`*`dut_module`*`>`	Specifies the top-level design unit for coverage. If the design unit does not exist in the design, an error message is generated. All of the design units instantiated within the `<`*`dut_module`*`>` are marked for coverage. You can specify the `-covdut` options multiple times to specify different top-level design units. It is better to use the more powerful `-featureFile` option instead the `-covdut` option. Mixing the `-featureFile` and `-covdut` options in the same compile is not recommended.

© 2021 All Rights Reserved.

Option	Xcelium Equivalent	Description
`-featureFile` <*`feature_file`*>	`-covfile` <*`file`*>	Used in conjunction with the `-coverage` option, the `-featureFile` option supports enabling and disabling coverage at the instance, design unit, or file level. For details, refer to Enabling and Disabling Features at Compilation Time.
`-featureFile` `expression_coverage`	`select_coverage` `deselect_coverage`	Enables or disables expression coverage on portions of the design using the `expression_coverage` feature specification file.
`+tcovTypes+`<*`type`*> (Only for toggle coverage)	None	Enables you to specify the types of variables to be tracked: ■ `p` - covers all the module and entity ports ■ `v` - covers all the Verilog and VHDL variables ■ `n` - covers all the Verilog nets and VHDL signals You can specify mutiple types of variables together, like `+tcovTpes+v:p:n`. The default value is `p`. If you specify this switch more than once, all the specified types are honored.

© 2021 All Rights Reserved.

Option	Xcelium Equivalent	Description
`+ccovModelSummary` (Only for toggle, expression, and block coverage)	None	Enables creation of two comma-separated value (`.csv`) files to help you identify the portions of the design that should be considered for code and expression coverage. The files are created as follows: ■ <*design_dir*>/ `ccov_model_summary_inst.csv` ■ <*design_dir*>/ `ccov_model_summary_type.csv` You can import these `.csv` files into a spreadsheet tool for browsing through the design to see how code coverage items are distributed. This is particularly useful when the available space on the emulator is limited and you need to prioritize the portions of the design to be covered. Using the `+ccovModelSummary` option with expression-coverage enabled, ensures that: ■ Expression items are included in the overall `Coverage Items` columns for both instance-based and type-based `.csv` files. ■ The `Number of Local Expressions + Number of Sub-Instance Expressions` column is added in the instance-based `.csv` file. ■ The `Number of Expressions` column is added to the type-based `.csv` file. Refer to Instance-Based Summary and Type-Based Summary for details.
`+covGenFeatureFile [...]`	None	Creates a feature file to identify the design units that are potential library cells.

© 2021 All Rights Reserved.

Option	Xcelium Equivalent	Description

`+ccovProfile`
 	
(Only for toggle and block coverage)
	None	Enables tracking of statistics related to code-coverage processing. For example, the time spent in processing code coverage at compile time is reported, such as:
		
		`Code Coverage - End: cpu time = 1.24 sec, elapsed time = 2 sec`
		
		`heap memory size = 0.00M bytes`
		
		This output is included in the `ixcom.log` file.
		
		Note: The counter used by this feature impacts other profiling counters used in the IXCOM compilation.

`+tcovMDA[+<`*`exponent`*`>]`	Coverage file option: `set_toggle_scoring -sv_mda`	Enables support for SystemVerilog MDA signals for toggle coverage where total size is less than or equal to $2^{<exponent>}$. By default, the `<`*`exponent`*`>` is 14, which is16k bits. The maximum supported value is 20, which is 1M bits.
`-covAddResetLogic`	None	Enables reset of toggle, block, expression, and assertion counters at run time in hardware. This option is required to get correct reset behavior when the design cannot be swapped to simulation. In addition, the optional value of `e` or `expr` can be used to directly specify expression coverage.
`-covfile <`*`file`*`>` (Only for covergroups in the testbench)	`-covfile <`*`file`*`>`	Specifies the file that lists the coverage-specific options in Xcelium format, to be processed by IXCOM. **Note:** Not all of the Xcelium-supported options are supported by IXCOM. See the section on Using the -covfile <file> Option with Expression Coverage for more detail.

© 2021 All Rights Reserved.

Option	Xcelium Equivalent	Description
`-covMakeNamesUnique`	None	Appends a unique identifier to the module name written into the Unicov database if that particular module name already exists in the database.
`+fcovCoverCnt` (Only for covergroups that reside in the DUT)	None	The `+fcovCoverCnt+<N>` option specifies the minimum number of bits [default: `N=8`] to be used for the bin counters of covergroups in the DUT. Valid values for `N` include `1`, `4`, `8`, `16`, and `32`. When values other than these are specified, IXCOM compiler generates a compile-time error. Once an individual counter reaches its maximum value, the counter stays fixed at that value.

Instance-Based Summary

The instance-based summary is useful for identifying how to partition a given design for code coverage. A sample of the `.csv` file is shown below.

Figure 20-4 Sample Instance-Based Summary

	A	B	C	D	E	F	G	H	I	J
1	Instance	Module Definition Name	Number of Local Toggles	Number of Local Blocks	Number of Expressions	Number of Local Coverage Items	Number of Local Toggles + Number of Sub-Instance Toggles	Number of Local Blocks + Number of Sub-Instance Blocks	Number of Local Expressions + Number of Sub-Instance Expressions	Number of Local Coverage Items + Number of Sub-Instance Items
2	top	top	0	11	858	869	4972792	275691	4935	5253418
3	top.DUT	DUT	835	0	0	835	4792686	250134	838	5043658
4	top.DUT.B_CORE	CORE	844	0	0	844	1195591	76847	356	1272794
5	top.DUT.A_CORE	CORE	844	0	0	844	1195591	76847	356	1272794
6	top.DUT.RCVR	RCVR	206	0	0	206	343975	4107	0	348082
7	top.DUT.A_CORE.FPU	FPU	368	0	0	368	294780	11222	0	306002
8	top.DUT.B_CORE.FPU	FPU	368	0	0	368	294780	11222	0	306002
9	top.DUT.A_CORE.IUNIT	IUNIT	840	0	0	840	262125	12702	0	274827
10	top.DUT.B_CORE.IUNIT	IUNIT	840	0	0	840	262125	12702	0	274827
11	top.DUT.XMITR	XMITR	1522	0	0	1522	265023	5321	0	270344
12	top.DUT.B_CORE.FPU.SUB	SUB	1041	0	0	1041	232687	8220	0	240907
13	top.DUT.A_CORE.FPU.SUB	SUB	1041	0	0	1041	232687	8220	0	240907
14	top.DUT.A_CORE.IUNIT.IPROC	IPROC	1257	0	0	1257	228057	10670	0	238727
15	top.DUT.B_CORE.IUNIT.IPROC	IPROC	1257	0	0	1257	228057	10670	0	238727
16	top.DUT.GDI	GDI	925	0	0	925	163705	19356	0	183061
17										

© 2021 All Rights Reserved.

Start from the last column that contains the cumulative coverage contained within the specified instance. The file is sorted by the last column so the instances with the most cumulative coverage are reported first. The other columns are useful for advanced analysis and sorting.

For example, according to cell H2, this design has approximately 5.25 million coverage items. If the main intent is to get coverage on the two `CORE` instances (`A_CORE` and `B_CORE`), this spreadsheet indicates that the number of coverage items will be reduced by about half if coverage is restricted to `top.DUT.A_CORE` and `top.DUT.B_CORE`. You can use the `-featureFile` option for controlling coverage for these instances.

> **Note:** Actual capacity usage might be different based on internal optimizations. The toggle values listed in the above table refer to the number of variable bits being toggled.

Type-Based Summary

The type-based summary is useful for identifying the high-capacity library cells that are not interesting for code coverage. A sample `.csv` file is shown below.

Figure 20-5 Sample Type-Based Summary

	A	B	C	D	E	F	G	H
1	Design Unit	File	Number of Toggles	Number of Blocks	Number of Expressions	Number of Coverage Items per Design Unit	Instances of Design Unit	Number of Coverage Items per Design Unit Times Instances of Design Unit
2	INV	prims.v	2	0	0	2	423345	846690
3	MUX2	prims.v	5	0	0	5	52407	262035
4	LATCH01X	prims.v	3	2	0	5	34485	172425
5	NAND2	prims.v	3	0	0	3	57415	172245
6	NOR2	prims.v	3	0	0	3	36509	109527
7	MUXE202X	prims.v	4	0	0	4	20850	83400
8	NAND202X	prims.v	3	0	0	3	23065	69195
9	INV02X	prims.v	2	0	0	2	33778	67556
10	MUXE201X	prims.v	4	0	0	4	15212	60848

Start from the last column that contains the total number of coverage items associated with a specific design unit. Often, the largest consumers of coverage space are low-level library cells. Depending on your requirements, you can decide whether or not coverage on these is important. Identify the non-interesting high-capacity design units, and remove these from the coverage consideration.

© 2021 All Rights Reserved.

For example, the previous spreadsheet indicated that there are approximately 5.25M coverage items in the design. This spreadsheet tells us that 846,690 coverage items (roughly 16% of the coverage items in the design) are related to a low-level inverter primitive. A primitive such as this is not typically interesting with respect to coverage reporting. In fact, mostly, all the primitives listed would not be interesting for code coverage. Then, you can use the `-featureFile` option to remove all the design units declared in `prims.v` from code-coverage consideration. This would lower the 5.25M coverage items reported down to 3.55M for just the nine design units shown.

> **Note:** Due to optimizations, the capacity change will not be directly proportional, but it will be a noticeable improvement.

These compile-time options must be specified either with the `ixcom` command or with the `xrun -hw` command. The `xrun -hw` command enables you to use the *xrun* utility with the Palladium Z1 emulator. This *xrun* use model is described in Chapter 25, "Compiling and Running Simulation Acceleration Designs with *xrun.*"

Limitations of Type-Based Reports

During the synthesis process, design units are made unique. This can have a negative effect on type-based reports because not all results will be aggregated together if the process to make design units unique affects the coverage collected for a design unit. An example of this is the toggle coverage associated with a parameterized bus:

```
reg [size-1:0] bus;
```

In addition to parameters, using the `-featureFile` option can also have similar results.

As this changes the coverage characteristics, the type information is split into separate modules. Whenever the design unit differs with respect to the coverage items, a separate entry is made in the type-based report.

Making Names of Modules and Design-Units Unique in the Type-Based Summary

The `-covMakeNamesUnique` option ensures that no module or design-unit has the same name in the Unicov database. This is achieved by adding a unique identifier to the name of the module when another module of that name already exists. Assuming a module is named `x`, a unique name would be of the form `x#(0x6326652f)` in which the number `0x6326652f` is a number unique to this kind of module `x`. This option is useful if type-based operations are used in IMC because IMC does not know how to handle scenarios in which a module is represented more than once in the database. Mostly, this option is not required because instance-based operations are most common and do not lose data. By applying a unique name to these duplicate module names, the data associated with instances of that particular type will be lost if not carefully scanned within IMC.

© 2021 All Rights Reserved.

When the `-covMakeNamesUnique` option is used, IXCOM generates this number internally. Therefore, the unique number can change from release to release; But it is consistent from compile to compile. Xcelium uses a similar format for parameterized modules. To visually differentiate from the Xcelium-naming convention, IXCOM uses the C-style `0x` prefix instead of the Verilog-style radix labeling.

Using the `-covMakeNamesUnique` option prohibits cross-engine instance and type merging when dealing with modules and design-units that are made unique in this fashion. The IMC and vManager tools treat such module names as non-matching. So, if you need cross-engine merging, do not use the `-covMakeNamesUnique` option.

The following module illustrates the different naming behavior.

```
module x();
   parameter sz=8;
   parameter width=32;
   wire [(sz-1):0] small_bus;
   wire [(width-1):0] big_bus;
endmodule
```

Type Name Table

Assume that toggle coverage is enabled and that there are two instances of the `x` module in the design with the following parameter values:

```
x #(2, 6)
x #(1, 8)
```

The following table shows the different naming techniques used by Xcelium and IXCOM.

	Type name for: sz = 2, width = 6	**Type name for: sz = 1, width = 8**
IXCOM (default)	`x`	`x`
IXCOM `-covMakeNamesUnique`	`x`	`x#(0x6326652f)`
Xcelium (default)	`x`	x (in the instance report) Non-existent in type report.
Xcelium (parameter-based option)	`x#(32'h00000002,32'h00000006)`	`x#(32'h00000001,32'h00000008)`

© 2021 All Rights Reserved.

Note that no spaces are allowed in the module name. The first module encountered by IXCOM preserves the original name.

Examples for Enabling Coverage with the IXCOM Compiler

Following is an example of the `ixcom` command that tracks the coverage for each instance of the `dut`, `alu`, and `cpu` design units, and tracks all the ports and variables:

```
ixcom [ixcom_switches] -coveragetoggle \
                       -covdut dut -covdut alu -covdut cpu +tcovTypes+pv
```

Following is the corresponding sample `xrun -hw` command that tracks the same toggle coverage metrics as the above `ixcom` command:

```
xrun -hw [other_switches] -coverage toggle \
                          -covdut dut -covdut alu -covdut cpu +tcovTypes+pv
```

Compile-Time Output for Toggle and Block Coverage

During compilation, the information is generated with respect to code coverage in the `ixcom.log` file. Following is an example of this information. `%d` indicates a numeric placeholder for actual values calculated during compilation.

```
Coverage Model Summary
            Block Coverage Items:          2
           Toggle Coverage Items:          9
  Expression Coverage Conditions:          1
-------------------------------------------
       Total Code Coverage Items:         12
```

The *User Summary* section details the information about the number of items tracked for coverage. These numbers should match the total number of coverage items in the run-time coverage report. Refer to Using Integrated Metrics Center for Analyzing Coverage Data.

Compiling Designs with Functional Coverage Constructs using IXCOM Compiler

To enable the parsing of the SystemVerilog covergroup constructs in the design, ensure that either the `+sv` option is included with the `ixcom` command, or the file has the `.sv` extension.

If you have not enabled functional coverage via the `-coverage functional` option, but you have specified the `+sv` option, the covergroup statements in both the hardware and software partitions are silently ignored.

© 2021 All Rights Reserved.

Controlling Functional-Coverage Counter Sizes

For performance reasons, coverage counters for synthesizable covergroups are stored on the emulator. The number of bits in these counters can impact capacity and in some cases the overall simulation performance. The `+fcovCoverCnt+<N>` option of the `ixcom` command enables some control over the bit size of the internal counter.

The `+fcovCoverCnt+<N>` option specifies the minimum number of bits [default: `N=8`] to be used for the bin counters of covergroups in the DUT. Valid values for `N` include `1`, `4`, `8`, `16`, and `32`. When values other than these are specified, IXCOM compiler generates a compile-time error. The value of `1` results in the most optimal instrumentation model. Once an individual counter reaches its maximum value, the counter stays fixed at that value.

Counter values beyond the specified size can occur under the following circumstances:

- If the `+fcovCoverCnt` value cannot represent the `at_least` value for the coverpoint, cross, or covergroup, the local counter is increased as needed, to the smallest valid `+fcovCoverCnt` value that can contain the number. For example, suppose you specify `+fcovCoverCnt+1`, but there is a local covergroup option of `at_least=5`, the maximum value that can be stored in `1` bit is the number `1`. For correct coverage, IXCOM increases the size of that covergroup's counter to `4` bits. An information message is also displayed. Note that a size of 1 results in the most optimal instrumentation model.
- If a coverpoint is feeding a cross. IXCOM implementation of crosses always maintains the minimum number of bits for each crossed combination. As a result, the coverpoint counts could go beyond the specified minimum in some scenarios.

Consider the following example.

Cross Example

```
typedef enum int {UP, DOWN} vertical;
vertical y;
typedef enum int {LEFT, RIGHT} horizontal;
horizontal x;
covergroup CG @(posedge clk);
    cross x,y;
endgroup

CG cg1 = new();
```


© 2021 All Rights Reserved.

Assuming `+fcovCoverCnt+1` has been specified, IXCOM maintain a `1` bit counter for each combination in the cross. These are:

x	y	Counter Size	Example Counts
LEFT	UP	1	1
LEFT	DOWN	1	0
RIGHT	UP	1	1
RIGHT	DOWN	1	1

From the *Example Counts* column, all the combinations except for `{LEFT,DOWN}` are covered. The side effect of maintaining this granularity is that the coverpoint can exceed the counter size. In the above example, the coverpoint for `x==RIGHT` will be incremented twice in the database. Once for `{RIGHT,UP}` and once for `{RIGHT,DOWN}`. This results in an end counter value of `2` even though only `1` bit for the minimum counter size is expressed.

Additional Limitations of Coverage Support

Note the following limitations when using the coverage features supported by IXCOM:

- Clocking blocks are not supported by IXCOM. Thus, using clocking block signals in covergroup constructs is not supported.
- Coverage collection for toggle, block, and covergroup counters in the DUT is not supported if low-power emulation is enabled with the following options of the `ixcom` command:
  - `-lps1801`
  - `-lps_cpf`
  - `-lps_1801+force`

Generating a Coverage Database at Run Time

Coverage statistics are written to the database during the end-of-simulation processing. The output data is stored in the Cadence UniCov format. This format is used by Xcelium, Specman, JasperGold Apps, and other utilities for generating coverage data. Use the IMC tool for analyzing coverage results. Note that for code coverage in VXE, the vPlanner tool is not supported.

© 2021 All Rights Reserved.

The run-time environment enables interactive dumping of coverage data. This is useful in Palladium environments in which you can run multiple *tests* within one Palladium session and want to get independent coverage results for the *tests* within a Palladium session. In addition to creating the database, operations to reset counters and to disable the automatic end-of-simulation creation of the database are included.

Use the `+covPostpone` so that the emulator does not have to wait for the full creation of the Unicov database.

The following figure illustrates the run-time environment with multiple tests and separate coverage databases.

The following table describes run-time options specific to code-coverage collection. These options can be used with the any of the following commands:

- `xrun`
- `xrun -xeDebug`
- `xrun -hw`
- `xeDebug`

Run-time Option	Description
`-covmodeldir` <*arg*>	Specifies the path to the coverage model or scope directory that should be used in place of the default model or scope directory. By default, this is set to <*cov_work*>/<*scope*>.
`-covnomodeldump`	Specifies that the coverage design database (model) should not be dumped.
`-covoverwrite`	Enables overwriting of existing coverage output files and directories.

© 2021 All Rights Reserved.

Run-time Option	Description
`-covscope` <*arg*>	Specifies the scope name for the coverage-results databases as <*covworkdir*>/<*covscope*>/<*covtest*>. By default, this is set to `scope`.
`-covtest` <*arg*>	Specifies the test name for the coverage-results databases as <*covworkdir*>/<*covscope*>/<*covtest*>. By default, this is set to `test`.
`-covworkdir` <*arg*>	Specifies the base name for the coverage-results databases as <*covworkdir*>/<*covscope*>/<*covtest*>. By default, this is set to `./cov_work`.

The options with arguments control the directory structure for storing the coverage information. In general, the format is as follows:

<*covworkdir*>/<*covscope*>/<*covtest*>

Keep the following points in mind about the run-time options:

- If you specify an option with an argument more than once, the first occurrence of the option is used and others are ignored with a warning.
- If you specify an option without an argument more than once, the extraneous occurrences are ignored without a warning.
- If the `-covmodeldir` option is used with the `-covnomodeldump` option, the `-covnomodeldump` option has higher precedence.
- If the `-covmodeldir`, `-covscope`, `-covtest`, or `-covworkdir` option is specified multiple times, the first instance is honored, and the others are ignored.

Coverage Run-time Output

By default, the model files specific to the topology of the design are stored in the <*covscope*> directory. If already present from a previous run, the model files are reused. The <*covtest*> directory contains test-specific data. Individual test results must be kept in

© 2021 All Rights Reserved.

separate <*covtest*> directories to that the data from each test is maintained. The following table describes the file types that are created at the end of simulation for code coverage:

File Extension	Description	Notes
`.*cmf`	Coverage Model Files	Model information used to generate Unicov Model and Data files
`.ucm`	Unicov Model File	Used by IMC
`.*cdf`	Coverage Data Files	Preprocessed coverage data from the emulator
`.ucd`	Unicov Data File	Used by IMC
`ixcc_*.ksh`	Generate Unicov Script	Shell script to allow decoupled creation of Unicov from the emulation run. Requires the `+covPostpone` option to be set. Specific file information is reported at the end of simulation.

Using xeDebug to Generate Coverage Information

You can also use the xeDebug tool in place of using the *xrun* utility to generate the coverage information at run time. The only difference in the usage is that the `xeDebug` command does not support the use of Xcelium-related options directly on the command line. Therefore, these Xcelium-related options must be passed through the xeDebug tool for run-time processing by using the `--xmsim` option of the `xeDebug` command.

For example:

```
xeDebug --xmsim -covworkdir /lan/cov_data \
        -covtest test1 -- [other_xeDebug_options]
```

Here, the `/lan/cov_data` and `test1` settings are specified, and for the other settings, the default values are used. The xmsim-specific arguments are encompassed within the `--` (hyphens) grouping. It is very important that the `--` (hyphens) are carefully placed. Refer to the `xeDebug` command line usage for more information in the *VXE Command Reference Manual*.

Run-Time Support for Xcelium coverage Command

The following table lists the options of the Xcelium `coverage` command that apply to covergroups and how these operate in VXE at run time. The Xcelium `coverage` command at run time is supported through `xc coverage_*` commands that you can specify on the `xmsim>` or `xeDebug>` command prompt.

© 2021 All Rights Reserved.

Note: These `xc coverage_*` commands only apply to coverage data stored in UXE Unicov databases. In other words, these commands only operate on coverage items that are handled by Palladium. For the coverage items that are handled by the simulator, such as when you have used the `+xcDesignTop` option of the `ixcom` command, use the appropriate simulation commands.

Xcelium `coverage` Command	**Equivalent VXE Command**	**Supported for**
`coverage -analyze`	Not Supported in VXE	N/A
`coverage -code`	`xc coverage_code -reset`	N/A
`coverage -dump`	`xc coverage_dump`	Toggle, Block, Assertions, and Covergroups.
`coverage -fsm`	Not Supported in VXE	N/A
`coverage -functional`	`xc coverage_functional -reset`	N/A
`coverage -off`	`xc coverage_off`	Toggle, Block, Assertions, and Covergroups.
`coverage -setup`	`xc coverage_setup`	Toggle, Block, Assertions, and Covergroups.
`coverage -toggle`	Not Supported in VXE	N/A
`coverage -vplan_analyze`	Not Supported in VXE	Deprecated in Xcelium

In addition to the above-mentioned Xcelium coverage command options, you can also use `xc coverage_reset` command (no Xcelium equivalent) to reset the counters for toggle, block, assertions coverage, and covergroups database.

This reset mechanism only clears the coverage counters that are being instrumented by the Palladium Z1 emulator. This command does not clear counters that are controlled by the simulator. Counters not cleared include covergroups in the testbench, and any coverage items in a testbench that was compiled with the `+xcDesignTop` option. For more information, on which coverage items are controlled by the emulator, see Coverage Databases and Co-simulation section.

© 2021 All Rights Reserved.

Running the Designs with Functional Coverage Constructs

When an IXCOM-compiled design including coverage constructs is run on the hardware by using the xeDebug run-time tool, the testbench covergroups are tracked by Xcelium, and in some scenarios, even by VXE. The DUT covergroups are tracked by VXE. See Run-Time Support for Xcelium coverage Command for more details.

Use `finish`, `exit`, or `$finish` statements to trigger the extraction of coverage counters from the hardware at the end of simulation.

Important

At times, the `run` and `finish` commands of Xcelium can generate recoverable errors. These errors do not terminate the simulation, but can cause xeDebug to exit with a non-zero exit status. To prevent this problem, encapsulate the `run` and `finish` simulation commands within a `catch` statement.

At the end of simulation, a coverage database is automatically generated. Following is an example of the default output at the end of simulation.

```
coverage setup:
    workdir  :  ./cov_work
    dutinst  :  top(top)
    dutinst  :  _ixc_isc(_ixc_isc)
    dutinst  :  xc_top(xc_top)
    design   :  design
    testname :  test
coverage files:
    model(design data) : ./cov_work/design/00000000_696b.ucm
    data               : ./cov_work/design/test/icc.ucd
```

Running Designs with Coverage in ICA and LA Modes

The In-Circuit Acceleration (ICA) mode enables mixing of uncontrolled clocks with a simulation environment.

The following restrictions apply while running covergroups in an uncontrolled domain or in the Logic Analyzer (LA) mode:

- Messages for illegal bins are suppressed.
- Covergroups where the number of bins is greater than 2^11 are not fully sampled.

Using Run-time Use Models

This section describes the following run-time use models associated with creating coverage databases:

© 2021 All Rights Reserved.

- **Attached-Coverage Generation** - Generates Unicov in a blocking fashion. This is the default model.

  Attached-coverage generation is the default mode of operation in which the run time pauses until the Unicov database has been fully created. This is useful when the database is immediately referenced and you need to ensure data availability.

  vManager expects the coverage database to be available when the simulator exits. Attached-coverage generation must be used when using vManager to manage your sessions.

  If you do not want to use the default attached-coverage generation model, you can select the run-time option to specify the coverage-generation mode as per your requirement.

- **Detached-Coverage Generation** - Generates Unicov in a non-blocking fashion

  Detached-coverage generation starts the Unicov generation on the workstation in the background. This is useful when local-workstation resources are ample and the Unicov coverage data is not immediately needed. The most common use model in which detached-coverage generation is useful is a single session that dumps the coverage database multiple times.

  Use the `+covDetach` option to detach the coverage generation from the emulation run-time process. Once the raw data files are written to disk, the process to create the Unicov database is spawned off on the local workstation. This allows the emulator to continue without having to wait for the Unicov database generation. An informational message is issued that contains the process identifier of the detached coverage generation process. This can help you determine when the process has completed.

- **Postponed-Coverage Generation** - Generates Unicov at a later time and (potentially) on a different workstation.

  Postponed-coverage generation allows you to decouple the generation of Unicov from the emulation run. This is useful for scenarios in which writing out large coverage databases impacts run-time performance. The raw Palladium data files are written to disk and emulation can continue. A script is created per data set to enable you to create a fully compliant Unicov database.

  Use the `+covPostpone` option to decouple coverage generation from the Palladium session. The raw coverage data files are placed in the same target directories in which the Unicov-generated files would be placed. The target-coverage directory contains the `ixcc_*.ksh` script with instructions for generating the Unicov database. You can run this script if the VXE, Xcelium, and MDV releases are correctly set up and the paths to the coverage database are accessible. If this setup is not correct or if the IXCC tool exits non-zero, the `ixcc_*.ksh` script exits non-zero. You must manage and, if required, remove the `ixcc_*.ksh` script and internal data files (`*.*cdf`) when you use postponed-coverage generation.

© 2021 All Rights Reserved.

An error message is printed and the simulator exits non-zero if you specify both `+covPostpone` and `+covDetach` options on the command line.

Using Integrated Metrics Center for Analyzing Coverage Data

Integrated Metrics Center (IMC) can be used to analyze and operate on coverage data. Typical operations of IMC include:

- Displaying coverage results
- Merging coverage data
- Ranking tests based on coverage amounts
- Refining tests

MDV 17.10 is recommended for analyzing coverage results.

Xcelium tools and Integrated Comprehensive Coverage (ICC) must be installed to create and interact with the coverage data. Refer to the *Integrated Coverage User Guide* available in the *Xcelium documentation set* for details on these and other operations.

The rest of this section covers the way that IMC interacts with the Palladium Z1 for coverage collection.

Important

To utilize the functional coverage feature support described in this chapter, use the Xcelium version recommended in Xcelium Versions Required for Coverage Features. IMC does not support merging coverage results that originate from different major versions of the Unicov library, `libucis.so`. Palladium uses the Unicov library from the Xcelium installation area. The major version of Xcelium used at run time must be consistent for all runs in order to merge results.

© 2021 All Rights Reserved.

Managing Data with IMC

The following table enumerates the basic features of IMC and their compatibility with a UXE Unicov database.

Batch Command	Support Information

`load`
`show`
`unload`
	Supported

`report -list`
`report -summary`
`report -detail`
	Supported for toggle and block coverage

`merge_config` `merge`	Supported when instance names are specified; Not supported when type names are specified.
`rank_config` `rank`	Supported when instance names are specified; Not supported when type names are specified.

`exclude`
`unexclude`
`load -refinement`
`save -refinement`
`source`
	Supported

`convert_icf`	Supported

`preferences`
`alias`
`unalias`
`sourcemap`
`history`
`help`
`exit`
	Supported

© 2021 All Rights Reserved.

Keep the following points in mind:

- Type-Based and Instance-Based Reports: The difference between type-based reporting and instance based is crucial to interpreting code coverage results. Type-based reporting accumulates the coverage results from each instance of a particular design unit into a single line item. Instance-based reporting lists each instance as a separate line item. Both type-based and instance-based reports are available via IMC.
- Type-Based Reports: Due to parameterization and other aspects that make design units unique, modules might appear multiple times in the type-based report. Each appearance represents a different compile-time configuration of that design unit. IMC does not summarize this information automatically when this occurs.
- Initial and Union resolutions: When two different coverage models are merged, IMC resolves the differences based on the specified initial model file or creates a union of the two model files. IMC is a powerful tool, which provides many options for merging data. Refer to the *IMC user guide* for more information.

Merging Emulation Coverage with Simulation Coverage Database

This section describes the process of merging the simulation (ICC) and emulation (UXE) coverage database in different scenarios. Toggle, block, assertion, and covergroup data can be merged between Palladium Z1 and Xcelium by using IMC. The process of merging data created by simulation with emulation is called cross-domain merging.

You can merge the UXE and ICC coverage databases in the following ways:

- Name-based merge - supported for toggle, assertion, and covergroups coverage

  Note: For details, refer to Covergroups, Toggle, and Assertions Merging - Name-based Merge section.

- Relaxed merge - supported for block coverage

  Note: Only a relaxed merge is supported for the coverage database. It is important that the coverage data being merged is for the same version of HDL source files. For details, refer to Block Merging - Relaxed Merge section.

© 2021 All Rights Reserved.

The following figure illustrates the typical cross-domain merge scenario in which multiple runs are being executed and the Unicov-based coverage data is being generated from both the simulation farm (using Xcelium) and the emulation farm (using Palladium Z1).

To create and load coverage databases, refer to the General Flow for Using Coverage and Generating a Coverage Database at Run Time sections. For details on loading, and merging coverage databases, refer to the *Integrated Coverage User Guide* available with the *Xcelium documentation set*.

To merge cross-domain (simulation and emulation) coverage databases (`icc_` to `uxe_`), use the following `merge` command in the IMC tool.

```
merge -cross_domain icc:uxe -initial_model <initial_model_path> \
      -out <output_test_dir> <vxe_test_dir> <icc_test_dir>
```

For example:

```
merge -cross_domain icc:uxe -initial_model dbIcc/scope/test \
      -out cov_work/scope/mDb cov_work/scope/uxetest dbIcc/scope/test
```


© 2021 All Rights Reserved.

Important

When merging cross-domain (simulation and emulation) coverage databases (`icc_` to `uxe_`), note that:

- It is necessary to use the `-cross_domain icc:uxe` option of the `merge` command in the IMC tool.
- For the `-initial_model` <*`initial model path`*> option, the <*`initial model path`*> is always the Xcelium (prefixed with `icc_`) model.

  For the relaxed cross-domain block merge, you can enable different design sections for the emulator and Xcelium by using constructs like `` `ifdef ``, and so on. However, as a result of the above limitation, coverage items that are only visible in Palladium (such as those from `` `ifdef ``) are not included in a cross-domain merge.

To know more about the syntax of the `merge` command, refer to the *Integrated Coverage User Guide* available with the *Xcelium documentation set*.

Covergroups, Toggle, and Assertions Merging - Name-based Merge

For merging covergroup, toggle, and assertions, name-based merging is used. Name-based merging is a merge algorithm that enables counters to combine when the hierarchical names and other labels or constants associated with the coverage item match. Name-based merging can be applied to toggle, assertion, and covergroup counters. Name-based merging only applies when performing a cross-domain merge.

Block Merging - Relaxed Merge

Block merging is a merge algorithm that enables counters to combine when source HDL information associated with the coverage item is a match. Block merging can be applied to block counters. Block merging only applies when performing a cross-domain merge. Block merging is considered a *relaxed merge* because it discards elaboration criteria to see if two given blocks are equivalent. As a result, it is imperative that a cross-domain merge that uses block merging, should be done only when both sets of databases originate from the same HDL source. Failure to do so might result in incorrect merging of coverage items that happen to match by chance.

Although it is a relaxed merging algorithm, ambiguous scenarios can result in blocks not matching. For example, if there are a number of processes written continuously on one line, but not all processes are supported by Palladium block coverage, the algorithm fails to merge these blocks because it cannot exactly match each one.

© 2021 All Rights Reserved.

For cross-domain merge, ensure that the coverage generated from Palladium Z1 and Xcelium is for the same design, and that the design has NOT been modified.

From IXCOM 20.12 release, along with Xcelium 20.09 version, the cross-domain merge for block coverage has been tuned to match the coverage contexts. Therefore, if the default behavior of IXCOM run does not match the Xcelium run, to ensure cross-domain merge, these runs should be made coverage-context equivalent.

For example, by default, IXCOM, in contrast to Xcelium, does not support coverage of implicit-if and implicit-case blocks by default. To make the IXCOM and Xcelium runs coverage-context equivalent in this case, rerun Xcelium with the following CCF commands:

```
set_implicit_block_scoring -off -if
set_implicit_block_scoring -off -case
```

After this, cross-domain merge should work.

Different Source Files Used on Palladium Z1 Emulator

Some of the design source files can be different across Palladium Z1 and Xcelium runs.

Consider the following case:

HDL Set 1 – `file1.v`, `file2.v`, `file3.v` - Xcelium coverage database

HDL Set 2 – `file1_synth.v`, `file2.v`, `file3.v` - Palladium Z1 coverage database

Here, `file1_synth.v` is used only for Palladium Z1 and `file1.v` is used only for Xcelium. While doing cross-domain merge in such a scenario, the block coverage data for items in `file1_synth.v` will not be merged in the output database because they do not exist in the simulation model. The block coverage items in `file2.v` and `file3.v` will be merged, and `file1_synth.v` will be flagged with a warning. Other coverage items like toggle and assertions will be merged because these are merged on hierarchical names.

© 2021 All Rights Reserved.

The following figure illustrates the scenario in which different source files of the design are enabled for the Xcelium simulator and Palladium Z1 emulator.

Different Versions Of Same Design (Recompile With Xcelium)

For cross-domain merging, the initial_model must be an Xcelium model. If the version of the source HDL changes, you must generate a new Xcelium coverage model for the new version of HDL to be used as the initial_model. Else, there is a risk of false positive if the parameters for relaxed merge like filename, line number, block type, and block index, and so on are found to be matching.

Suppose, three different versions of HDL - HDL_V1, HDL_V2, and HDL_V3, are fed into the Palladium Z1 and Xcelium farms. In this scenario, it is mandatory that:

- An initial_model is generated with Xcelium compile for each version of HDL.
- The coverage data being dumped out by emulation farm and simulation farm for each HDL version is merged first to create an Xcelium-compliant `icc` database. After that, the three Xcelium datasets can be merged together to get the final coverage numbers.

© 2021 All Rights Reserved.

The following figure illustrates the scenario in which different versions of the design are enabled for the Xcelium simulator and Palladium Z1 emulator.

Selective Enabling of HDL

For cross-domain block merging, you can choose to enable different design sections for Palladium Z1 and Xcelium by using constructs like `` `ifdef ``. Consider the following case:

HDL Set – `file1.v`, `file2.v` -- for both Palladium Z1 and Xcelium

Here, in `file1.v` a different design section is enabled using `` `ifdef SIMU ``. This is enabled only for Xcelium compilation. For Palladium Z1 compilation, on the other hand, the design section under `` `else `` is enabled.

© 2021 All Rights Reserved.

When a cross-domain merge is tried for databases created in this scenario, the design section within `` `else `` for Palladium Z1 is NOT merged because it is not visible in the `initial_model`.

The following figure illustrates the scenario in which different source files of the design are enabled for the Xcelium simulator and Palladium Z1 emulator.

Examples of Different Kinds of Merging of Simulation and Emulation Data

Following are some scenarios for merging Xcelium and VXE data.

Scenario 1: VXE database and Xcelium coverage data in different test directories

Example directory structure:

```
cov_work/scope/dbicc/icc_1a91d702_00000000.ucd
cov_work/scope/dbuxe/uxe_6d41aadc_00000000.ucd
cov_work/scope/icc_1a91d702_00000000.ucm
cov_work/scope/uxe_6d41aadc_00000000.ucm
```

Loading the database:

Each database can be loaded and viewed separately in the IMC tool.

Merging the databases:

```
merge <vxe_test_dir> <icc_test_dir> \
      -cross_domain icc:uxe \
      -initial_model <initial_model_path> \
```


© 2021 All Rights Reserved.

```
          -out <output_test_dir> \
          -message 1
```

Following is a sample command for merging the databases in the above-mentioned directories:

```
imc>merge dbicc dbuxe -initial_model dbicc -out mDb -overwrite \
          -cross_domain icc:uxe -message 1
```

In the above command:

- `dbicc` - Specifies the Xcelium-generated simulation coverage database.
- `dbuxe` - Specifies the VXE-generated emulation coverage database.
- `mDB` - Specifies the name for storing the merged database is to be stored.
- `-message 1` - Specifies that more details of the messages and conflicts should be generated while merging.

Scenario 2: VXE and Xcelium coverage databases available in the same test directory

Example directory structure: (`mytest`)

```
cov_work/scope/mytest/icc_1a91d702_00000000.ucd
cov_work/scope/mytest/uxe_6d41aadc_00000000.ucd
cov_work/scope/icc_1a91d702_00000000.ucm
cov_work/scope/uxe_6d41aadc_00000000.ucm
```

For details about loading the above databases, refer to the *IMC User guide*.

```
imc> load mytest
imc> report -summary -metrics code
name                Code Average                Code Covered
------------------------------------------------------------
top                    0.00%                       0.00% (0/1)
top                    0.00%                       0.00% (0/1)
```

Note: Two independent hierarchies are reported.

To merge the databases in the same directory, use the following command:

```
merge <input_test_dir> -initial_model <input_test_dir> \
      -out <output_test_dir> -cross_domain icc:uxe -message 1
```

Following is the sample command for merging the above-mentioned databases:

```
imc> merge mytest -initial_model  mytest -out mDb -overwrite \
           -cross_domain icc:uxe
```

In the above command:

- `mytest` - test directory where both the dbs to be merged are co-located.
- `mDB` - where merged db is to be stored.

© 2021 All Rights Reserved.

- `-message 1` - for more detailed messages and conflicts while merging.

Differences When Using IMC for VXE versus Xcelium

Following are the expected differences when using IMC with VXE as compared to its usage with Xcelium.

- The VXE statistics are submitted to Xcelium at the end of simulation during the final block processing. If you purposefully bypass the end-of-simulation processing, or abruptly exit the simulation, the coverage information will not be submitted to Xcelium. You must end the simulation with the `finish` or `exit` command, or by hitting a `$finish()` in the HDL.
- The Palladium Unicov database contains only hierarchical entries needed to represent coverage data. All other hierarchical components are omitted to improve performance. In software simulation, using Xcelium utilities, you can perform more complex merging operations for coverage. So, you might need to maintain the entire Xcelium hierarchy. By default, Xcelium creates the entire design hierarchy when creating a Unicov database. You can change this default behavior to match Palladium by using the `deselect_coverage -remove_empty_instances <covfile>` command.
- The IXCOM compiler might split a module into multiple variants to satisfy requirements like parameter uniquification and testbench-versus-DUT variations. Under these circumstances, the internally generated names are used for reporting. The internally generated names start with the original name. But these are suffixed with additional information. Refer to the explanation for the `-covMakeNamesUnique` option for details related to more control over the naming.

  Note: Xcelium has similar functionality, but only for modules in which the parameters directly affect covergroups.
- Functional Coverage Only:

  Coverage items in the testbench that are processed by IXCOM might be displayed with file and line information in the `xc_work/v` HDL. Use the `+xcDesignTop` option to ensure that IMC reports the correct file and line information.

  For performance reasons, IXCOM optimizes away coverage items that do not influence coverage calculations. Examples of such coverage items include empty covergroups and default bins. When these items are optimized away, an information message is generated during compilation.

© 2021 All Rights Reserved.

Limitations in Cross-Domain Merging

- For name-based merging, the hierarchal names in the two blocks to be merged must align. Assertions in VHDL with escaped labels are not merged because of the mismatched names.

  Blocks are not merged if there are multiple processes on the same line. For example, the following kind of blocks are not merged:

  ```
  initial a = 0; initial if(p == 0) d = 0; initial if(p == 1) e = 0;
  ```

  Consider the following HDL that has four processes on the same line, and each process has coverage blocks, on the same line:

  ```
  P1{b1, b2, b3}; P2{b1, b2, b3}; P3{b1, b2}; P4{b1, b2, b3};
  ```

  If `P3` process is NOT supported on the Palladium Z1 emulator, the VXE coverage database will have three processes for this line. The Xcelium coverage database, on the other hand, will have four processes on the same line. So, merging data for these processes or blocks is not possible. As a workaround, you can add a newline between these blocks to enable merging of the data counters.

  > **Note:** Multiple coverage blocks within a single process (for example `P1{b1, b2, b3}`) on the same line are merged.

- Data for any blocks that are unsupported by Palladium Z1, such as implicit blocks, is not generated, and merging does not happen.

- For cross-domain merge (using `merge` and `merge_config` commands), only the `initial_model` option is supported. Also, the `initial_model` must be an Xcelium model. Other modes like `union`, `primary`, `empty`, and so on are not supported.

Using vManager for Coverage Analysis

Using vManager, you can perform coverage-metric analysis by presenting data gathered by multiple agents (like the Palladium Z1 emulator, Xcelium Simulator and Specman) in the same window.

To use vManager for analyzing runs, you can opt for the following *methodologies* or *sessions*:

- Collecting Runs: vManager reads or loads a group of coverage databases from disk. The collection of coverage databases loaded is treated as one session. IXCOM supports this methodology.

- Importing Sessions: vManager reads VSOF files that have extended information about the run.

© 2021 All Rights Reserved.

- Launching Sessions: vManager reads a VSIF file that includes the attributes that specify the tests to be run, and the ways to run the tests in a vManager session. One of these attributes, `write_metrics` is enabled by default. Typically, this method uses a distributed resource management system, such as LSF, to manage different simulations.

Using vManager for Post-Processing Runs Analysis

The vManager and Palladium tools are integrated to help you use vManager for analyzing design runs previously executed on the Palladium Z1 emulator. You can run the design on Palladium Z1 emulator, without launching the run from vManager. Later, as a post-processing step after the run is complete, if required, you can analyze the coverage metrics of the VXE runs through vManager.

For this post-processing analysis of VXE runs through vManager, take the following steps:

1. Run designs on the Palladium Z1 emulator using the VXE software tools with coverage enabled. Use the `-write_metrics` option at run time to write the coverage database into a Verification Session Output File (`vsof`) file that contains the session information.
2. Invoke vManager tool by using the `vmanager -cs` command.
3. Load the VSOF file into vManager for coverage-metrics analysis.

Using Palladium-Generated VSOF Files

Each VSOF file contains pairs of attributes and values, such as log file and the location of the corresponding coverage database. You can load the VSOF file into vManager for coverage-metrics analysis.

Note: This option has no effect on a design that is not compiled with coverage.

You can choose to specify the `-write_metrics` option with any of the following commands:

- `xrun -R -write_metrics`
- `xrun -hw -write_metrics`
- `xeDebug --xmsim -write_metrics --`

  Note: Using the `-write_metrics` option with the IXCOM compiler (that is with the `ixcom` command) results in an error.

An alternate way to generate the VSOF file is to set the environment variable `VMANAGER_WRITE_METRICS` to `all`.

© 2021 All Rights Reserved.

In the emulation domain, the initial time required to load a design is significant and the time cost for a user to occupy a particular emulation machine is much higher than the time required on a software simulator. After occupying the emulator, a user would, typically, batch as many short tests into one session as possible, instead of wasting time in reloading the design. Therefore, multiple coverage databases are expected to be generated from a series of tests within one session. These are referred to as sub-sessions. In the simulation domain, such as Xcelium, a user can start a simulation job in effectively zero time. As a result, the concept of sub-sessions does not exist in Xcelium. On the other hand, in a Palladium run, multiple coverage databases might be generated in a single session run. For each coverage database, Palladium generates a corresponding VSOF (`uxe.vsof`) file to store the session information. Thus, multiple VSOFs might be generated in a single session of emulation.

Note that the distinction between a session and a sub-session in vManager is based on the number of coverage databases that are generated. The following figure depicts this scenario.

Each session can have multiple sub-sessions and each sub-session has its own coverage database and corresponding VSOF file. If three coverage databases are generated in a single session, the vManager GUI shows three runs. You can create new contexts on these three runs to perform individual coverage-metrics analysis or perform a coverage-metrics analysis on the merged coverage database.

While generating the VSOF files by using the -write_metrics option with the xrun or xeDebug commands on Palladium, keep the following limitation in mind:

Scanning of logs for multiple coverage databases in a single session: For each coverage database that is generated, vManager scans the logs to extract the various errors and notification messages. Xcelium creates a single log file that is scanned by

© 2021 All Rights Reserved.

vManager. In Palladium, however, the coverage database is generated multiple times within a single session, and, consequently, the scan of the log includes messages from the previous runs. If an error message is recorded during sub-session #5, it will be reported by vManager in sub-sessions #5, #6, #7, and so on.

By default, `xeDebug` suppresses the generation of the XLM log, that is `xrun.log`. To ensure that `xrun.log` is generated, explicitly pass the `-log` option to `xmsim` as follows:

```
xeDebug --xmsim -log xrun.log -write_metrics --
```


© 2021 All Rights Reserved.

Performance Considerations for Coverage Support

Coverage is defined by the following:

coverage=({number of items covered}/{number of items})

To achieve emulation run-time speeds, coverage instrumentation logic is used to track these values on the hardware. Before deciding if coverage can be used, it is important to determine the unused hardware capacity when your design is run on the Palladium Z1 emulator. In order to compile, enough emulator capacity needs to be available to fit the design under test, and any desired coverage instrumentation logic. You need to ensure that the number of items does not exceed emulator capacity. This is required for both functional and code coverage.

Performance Considerations for Code Coverage

When code-coverage instrumentation is inserted by the compiler into the design code:

- The code to be tracked for code coverage cannot be optimized by the compiler.
- The code-coverage instrumentation logic can possibly double the capacity of the area being covered.
- The code-coverage instrumentation logic is also added to the software model. This can cause a noticeable slowdown in software-simulation performance if the code coverage area is large.

Lack of optimization and increase in area make code coverage extremely expensive. Optimizations are done to keep this overhead as low as possible, but in general, it is still very expensive. Therefore, Cadence recommends that code coverage be limited to as small an area as possible.

For system-level verification, the most efficient use of code coverage resources is to focus the coverage on system-level interconnect logic, sub-system interfaces, and specific blocks of interest.

Before deciding to use code coverage, it is important that you ascertain the unused capacity while running the design on the Palladium Z1 emulator. Then, you can use the code-coverage options to control the portions of the design to be covered.

The best way to measure the space available on the emulator is by domains. For example, if the Palladium Z1 emulator you are using has 16 domains, and the design requires eight domains, then up to eight domains can be used for toggle coverage. The projected minimum number of domains is reported in the `ixcom.log` file.

© 2021 All Rights Reserved.

To determine the capacity associated with code coverage, consider the following table that lists expected coverage cost for a design that has coverage enabled across the entire DUT. A number is given for the typical capacity overhead up to a maximum value.

Coverage Type	Typical Increased Capacity as Compared to No Coverage	Maximum Increased Capacity compared to No Coverage
Toggle - ports (default)	1.5x	2x
Toggle - ports, variables	2x	3x
Toggle - ports, variables, nets	3x	4x
Block	1.2x	1.5x
Expression - Control	1.3x	1.4x

For a typical eight-domain design, for example, the expected cost of basic toggle coverage (ports only) will be around 4 - 8 more domains.

Caveats:

- The numbers in the table might change in future releases as the supported subset increases (negative impact) and as more optimizations are added (positive impact).
- Some toggle scenarios can result in higher than expected results if:
  - The design is a gate-level design. For this scenario, it is recommended that you remove the gate-level primitives that are from coverage consideration by using the `+ccovModelSummary` and `+ccovGenFeatureFile` options.
  - There is a high amount of concatenation and other expression operations in the port signals.

If there is not enough capacity on the emulator, the capacity cost of code coverage can be reduced by limiting the area of the design being covered. Explicitly mark the required modules and design units for code coverage to only track the areas of highest concern. Use the `+ccovModelSummary` and `+covGenFeatureFile` options to gather statistics about reducing coverage.

Managing Coverage Data

Cadence recommends that coverage data should be placed in its own directory away from the design. This is a common practice in simulation environments to ensure that all the

© 2021 All Rights Reserved.

coverage results can be easily accessed. Even if you only have one Palladium Z1 emulator, it is still recommended that you follow this practice.

This can be done by using the `-covworkdir` and `-covtest` options, which are natively included with the `xrun` command.

If you do not specify these options, the compiler works with the default values. The default directory in the case of both options is "." or the current directory. Using these options is useful for managing the data generated over multiple simulation runs.

You should explicitly specify these commands if you are trying to collect data over multiple runs, and do not want the data to be overwritten. By using these options, you can ensure that the files are not accidentally deleted if another user cleans up the design directory.

Note: The data will not be overwritten if you have not used the `-covOverwrite` switch.

Considerations Related to Coverage Verification on Palladium Z1 Emulator

Here are certain points to be kept in mind when running designs with coverage enabled.

Performance

The major attraction of enabling coverage on Palladium Z1 is to leverage the high capacity and run-time performance of the emulator. Minimal degradation in hardware performance at run time is expected when coverage is enabled. However, this is not true when the design is swapped out of the emulator. With all the instrumentation monitors introduced after coverage is enabled, the software-simulation environment is expected to be slower than with coverage disabled. Therefore, Cadence recommends that when you intend to enable coverage, swap to the software mode only for design initialization and debugging.

Merging Code Coverage and Functional Coverage

Code coverage can be used in conjunction with the functional coverage features provided by Palladium Z1. Both functional- and code-coverage databases are generated in the Unicov format that can be read by IMC.

Coverage generated by Xcelium is placed into an ICC Unicov database. Coverage generated by Palladium is placed in a UXE Unicov database that preserves the original source file information. When a single simulation run generates both of these databases, cross-domain merging can be used to combine them. Otherwise, two hierarchies are

© 2021 All Rights Reserved.

shown (one for simulation content and emulation content) when the coverage results are loaded by IMC.

Note: Do not confuse this merging of databases with the merging of simulation-based data and hardware-based data.

Code Coverage for Testbench Signals

The toggle coverage support with Palladium Z1 applies only to those design units targeted for the emulator. Design units that comprise the testbench or extracted hierarchies are not considered for code coverage. Compile-time options that include such design units are ignored by the IXCOM compiler with a warning message.

Passing Coverage Information Directly to Xcelium without IXCOM Processing

The Xcelium simulator uses a coverage configuration file to provide customizations regarding coverage instrumentation and reporting. All testbench coverage written into the ICC Unicov database is affected by this file. These settings mostly apply to testbench covergroups that are not accelerated. The following configuration setting can influence covergroup reporting, but ONLY when Palladium coverage is written out to an ICC database:

```
set_covergroup -per_instance_default_one
```

Note: The configuration file can be specified using the `-covfile` option of the `xrun` command. Refer to the *ICC User Guide* for more details about this option. To limit the `-covfile` options to just the ***xrun*** portion of the testbench, use the `-xmelabargs "-covfile <file>"` option of the `ixcom` command.
Avoid using any other options of the `set_covergroup` command because the resulting behavior might be unexpected with respect to the accelerated covergroups in the DUT. Coverage written out to the UXE Unicov database cannot be influenced by the Xcelium coverage configuration file. UXE Unicov assumes that `per_instance = 1` is enabled.

Design Units NOT Included in the Coverage Report

This section specifies the modules and design units that will not be included in the coverage report.

Coverage on Cell Libraries

Design units marked as cell libraries via `` `celldefine `` or `` `qtrefcell `` are not instrumented for code coverage. No warning message is issued. This is similar to Xcelium behavior.

© 2021 All Rights Reserved.

Code Coverage Limited to DUT on Palladium Z1

The primary purpose of code coverage is to ensure that there are no logically unreachable or untested areas of the design. In VXE, code coverage applies only to the design units that are targeted for the emulator. The design units that comprise the testbench or extracted hierarchies are not candidates for code coverage.

Note: If required, use the `+xcDesignTop` option of the `ixcom` command and compile the testbench to run in a separate simulation environment. Refer to Compiling Top-Level Testbenches Using +xcDesignTop for details. Note that when coverage is used, you can specify the `+xcDesignTop` option only once in an `ixcom` command.

Coverage on Hidden Modules

Design units that are protected are not included in the coverage report. Modules instantiated from a protected module are also not included in the coverage report. This is consistent with the behavior of the Xcelium tool.

Coverage within the Netlists

Design units within an imported netlist region are not instrumented. Design units that are in the exported netlist region can be instrumented and by using options, such as the `-featureFile` option.

© 2021 All Rights Reserved.

Quick Reference Guide for Covergroup Synthesis

This is a quick reference guide for building covergroups that can best be used in a Palladium emulator environment. The key difference between the Palladium emulator environment and a simulation environment is that the design under test must be statically allocated at compile time. As the coverpoints or cross bins increase, more Palladium resources will be consumed. This can result in a performance degradation, or the possibility of your design not fitting in your current Palladium configuration.

Making Covergroups Palladium-Compatible

Take the following actions to help ensure that covergroups are created in a Palladium-compatible manner.

Actions to be Taken	Reasons for these Actions
Make sure that all covergroup, cross, and coverpoint items have an explicit identifier.	If you do not specify an identifier, the compiler generates one for you. This can create cross-tool merging problems because different tools (or different versions of the same tool) can have different naming conventions.
Make sure the covergroup has an explicit clocking condition.	Covergroups on hardware run as a process. Specifying the clocking condition tells when this process should be scheduled. The `sample()` method is behavioral by design, and cannot be synthesized.
Avoid using the `sample()` method.	As mentioned above, the purpose of the `sample()` method is to provide behavioral level control as to when a covergroup gets sampled. Other than trivial cases, these cannot be synthesized without losing the behavioral control.

© 2021 All Rights Reserved.

Actions to be Taken	Reasons for these Actions
Make sure to use statically created covergroups.	Unlike testbench covergroups, synthesized covergroups must be static. If some level of timing control is needed, use the `start()` and `stop()` methods to disable sampling during reset periods.
When creating covergroups, consider the usage of the covergroup beyond the immediate usage. Use `` `ifdef `` or `bind` directives to ensure that other members of your team can enable and disable covergroups based on verification level.	If you are creating covergroups while trying to verify a design block, the covergroup has a specific purpose. That purpose may or may not make sense when running the larger sub-system or overall chip. By identifying the "life-time" of the covergroup, you can convey the usefulness of this covergroup to another user.
Keep the number of coverpoint bins as small as possible.	It is easy to create a coverpoint that tracks a large number of values, but it has "*costs*" further down stream. Large numbers of bins will be reflected in the overall capacity associated with the Palladium, increase your coverage data files, and can "*overshadow*" other covergroups that might be better designed.
Avoid using illegal bins.	Illegal bins are supported well on Palladium. But, often, SystemVerilog assertions are a better choice because run-time and debug environments have extensive debug features that are built around assertions.
Consider using SystemVerilog 2012-specific features	The new covergroup features in SystemVerilog 2012 are still in the process of being supported on Palladium. If you want to use specific constructs, let the Cadence Support engineer know so that these constructs can be appropriately priortized in future Palladium releases.
Check all the uses of wildcard bins to ensure that the resulting set of bins does not explode.	Wildcard bins are a syntactic element that can easily translate into a large number of bins. This can have an adverse performance impact on the Palladium emulator.

© 2021 All Rights Reserved.

Choosing Between Covergroups and Cover Directives

Creating a covergroup using assertion syntax, typically, requires a custom script for generating cover directives to achieve equivalent counters. Using cover directives can have the following problems:

- To create a covergroup using assertion syntax, typically required a script for generating cover directives to achieve equivalent counters. Contrastingly, covergroup syntax is more compact and easier to maintain.
- The instrumentation associated with assertions typically has one counter per assertion. Instrumentation for covergroups can leverage one counter per coverpoint or cross. Covergroups typically have a lower capacity cost than the equivalent assertions.
- Using many cover directives can create a problem during the analysis stage. Independently created assertions lose the scoping capabilities that a covergroup provides. One alternative is to put all the related assertions in a sub-scope. However, covergroups have many built-in features to control weights, goals, and minimum hit counters etc.

The above-mentioned problems do not imply that you should always use a covergroup. Cover directives have the following advantages:

- Cover directives provide better support for handling sequences of events.
- Cover directives support complex antecedents.

To summarize, Palladium supports both covergroups and cover directives. Based on the information given above, you can choose the right one for your designs. If you are still unsure after reading the above points, choose the one that is easiest to write and maintain.

© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

22

Modeling Clocks in SA Mode

This chapter~~appendix~~ provides the guidelines for modeling clocks in SA mode. For reviewing concepts related to modeling clocks in ICE mode, refer to Modeling Clocks in ICE Mode. The Modeling Clocks in ICE Mode chapter also provides details on basic concepts related modeling clocks, such as Supported Clock Technologies, Using the 1X Mode, and Using the CAKE 0.5X Mode etc.

This chapter provides details only for modeling~~doeling~~ clocks in SA mode. To model clocks in SA mode, you can opt to use the *ixclkgen* utility, which enables automatic generation of the clocks or manually generate the clocks by using the Verilog delay control statements.

These two methods are described in the following sections:

- Manually Modeling Clocks in SA Mode
- Using userData to Create a Clock-Generation Module for SA Mode
- Oversampling in SA Mode

Manually Modeling Clocks in SA Mode

This section describes how Verilog delay control statement (`#delay`) can be directly used to model clock generation in the DUT using the IXCOM compiler.

IXCOM supports clock-generation modeling with the `always` procedure as well as `initial` procedure with a `forever` loop. For both modeling styles, the delay control statement (`#delay`) is used to specify the phase delays of the clock.

The following types of delays are supported.

- Non-zero constant delay control
  - Integer constants, such as `#1`, `#20`
  - Real constants, such as `#1.2`, `#0.5`
  - Time constants, such as `#5ns`, `#10fs`

© 2021 All Rights Reserved.

For example:

```
always #1 clk1 = ~clk1;
always begin
    clk2 = 0; #0.5;
    clk2 = 1; #1.5;
end
initial forever #5ns clk3 = ~clk3;
```

- Variable delay control

  `#Td` where `Td` is a net or variable of integer data types

  For example:

```
reg [3:0] Td1;
reg [3:0] Td2;
initial begin
    Td1 = 3; Td2 = 3;
    clk4 = 0;
    forever begin
        #Td1; clk4 = 1;
        #Td2; clk4 = 0;
        // conditionally increment delay to rising edge
        if (Td1 < 10) Td1 = Td1 + 1;
    end
end
```

In addition to delay control statements, the following non-time-consuming Verilog statements are allowed in clock modeling:

- Blocking and non-blocking procedural assignments without intra-assignment delay
  - `clk1 = 0;`
  - `clk2 <= 1;`
- Conditional statements containing no delay control statements
  - `if (condition)...`
  - `if (condition)... else ...`
- Case statements containing no delay control statements
  - `case (expression)...`
  - `casez (expression)...`
  - `casex (expression)...`

© 2021 All Rights Reserved.

Enabling Clock Model Compilation

To compile clock models described using delay control statement (`#delay`) in a module in the DUT, interface compilation for delay needs to be enabled by using either of the following options:

- Specify the module with the `+iscDelay+<mod_name>` option of the `ixcom` command. The `<mod_name>` refers to the name of the DUT module that you have defined.

  For example:

  ```
  ixcom -ua clkgen.v +dut+clkgen +iscDelay+clkgen ...
  ```

  In the above command, `clkgen` is the name of the module in the DUT.

- Add the following `map_delays` directive in an `initial` procedure in the module containing the clock model:

  ```
  $ixc_ctrl("map_delays");
  ```

Note: IXCOM generates a compilation error if interface compilation for delay is enabled but no valid clock model is found under the specified modules in the DUT.

Modeling Clocks without Initial Delay

Here is a waveform to illustrate clocks without initial delay:

To model a clock without initial delay as shown in the waveform above, the following clock model templates are supported.

- Clock model with `always` procedure

  ```
  always begin
        [#Td; <non-time-consuming_statements>;]+
  end
  ```

  For example:

  ```
  reg clk = 0;
  reg [3:0] td1 = 5;
  reg [3:0] td2 = 10;
  always begin
        #td1; clk = 1;
        #td2; clk = 0;
  end
  ```


© 2021 All Rights Reserved.

- Clock model with a `forever` loop in `initial` procedure

```
initial begin
       [<non-time-consuming_statements>;]*
       forever begin
              [#Td; <non-time-consuming_statements>;]+
       end
end
```

For example:

```
initial begin
     // initialize clk and phase delays td1 and td2
     clk = 0;
     td1 = 5; td2 = 10;
     forever begin
          #td1; clk = 1;
          #td2; clk = 0;
     end
end
```

Modeling Clocks with Initial Delay

The following waveforms illustrate clocks with initial delay:

To model a clock with an initial phase delay as shown in the waveforms above, the following clock model templates are supported.

- Clock model with `always` procedure

```
initial td = <initial_delay_to_first_clock_edge>;
always begin
      #td; <first_clock_edge>;
      #<regular_delay_to_second_clock_edge>; <second_clock_edge>;
      td = <regular_delay_to_frist_clock_edge>;
end
```

For example:

```
reg clk = 0;
reg [3:0] td0 = 20;
reg [3:0] td1 = 5;
```


© 2021 All Rights Reserved.

```
reg [3:0] td2 = 10;
reg [3:0] td;
initial td = td0;
always begin
      #td; clk = 1;
      #td2; clk = 0;
      td = td1;
end
```

- Clock model with a `forever` loop in `initial` procedure

```
initial begin
      [non-time-consuming statements;]*
      #<initial_delay_to_first_edge>;
      forever begin
            <first_clock_edge>;
            #<regular_delay_to_second_clock_edge>;
            <second_clock_edge>;
            #<regular_delay_to_frist_clock_edge>;
      end
end
```

For example:

```
initial begin
      // initialize clk and phase delays td0, td1 and td2
      clk = 0;
      td0 = 20; td1 = 5; td2 = 10;
      #td0;
      forever begin
            clk = 1;
            #td2;
            clk = 0;
            #td1;
      end
end
```

Modeling Clocks with Variable Phase Delay

To model a clock with variable phase delay, use a variable to capture the phase delay and change it conditionally or unconditionally inside the clock model. All the clock model templates presented in Manually Modeling Clocks in SA Mode support variable phase delay.

- Example of clock model using `always` procedure with conditional statement

```
module clkgen(clk, change_delay, new_delay);
output clk;
input change_delay;
input [3:0] new_delay;
reg clk = 0;
reg [3:0] td = 10; // initialize td to first delay;
always begin
      #td; clk = ~clk;
      if (change_delay) begin
            td = new_delay;
      end
end
endmodule
```


© 2021 All Rights Reserved.

- Example of clock model using a `forever` loop in `initial` procedure with `case` statement

```
module clkgen(clk, rising_delay, falling_delay, d1, d2, d3);
output clk;
input [1:0] rising_delay, falling_delay;
input [3:0] d1, d2, d3;
reg clk;
reg [3:0] td1, td2;
initial begin
      // initialize clk and rising/falling delays td1/td2
      clk = 0;
      td1 = 5; td2 = 10;
      forever begin
            #td1; clk = 1;
            case (falling_delay)
                  0: td2 = d1;
                  1: td2 = d2;
                  2: td2 = d3;
                  //3: td2 stays the same
            endcase
            #td2; clk = 0;
            case (rising_delay)
                  0: td1 = d1;
                  1: td1 = d2;
                  2: td1 = d3;
                  //3: td1 stays the same
            endcase
      end
end
endmodule
```

IXCOM Messages Related to Clock Model Compilation

Following is a list of messages related to clock-model compilation, generated during IXCOM interface compilation for delay:

- `ISCDELAY_IGNORE_INITIAL_CLOCK`
  - Warning message: "`Blocking delay in initial block does not match supported clock models`"
  - Impact: Delay control not matching supported clock models would stay in the `initial` block, possibly preventing swapping into emulator until the simulator completes execution of the specified delay.
- `ISCDELAY_INITIAL_CLOCK_FOUND`
  - Info message: "`Found clock specification in initial block`"
  - Impact: Clock model in indicated `initial` block is successfully identified and extracted to an `always` block for execution in the emulator.
- `ISCDELAY_CLOCK_FOUND`

© 2021 All Rights Reserved.

- Info message: "`Found clock specification in always block`"
- Impact: Clock model in indicated `always` block is successfully identified and converted to clock-generation logic for the emulator.

  This includes always block extracted from initial block `[ISCDELAY_INITIAL_CLOCK_FOUND]`.

■ `ISCDELAY_INITIAL_CLOCK_MISSING`

- Warning message: "`Clock specification extracted from initial block is missing during transformation`"
- Impact: Clock model extracted from `initial` block is missing. No clock-generation logic is created.

When a clock model specified in an `always` procedure has been successfully transformed to clock-generation logic, a corresponding `ISCDELAY_CLOCK_FOUND` message is displayed.

When a clock model specified with a `forever` loop in an `initial` procedure has been successfully transformed to clock-generation logic, both `ISCDELAY_INITIAL_CLOCK_FOUND` and `ISCDELAY_CLOCK_FOUND` messages are displayed.

The first message indicates that the clock specification has been successfully identified in an `initial` procedure and has been extracted as an `always` clock specification for hardware execution. The second message indicates that the extracted `always` clock specification has been successfully transformed to clock-generation logic.

Dynamic Clock Modeling

Dynamic clock frequency and shape modeling can be carried out by first specifying the variable delays in the code, and then using the `force` command on the `xe>` prompt to change the delay value before the simulation starts. The delay value can also be changed in the middle of the simulation, before and after swapping the design between hardware and software.

For example:

```
reg clk = 0;
reg [7:0] hT = 5;
reg [7:0] lT = 3;
always begin
   #hT clk = ~clk;
   #lT clk = ~clk;
end
```

You can change the frequency and the shape of this clock at the `xe>` command prompt:

© 2021 All Rights Reserved.

```
xe> force hT 7
```

Using Clock Generators with VHDL

To program one or more clocks in a VHDL design, write a Verilog module for clock generation, and then instantiate that module in the VHDL DUT. The Verilog module for clock generation is given below:

```
`timescale 1ns/1ns
module clk_gen(vlg_clk);
output reg vlg_clk = 0; //clock is output to VHDL
always #5 vlg_clk = ~vlg_clk;
endmodule
```

Then, instantiate the Verilog module in the DUT as follows:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity DUT is -- note: DUT does not have ports for emulation
end DUT;

architecture RTL of DUT is
component clk_gen
port (vlg_clk : out std_logic);
end component;
signal sys_clk : std_logic;
begin

   clk_gen1 : clk_gen
   port map (vlg_clk => sys_clk);

   . . .
end RTL;
```

Analyze and compile the design as follows:

```
vlan -work work clk_gen.v
vhan -work work clk_gen.vhd
vhan -work work DUT.vhd [other_options]

ixcom -top DUT -ua +dut+DUT +iscDelay+clk_gen
```

Handling Single-Edged Clocks

If the IXCOM compiler detects a primary input (clock) active on a single edge (posedge-only or negedge-only), the compiler skips the hardware evaluation of the unused edge. This can result in a performance improvement of nearly 2X for some designs.

Keep in mind the following implications of the above performance enhancement measure:

- The clock waveform from hardware probes only shows a zero-delay pulse at the rising (or falling) edge of the original clock signal. The waveform does not show changes for all the DUT signals at the falling (or rising) edge of the clock signal. The `-event` option of the `probe` command needs to be used.

© 2021 All Rights Reserved.

- In xeDebug, the `value` command for the clock signal returns `0` (or `1`) all the time.

Tip

For the list of single-edged clock inputs optimized by the compiler, in the `tmp/precompile.msg` file, search for `PECLK` or `NECLK`.

Performance Considerations for Modeling Clocks in SA Mode

All the clocks should be either in the DUT or testbench partition, but not in both partitions. When clocks exist in both hardware and software partitions, clock synchronization overheads adversely impact the performance.

If the clocks need to exist in the hardware as well as software partition, for better performance, generate all clocks in the DUT (or hardware side):

Use the hardware-modeling methods as defined above to model clock generators in DUT. You can send the DUT-generated clock to testbench through output ports or cross-module reference. You also need to use the `$export_event` directive as shown below:

```
initial $export_event(dut.to_TB_cllk):
```

The `$export_event` directive exports a DUT clock to testbench so that IXCOM compiler can send out the clock event accurately from the DUT to the testbench to preserve the functionality.

In this scenario, you can use `xc on -tbrun` or `xc on -run` modes without incurring any expensive clock synchronization overheads because the timed event is in hardware partition.

Synchronizing the Fastest Design Clock with Emulator FCLK

On the emulator side, 1X mode is a compilation mode that allows the fastest design clock to run at the same frequency as the emulator's FCLK. Without the use of 1X mode, the fastest design clock would run at a speed at least two times slower than FCLK.

In SA mode, using the 1X mode improves the performance of Palladium Z1, but it might increase capacity requirements. To ensure the usage of 1X mode, use the `+1xua` option with the `ixcom` command. Note that IXCOM passes this option automatically to the back-end compiler during the pre-compilation state. You must not pass this option through the `compilerOptions.qel` file.

The 1X mode can be invoked in both single-step and multi-step compilation.

© 2021 All Rights Reserved.

Multi-Step Compilation with +1xua Option

For a mixed-HDL or VHDL design, the multi-step compilation process needs to be used, involving a combination of the `vlan`, `vhan`, and `ixcom` commands.

The `+1xua` option must be used with the `ixcom` command.

1. Analyze the VHDL and Verilog parts of the design separately, using the `vlan` and `vhan` commands.

   ```
   vlan ${IXCOM_INSTALL_HOME}/share/vxe/etc/ixcom/IXCclkgen.sv
   ```

2. Compile the entire design to generate the simulation snapshot using the `ixcom` command.

   ```
   ixcom -ua +1xua -top <top_module>
   ```

Single-Step Compilation with +1xua Option

For a Verilog-only Design, the single-step compilation process is used to generate the simulation snapshot, involving only the `ixcom` command. Separate `vlan` and `vhan` steps are not required.

```
ixcom -ua +1xua <all_source_files> -v ${IXCOM_INSTALL_HOME}/share/vxe/etc/
ixcom/IXCclkgen.sv
```

Using the 1X Mode Clock Generator Library: IXCclkgen.sv

The `${IXCOM_INSTALL_HOME}/share/vxe/etc/ixcom/IXCclkgen.sv` library file is sourced for the RTL modules required for generating the clocks for 1X mode.

Using userData to Create a Clock-Generation Module for SA Mode

This section describes how ICE clocks can be automatically generated while using the IXCOM compiler in SA mode by using the *ixclkgen* utility.

The automatic generation of the ICE clocks is done by a standalone utility called *ixclkgen*, which is provided in the IXCOM~~VXE~~ installation directory to generate clocks from User Data. This utility accept the XEL User Data as input, and generates a Verilog module containing the modeled clocks.

In ICE mode, the design clocks are specified using the XEL User Data commands instead of through the HDL description. Then, the ICE compiler automatically generates the clocks for the design based on the User Data specification, such as the following:

- Clock-generation technology (edge alignment) and sampling

© 2021 All Rights Reserved.

- Clock specification using relative frequency
- Automatic driver-logic insertion to disconnect the original driver for the clock source in the design

In SA mode, you need to specify the design clocks by using the RTL model. You can either describe the clocks using always blocks or use the *clock-generation module(s)* provided as a part of the IXCOM software in the installation directory (`<IXCOM_INSTALL_HOME>/share/vxe/etc/ixcom/IXCclkgen.sv`) to model ICE-clock-generation technology. The modeled clock(s) are compiled with the original RTL as a part of a side file (in Verilog or System Verilog).

The *ixclkgen* utility supports the ICE-clock generation and enables you to:

- Specify the clocking information using ICE User Data (or command) syntax.
- Generate uncontrolled (or free running) clocks as in ICE mode.
- Automatically generate the Verilog module to model clocks for the information that you specified by using the ICE User Data as described in Clock Module Generation by ixclkgen Utility on page 1121.

To use the *ixclkgen* utility, use the following command:

```
ixclkgen -input <file> \
         -output <file>\
         [ -controlled <clock_source>[,<clock_source> ...] ] \
         [ -default <controlled | uncontrolled> ] \
         [ -external_netlist <string> ] \
         [ -help ] \
         [ -hierarchy <string> ] \
         [ -module <name> ] \
         [ -rtc ] \
         [ -spd ] \
         [ -uncontrolled <clock_source>[,<clock_source> ...] ]
```

Refer to the description of the `ixclkgen` command in the *Commands and System Tasks for SA Mode* chapter of the *VXE Command Reference Manual* for details of the options of the `ixclkgen` command. Some options with their descriptions in brief are listed in the following table.

`-input <file>`	Specifies the file that contains the clocking User Data in XEL format. See Specifying Clocking Information for a list of the User Data that can be specified to *ixclkgen*.
`-output <file>`	Specifies the output filename for the generated clock module.
`-controlled <clock_sources>`	Specifies the controlled clock sources as comma-separated names.

© 2021 All Rights Reserved.

`-input` <*file*>	Specifies the file that contains the clocking User Data in XEL format. See Specifying Clocking Information for a list of the User Data that can be specified to *ixclkgen*.
`-default` <`controlled` | `uncontrolled`>	Specifies the default clock-generation behavior for the clock sources not listed explicitly using the `-controlled` or `-uncontrolled` option. The default behavior for the clock source is controlled.
`-hierarchy` <*string*>	Specifies the hierarchical path for the clock signals referred in the User Data.
`-module` <*name*>	Specifies the name of the Verilog module for the clock-generation module. If you do not specify this option, the *ixclkgen* utility uses the default module name, which is `_ixc_clkgen`.
`-rtc`	Makes it possible to change the frequency of the fastest clock at run time using `clockConfig` command.
`-uncontrolled` <*clock_sources*>	Specifies the uncontrolled or free-running clock sources as comma-separated names. The default behavior for the clock source is controlled.

The generated Verilog module uses the clock-generation modules defined in the following directory:

```
<XE_INSTALL_DIRIXCOM_INSTALL_HOME>/share/vxe/etc/ixcom/IXCclkgen.sv
```

You need to compile the generated Verilog module file with the original RTL source files by using the IXCOM tools (`vlan`, `vhan`, `ixcom`) either in single-step or multi-step compilations as shown below:

- To generate and compile the clock module as the top-level DUT design unit in a single-step IXCOM compile, use the following command:

```
ixclkgen -input ud.qel -output clk.sv -module Clk
ixcom <source_files_and_options> -ua +dut+Clk clk.sv \
       -v <XE_INSTALL_DIR>/share/vxe/etc/ixcom/IXCclkgen.sv \
       -timescale <unit_or_precision>
```

- To generate and compile the clock module as the top-level DUT design unit in multi-step IXCOM compile, use the following command:

```
ixclkgen -input ud.qel -output clk.sv
vlan <source_files_and_options>
vlan [options] clk.sv -v <XE_INSTALL_DIR>/share/vxe/etc/ixcom/IXCclkgen.sv
ixcom [options] -ua +dut+_ixc_clkgen -top _ixc_clkgen -top <du> \
      -timescale <unit_or_precision>
```

Use `ixclkgen -help` for more examples with additional options to the utility.

© 2021 All Rights Reserved.

The *ixclkgen* utlity can also be invoked using the `-ixclkgen` option of the `ixcom` command. See the description of the `-ixclkgen` option in the *VXE Command Reference manual* for more details.

The *ixclkgen* utility inserts the timescale directive into the clock-generation module. The IXCOM compiler follows the language standard to process the `timescale` directive. Both single-step and multi-step compilation require explicit specification of the `timescale` for the design. For single-step compilation, specify the generated clock-generation module as the last source file. To use CAKE 1 clocking technology, specify `+1xua` option with the `ixcom` command.

Specifying Clocking Information

The syntax and format for specifying the XEL User Data (or command) is the same as in ICE mode. The following subset of XEL User Data (commands) are supported for specifying clocking information when using the *ixclkgen* utility:

- `clockOption`: Specifies the clock-generation technology (CAKE or SCM) and oversampling.

```
clockOption -add { technology CAKE N } // CAKE oversampling value N
                                       // specified must be 1 or higher.

clockOption -add { activeEdge both | positive | negative } // Supported

clockOption -add { ignoreOutput on | off  } // Supported, but ignored by
the                                         // ixclkgen utility with an
                                            // information message

clockOption -add { ignoreEvent on | off } // Supported, but ignored by the
                                          // ixclkgen utility with an
                                          // information message
```

- `clockSource`: Selects the net (or signal) in the design as clock source. This User Data also disconnects the original driver logic on the net (or signal) when driving with the CAKE-or SCM-generated clock.

```
clockSource -add <clock_source> // Supported
clockSource -rm <clock_source> // Supported
```

Note: This User Data has no effect on the generated code.

```
primaryClockSource -add <clock_source> // Supported
primaryClocksource -rm <clock_source> // Supported
```

Note: This User Data has no effect on the generated code. The `primaryClockSource` command enables you to specify the fastest CAKE clock without needing to check its frequency. If the frequency of the specified clock source is not the highest, the precompiler automatically raises the frequency as required.

© 2021 All Rights Reserved.

If you specify clockSource without clockFrequency, the net would be a non-instrumented clock. This means that this net is driven normally from the netlist, but it is considered a clock for loop-breaking purposes (paths from this clock should not be broken).

Note that the clockSource command is not required if you have specified the clockFrequency command.

- clockFrequency: Specifies frequency for the given clock source

```
clockFrequency -add { <clock_source> <frequency> } // Supported
clockFrequency -rm <clock_source> //Supported
```

- clockAssign: Specifies the derivation of the behavior of one signal from the behavior of another signal

```
clockAssign -add  { <clock_signal_1>  [ ~ ] // Supported
                    <clock_signal_2>  [ N ] } // Supported
```

- clockDelay: Adds, removes, or modifies the clock delay value, which can be a whole number, greater or equal to zero. <*nDly*> is interpreted as the delay in FCLK cycles). You cannot use this command if the design is compiled for 1X mode.

```
clockDelay -add { <clock_source> <nDly> //Supported
clockDelay -rm { <clock_source> } //Supported
```

The *ixclkgen* utility supports the following ICE-mode features:

- Hierarchical name is allowed for clock signals or nets referred in clockSource, primaryClockSource, clockFrequency, clockAssign, and clockDelay User Data.
- XEL-style comment lines can be specified using the # symbol.
- XEL-style double-quote (“”)can be specified as delimiters in place of open-braces and close-braces.

The following limitations apply to the content in the input XEL file:

- IXCOM does not support CAKE clocking with 0.5x oversampling or SCM technology. The User Data specification for this clocking technology is reported as unsupported.
- For all User Data, which is valid, but is unsupported and, therefore, ignored by the *ixclkgen* utility, the utility writes out the User Data into a file named compilerOptions_ixclkgen.qel. You can include or source this file from the compilerOptions.qel file, which is currently passed with the ixcom command during the IXCOM compilation process.

© 2021 All Rights Reserved.

Clock Module Generation by ixclkgen Utility

The *ixclkgen* utility processes the User Data commands and generates a Verilog module with the modeled clocks to be compiled with the IXCOM compiler. These steps include:

- Determining the clocking technology and oversampling from the input User Data (`clockOption`).
- Converting the clock-frequency value to phase-length values and scale to module timescale.
- Collecting the clocking delay values specified.
- Mapping the information specified above to generate clocks IXCOM primitives.
- Generating the HDL to disconnect the driver logic for the clock source and drive the original clock source with the generated clock.
- Generating the HDL for clock assign (if the `clockAssign` User Data is specified).
- Generating the information about clocks for use at run time.
- Generating the clock-bind statements to connect the clock signals in the external netlist module hierarchy (if the `-external_netlist` option is specified with the `ixclkgen` command)

The following sections describe the clock-module generation by the `ixclkgen` utility for different User Data command sets.

For all the modules and primitives mentioned in the following examples, refer to the following file:

<~~XE_INSTALL_DIR~~IXCOM_INSTALL_HOME>/share/vxe/etc/ixcom/IXCclkgen.sv

Using CAKE 1X Technology

The *ixclkgen* utility generates the CAKE 1X clocks if the input file contains the following set of User Data:

```
clockOption -add { technology CAKE 1 }
clockFrequency -add { <clock_source> <frequency> }
```

Following is an illustration of the generation of the CAKE 1X clock module using a sample set of User Data:

Command:

```
ixclkgen -input clock-ud.qel -output clk.sv
```

Input File:

© 2021 All Rights Reserved.

```
# File: clock-ud.qel

clockOption -add { technology CAKE 1 }

clockFrequency -add { dut.clkA 500 MHz }
clockFrequency -add { dut.clkB 300 MHz }
```

Output File:

```
// This is an auto-generated file using the ixclkgen utility.
//   Input  File: clock-ud.qel
//   Output File: clk.sv
// NOTE: Please refer to VXE user guide for clock modeling in IXCOM. Try:
//   ixclkgen -help
//   for usage and quick guide to compiling generated clock module.
//////////////////////////////////////////////////////////////////////////////

`timescale 1 ns / 1 ns
`define IXCclkgenTs 1 ns / 1 ns
module _ixc_clkgen;
  wire clkA_0;
  wire clkB_1;

   initial $ixc_ctrl("map_delays");

  // Generate logic for clock sources
  IXCclkgen  #(1) ixcg_0 (clkA_0, );
  IXCcake1x  #(300, 500, 0) ixcg_1 (clkB_1, clkA_0);

  // Bind clock sources to generated clock signal
  ixc_cakebind ixcb_0 (dut.clkA, clkA_0);
  ixc_cakebind ixcb_1 (dut.clkB, clkB_1);
  // ...
endmodule
```

Important

If the clocking technology is not specified, by default the technology is set to CAKE 2X.

Using CAKE 2X Technology

The *ixclkgen* utility generates CAKE 2X clocks if the input file contains the following set of User Data:

```
clockOption -add { technology CAKE N } # N = 2
clockFrequency -add { <clock_source> <frequency> }
```

Following is an illustration of the generation of the CAKE 2X clock module using a sample set of User Data:

Command:

```
ixclkgen -input clock-ud.qel -output clk.sv
```

Input File:

```
# File: clock-ud.qel
```


© 2021 All Rights Reserved.

```
clockOption -add { technology CAKE 2 }

clockFrequency -add { dut.clkA 500 MHz }
clockFrequency -add { dut.clkB 300 MHz }
```

Output File:

```
// This is an auto-generated file using the ixclkgen utility.
// Input  File: clock-ud.qel
// Output File: clk.sv
// NOTE: Please refer to VXE user guide for clock modeling in IXCOM. Try:
//   ixclkgen -help
//   for usage and quick guide to compiling generated clock module.
//////////////////////////////////////////////////////////////////////////////

`timescale 1 ns / 1 ns
`define IXCclkgenTs 1 ns / 1 ns
module _ixc_clkgen;
  wire clkA_0;
  wire clkB_1;

  initial $ixc_ctrl("map_delays");

  // Generate logic for clock sources
  IXCclkgen  #(1) ixcg_0 (clkA_0, );
  IXCcake2x  #(300, 500, 0) ixcg_1 (clkB_1, clkA_0);

  // Bind clock sources to generated clock signal
  ixc_cakebind ixcb_0 (dut.clkA, clkA_0);
  ixc_cakebind ixcb_1 (dut.clkB, clkB_1);
  // ...
endmodule
```

Note: In SA mode, the IXCOM compiler maps the higher oversampling value (CAKE N > 2) to CAKE 2X.

Using SCM Technology

In ICE mode, the following User Data commands are used to generate SCM clocks. However, because SCM is not supported by the *ixclkgen* utility, it reports an error for the following User Data.

```
clockOption -add { technology SCM }
```

Specifying the Active Clock Edge

The *ixclkgen* utility supports active-edge specification using the following User Data:

```
clockOption -add { activeEdge both | positive | negative }
```

This User Data has an effect only with CAKE 1X technology.

- If the above User Data is not specified, by default, the active edge is set to `positive`.

© 2021 All Rights Reserved.

- Just like in ICE mode, specifying the active edge as `both` has the same effect as specifying `positive` active edge.
- If active edge is specified as negative, the *ixclkgen* utility uses the `IXCcake1xN` primitive as shown in the example below.

Command:

```
ixclkgen -input clock-ud.qel -output clk.sv
```

Input File:

```
# File: clock-ud.qel

clockOption -add { technology CAKE 1 }

clockOption -add { activeEdge negative }

clockFrequency -add { dut.clkA 500 MHz }
clockFrequency -add { dut.clkB 300 MHz }
```

Output File:

```
// This is an auto-generated file using the ixclkgen utility.
//   Input  File: clock-ud.qel
//   Output File: clk.sv
// NOTE: Please refer to VXE user guide for clock modeling in IXCOM. Try:
//   ixclkgen -help
//   for usage and quick guide to compiling generated clock module.
//////////////////////////////////////////////////////////////////////////////
`timescale 1 ns / 1 ns
`define IXCclkgenTs 1 ns / 1 ns
module _ixc_clkgen;
wire clkA_0;
wire clkB_1;

  initial $ixc_ctrl("map_delays");

  // Generate logic for clock sources
  IXCclkgen  #(1) ixcg_0 (clkA_0, );
  IXCcake1xN #(300, 500, 0) ixcg_1 (clkB_1, clkA_0);

  // ...
  // See Using CAKE 1X Technology for the rest of the module content.
endmodule
```

Specifying Clock Delays

Clock-delay specification is supported using the following User Data syntax:

```
clockDelay -add { <clock_source> <delay> }
clockDelay -rm  { <clock_source> }
```

In the generated clock module, the delay value is applied to the clock signal as a parameter in the `IXCcake1X`, `IXCcake1N`, and `IXCcake2x` primitive instances for the corresponding clock signal.

The following example shows the generated clock module with delay specification:

© 2021 All Rights Reserved.

Command:

```
ixclkgen -input clock-ud.qel -output clk.sv
```

Input File:

```
# File: clock-ud.qel

clockOption -add { technology CAKE 2 }

clockFrequency -add { dut.clkA 500 MHz }
clockFrequency -add { dut.clkB 300 MHz }

clockDelay -add { dut.clkB 25 }
```

Output File:

```
// This is an auto-generated file using the ixclkgen utility.
//   Input  File: clock-ud.qel
//   Output File: clk.sv
// NOTE: Please refer to VXE user guide for clock modeling in IXCOM. Try:
//   ixclkgen -help
//   for usage and quick guide to compiling generated clock module.
//////////////////////////////////////////////////////////////////////////

`timescale 1 ns / 1 ns
`define IXCclkgenTs 1 ns / 1 ns
module _ixc_clkgen;
  wire clkA_0;
  wire clkB_1;

   initial $ixc_ctrl("map_delays");

  // Generate logic for clock sources
  IXCclkgen  #(1) ixcg_0 (clkA_0, );
  IXCcake2x  #(300, 500, 25) ixcg_1 (clkB_1, clkA_0);

   // ...
endmodule
```

For CAKE 1X technology, the clock delay value is applied to the `IXCcake1x` and `IXCcake1xN` primitive instances similar to the example shown above.

Specifying Clock Assignment

Clock assignment specification is supported using the following User Data syntax:

clockAssign -add { <*clock_signal*-1> [~] <*clock_signal*-2> [N] }

Following is the generated clock module for a sample set of User Data with different clock assignments:

Command:

```
ixclkgen -input clock-ud.qel -output clk.sv
```

Input File:

```
# File: clock-ud.qel

clockOption -add { technology CAKE 1 }
```


© 2021 All Rights Reserved.

```
clockFrequency -add { dut.clkA 0.5 Ghz }

clockAssign -add { dut.clkB   dut.clkA   }
clockAssign -add { dut.clkC ~ dut.clkA   }
clockAssign -add { dut.clkD   dut.clkA 2 }
clockAssign -add { dut.clkE   dut.clkA 4 }
clockAssign -add { dut.clkF ~ dut.clkA 8 }
```

Output File:

```
// This is an auto-generated file using the ixclkgen utility
// Command-line specified:
//   ixclkgen -i ud.qel  -o clk.sv
//

// NOTE: Please refer to VXE user guide for clock modeling in IXCOM. Try:
//   ixclkgen -help
//   for usage and quick guide to compiling generated clock module.
////////////////////////////////////////////////////////////////////////////////
`timescale 1 ns / 1 ns
`define IXCclkgenTs 1 ns / 1 ns
module _ixc_clkgen;
  wire clkA_0;
`ifdef IXCOM_UXE
  initial $ixc_ctrl("map_delays");
`endif
  // Generate logic for clock sources
  IXCclkgen  #(1) ixcg_0 (clkA_0  );
  // Bind clock sources to generated clock signal
  ixc_cakebind ixcb_0 (dut.clkA, clkA_0);
  // Generate logic for clock assigns
  ixc_1xbuf1p ixca_0 (dut.clkB, clkA_0);
  wire clkC_2;
  ixc_inverter ixci_1 (clkC_2, clkA_0);
  ixc_1xbuf1n ixca_1 (dut.clkC, clkC_2);
  wire clkD_3;
  ixc_posedge_clk_divider #(1, 2, 1) ixcd_2 (clkD_3, clkA_0);
  ixc_1xbuf1p ixca_2 (dut.clkD, clkD_3);
  wire clkE_4;
  ixc_posedge_clk_divider #(1, 4, 2) ixcd_3 (clkE_4, clkA_0);
  ixc_1xbuf1p ixca_3 (dut.clkE, clkE_4);
  wire clkF_5;
  ixc_negedge_clk_divider #(1, 8, 3) ixcd_4 (clkF_5, clkA_0);
  ixc_1xbuf1n ixca_4 (dut.clkF, clkF_5);
`ifdef IXCOM_UXE
  initial begin
    $ua_cmd("cakeClk", "dut.clkA", "ixcg_0", "500000kHz", "B1", "0");
  end
`endif
endmodule // _ixc_clkgen
```

Note: In the above example, the CAKE 1X clock is generated to show the variants of generated logic. The clock signal referenced in the `clockAssign` User Data can be any of the design signals; it does not have to be driven by a clock source.

© 2021 All Rights Reserved.

Using Uncontrolled Hardware Clocks

The default behavior for CAKE clocks is controlled if you are using the STB mode or with a static or no target in ICE mode. On the other hand, the clocks are uncontrolled (or free running) in ICE mode if you use a dynamic target (`configPM -la` command).

In SA mode, all clocks are generated as controlled clocks. To generate ICE-mode-like uncontrolled clocks, you must specify a list of uncontrolled clocks by using the `-uncontrolled` option of the `ixclkgen` command as shown in the example below:

Command:

```
ixclkgen -input clock-ud.qel -output clk.sv \
         -uncontrolled dut.clkB -uncontrolled dut.clkC,dut.clkD
```

Input File:

```
# File: clock-ud.qel

clockOption -add { technology CAKE 2 }

clockFrequency -add { dut.clkA 677 MHz }
clockFrequency -add { dut.clkB 433 MHz }
clockFrequency -add { dut.clkC 344 MHz }
clockFrequency -add { dut.clkD 500 MHz }
clockFrequency -add { dut.clkE 545 MHz }
```

Output File:

```
// This is an auto-generated file using the ixclkgen utility.
//   Input  File: clock-ud.qel
//   Output File: clk.sv
// NOTE: Please refer to VXE user guide for clock modeling in IXCOM. Try:
//   ixclkgen -help
//   for usage and quick guide to compiling generated clock module.
////////////////////////////////////////////////////////////////////////////
`timescale 1 ps / 1 ps
`define IXCclkgenTs 1 ps / 1 ps
module _ixc_clkgen;
  wire clkA_0;
  wire clkB_1;
  wire clkC_2;
  wire clkD_3;
  wire clkE_4;

  initial $ixc_ctrl("map_delays");

  // Generate logic for clock sources
  IXCclkgen  #(738) ixcg_0 (clkA_0, );
  IXCcake2x  #(433, 500, 0) ixcg_1 (clkB_1, clkD_3);
  IXCcake2x  #(344, 500, 0) ixcg_2 (clkC_2, clkD_3);
  ixc_hwclksrc ixcg_3 (clkD_3, 1'b1);
  IXCcake2x  #(545, 677, 0) ixcg_4 (clkE_4, clkA_0);

  // Bind clock sources to generated clock signal
  ixc_cakebind ixcb_0 (dut.clkA, clkA_0);
  ixc_cakebind ixcb_1 (dut.clkB, clkB_1);
  ixc_cakebind ixcb_2 (dut.clkC, clkC_2);
  ixc_cakebind ixcb_3 (dut.clkD, clkD_3);
  ixc_cakebind ixcb_4 (dut.clkE, clkE_4);
```


© 2021 All Rights Reserved.

```
//...
endmodule
```

Note: The default clock generation behavior for the clock sources can be specified separately using the `-default` option. Use `ixclkgen -help` for more details.

Specifying Hierarchy

In ICE mode, a reference clock signal (or source) in the User Data does not include the top-level module information. Also, in SA mode, you might want to instantiate generated clock module in testbench hierarchy relative to the design referred by the User Data.

The following tool invocation shows the use of the `-hierarchy` option to insert the hierarchy that you have specified for the clock signals referred in the User Data file:

Command:

```
ixclkgen -input clock-ud.qel -output clk.sv -hierarchy "Tb.dut."
```

Input File:

```
# File: clock-ud.qel

clockOption -add { technology CAKE 2 }

clockOption -add { technology CAKE 2 }
clockFrequency -add { clkA 100 MHz }
clockFrequency -add { sub.clkB 125 Mhz }
```

Output File:

```
// This is an auto-generated file using the ixclkgen utility.
//   Input  File: clock-ud.qel
//   Output File: clk.sv
// NOTE: Please refer to VXE user guide for clock modeling in IXCOM. Try:
//   ixclkgen -help
//   for usage and quick guide to compiling generated clock module.
//////////////////////////////////////////////////////////////////////////
`timescale 1 ns / 1 ns
`define IXCclkgenTs 1 ns / 1 ns
module _ixc_clkgen;
  wire clkA_0;
  wire clkB_1;

  initial $ixc_ctrl("map_delays");

  // Generate logic for clock sources
  IXCcake2x  #(100, 125, 0) ixcg_0 (clkA_0, clkB_1);
  IXCclkgen  #(4) ixcg_1 (clkB_1, );

  // Bind clock sources to generated clock signal
  ixc_cakebind ixcb_0 (Tb.dut.clkA, clkA_0);
  ixc_cakebind ixcb_1 (Tb.dut.sub.clkB, clkB_1);
  // ...
endmodule
```


© 2021 All Rights Reserved.

Specifying the Clock Configuration at Run Time with ixclkgen Utility

By default, the *ixclkgen* utility maps the design clocks to the minimum required clock-generation primitives. The generated clocks cannot be configured at run time. To enable run-time configuration of the design clocks using the `clockConfig` command, you must specify the `-rtc` option of the `ixclkgen` command.

To support run-time configurable user clocks, the *ixclkgen* utility generates an internal clock, `_ixci_clk` for controlled clocks and `_ixci_hclk` for uncontrolled clocks, with the fastest user-clock frequency and all the design clocks are generated relative to the generated internal clock.

Command:

```
ixclkgen -input clock-ud.qel -output clk.sv -rtc
```

Input File:

```
# File: clock-ud.qel

clockOption -add { technology CAKE 2 }

clockFrequency -add { dut.clkA 500 MHz }
clockFrequency -add { dut.clkB 300 MHz }
```

Output File:

```
// This is an auto-generated file using the ixclkgen utility.
//   Input  File: clock-ud.qel
//   Output File: clk.sv
// NOTE: Please refer to VXE user guide for clock modeling in IXCOM. Try:
//   ixclkgen -help
//   for usage and quick guide to compiling generated clock module.
////////////////////////////////////////////////////////////////////////////
`timescale 1 ns / 1 ns
`define IXCclkgenTs 1 ns / 1 ns
module _ixc_clkgen;
  wire clkA_0;
  wire clkB_1;
  wire _ixci_clk;

  initial $ixc_ctrl("map_delays");

  // Generate logic for clock sources
  IXCcake2x #(500, 500, 0) ixcg_0 (clkA_0, _ixci_clk);
  IXCcake2x #(300, 500, 0) ixcg_1 (clkB_1, _ixci_clk);
  IXCclkgen #(1) ixcg_2 (_ixci_clk, );

  // Bind clock sources to generated clock signal
  ixc_clkbind ixcb_0 (dut.clkA, clkA_0);
  ixc_clkbind ixcb_1 (dut.clkB, clkB_1);


`ifdef IXCOM_UXE
  initial begin
    $ua_cmd("cakeClk", "dut.clkA", "ixcg_0", "500MHz", "C ", "0");
    $ua_cmd("cakeClk", "dut.clkB", "ixcg_1", "300MHz", "C ", "0");
  $ua_cmd("cakeClk", "_ixci_clk", "ixcg_2", "500MHz", "B2", "0");
  end
```


© 2021 All Rights Reserved.

```
    `endif
    endmodule
```

The following syntax is supported for the clockConfig command when the *ixclkgen* utility is used:

```
clockConfig
          [-sample <value>]
          [-dly <clockName> <value>]
          [-clk <clockName> <value>]
          [-list]
          [-restore] [<filename>]
          [-save] [<filename>]
          [-actual]
```

For details about the run-time clock configuration using the `clockConfig` command, refer to *VXE Command Reference Manual*.

Specifying the Clock Configuration

To support clock configuration at run-time using the clockConfig user command, the *ixclkgen* utility passes the information about the clocks by using the `$ua_cmd` statements in the generated code with the following syntax:

```
$ua_cmd("cakeClk", "<clock_path>", "<instance_name>", "<frequency>", "type",
        "<delay>");
```

Here,

`cakeClk`	Specifies the fixed tag for the `ixc_ctrl` statement.
<clock_path>	Specifies the hierarchical path to clock source from the User Data
<instance_name>	Specifies the IXC CAKE clock primitive instance name
<frequency>	Specifies the clock frequency value from User Data
<type>	Specifies one of the following *<type>* values to indicate the cake clock properties or attributes
`B1`	Used for controlled base clock 1X mode
`B2`	Used for controlled base clock 2X mode
`C`	Used for controlled CAKE clock
`c`	Used for uncontrolled CAKE clock
`b`	Used for uncontrolled `ixc_hwclksrc`
<delay>	Specifies the clock delay value (if any) from User Data

© 2021 All Rights Reserved.

For the controlled and uncontrolled clocks referenced in Using Uncontrolled Hardware Clocks, additional HDL is generated in the Verilog module as shown below:

Command:

```
ixclkgen -input clock-ud.qel -output clk.sv
```

Output File:

```
// This is an auto-generated file using the ixclkgen utility.
// Input  File: clock-ud.qel
// Output File: clk.sv
// NOTE: Please refer to VXE user guide for clock modeling in IXCOM. Try:
//   ixclkgen -help
//   for usage and quick guide to compiling generated clock module.
//////////////////////////////////////////////////////////////////////////////
`timescale 1 ps / 1 ps
`define IXCclkgenTs 1 ps / 1 ps
module _ixc_clkgen;
  // See Using Uncontrolled Hardware Clocks for the rest of the generated module
     content
  `ifdef IXCOM_UXE
    initial begin
      $ua_cmd("cakeClk", "dut.clkA", "ixcg_0", "677MHz", "B2", "0");
      $ua_cmd("cakeClk", "dut.clkB", "ixcg_1", "433MHz", "c ", "0");
      $ua_cmd("cakeClk", "dut.clkC", "ixcg_2", "344MHz", "c ", "0");
      $ua_cmd("cakeClk", "dut.clkD", "ixcg_3", "500MHz", "b ", "0");
      $ua_cmd("cakeClk", "dut.clkE", "ixcg_4", "545MHz", "C ", "0");
    end
  `endif
endmodule
```

Oversampling in SA Mode

While compiling designs in SA mode, if you omit specifying the `+1xua` option of the `ixcom` command, then the designs are run with 2X oversampling ratio. However, you can increase the oversampling ratio at run time. To increase the ratio to a specified limit, execute the `xc fclkPerEval <N>` command.

The following table specifies the correspondence between the value of <*N*> and the oversampling ratio.

Value of <*N*>	Command	Corresponding Sampling Ratio = 2*(N+1)X
N = 1	`xc fclkPerEval 1`	4X
N = 2	`xc fclkPerEval 2`	6X
N = 3	`xc fclkPerEval 3`	8X

© 2021 All Rights Reserved.

For details on the `xc fclkPerEval <N>` command, refer to the *SA Run-Time and Debug Commands* chapter of the *VXE Command Reference Manual*.

© 2021 All Rights Reserved.

23

Design Profiling for Palladium Z1 Acceleration

Profiling a design in the context of hardware acceleration using Palladium Z1 involves measuring the time spent in running the DUT on the hardware and remaining parts of the design running on the software simulator. A run-time profile file that contains the simulation-run-time performance information is generated. This information enables you to identify bottlenecks in the non-acceleratable testbench environment, which can then be optimized to enhance performance with the Palladium Z1 system.

The Xcelium profiler is used to estimate the potential speed up before an emulator is available. To analyze the performance speed up possible with the use of the Palladium Z1 emulator, use the XC Profiler.

IXCOM works in an integrated manner with the Xcelium advanced profiler, xprof.

This chapter covers the following topics:

- Design Profiling on Software Using Xcelium Profiler
- Profiling Design Run-Time Performance on Hardware with XC Profiler
- Integration with Advanced Profiler - xprof

Design Profiling on Software Using Xcelium Profiler

You can use the Xcelium profiler on Palladium Z1 to:

- Estimate the expected speedup/performance on a given design when it will be run in COSIM (SA) mode on Palladium Z1 when compared to the current pure simulation run.
- Identify the bottlenecks in the non-acceleratable testbench (non-DUT HDL) environment or embedded software modules/functions (TBCalls) that can be optimized to achieve better acceleration performance with the Palladium Z1 system.

© 2021 All Rights Reserved.

When you enable the Xcelium profiler, you can obtain cumulative statistics of the CPU time spent within portions of the design. These statistics are divided into the following categories:

- `DUT` - Design units or scopes (including complete hierarchies below a scope) of HDL that you have marked as `DUT`.
- `OTHER` - Other units or the scope outside of DUT scopes or those that you have explicitly marked as `OTHER`.
- User C applications in the form of shared object libraries communicating with the simulator using PLI/VPI/VHPI or DPI methods.

The DUT summary section gives a cumulative CPU time by adding the CPU time of each module according to their hierarchy (DUT or OTHER). In addition to this cumulative information, the Xcelium profiler presents the modules that are top CPU-time consumers.

The Xcelium profiler works by interrupting the simulation at regular intervals and noting what was executing at that time. It tracks the number of “hits” on different activities, and this number indicates the approximate amount of CPU time spent in these activities.

The Xcelium profiler is easy to run and has minimal impact on simulation performance and memory usage. Cadence recommends that for an apples-to-apples comparison with final hardware-based performance (that offers full waveform debug capabilities all the time), a full debug-enabled (`+access+rwc`) snapshot should be created.

Commands for Design Profiling Using the Xcelium Profiler

Invoke the Xcelium-based design profiler by using one of the following commands:

- Using the -profile Option of the xrun Command
  - Following are the elaboration-time profiling options of the `xrun` command related to the `xmelab` command:
    - `-dut_prof_elab`
    - `-cycle_prof`
  - Following are the run-time profiling options of the `xrun` command related to the `xmsim` command:
    - `-profile`
    - `-dut_prof <dut_spec_file>`
- Using the +profile Option of the ixcom Command

© 2021 All Rights Reserved.

- Using the Tcl profile Command

Using the -profile Option of the xrun Command

You can use the `-profile` option with the `xrun` command to generate profiling statistics for a design, as shown below:

```
% xrun -profile [other_xrun_options] <source_files>
```

When the simulator exits, the profiler creates an output file. By default, the file is created in the current working directory and is called `ncprof.out`.

If you start a simulation with the `-profile` option, profiling can be stopped only by exiting or resetting the simulation. Resetting the simulation with a `reset` command switches off the profiling automatically.

For more information, see *Running the Profiler* section in the *Maximizing Simulation Performance* document in the Xcelium documentation set.

Using the -dut_prof Option for Reporting DUT Profiling Information

Accurately identifying the time spent in the DUT is the most important step in determining if a design is suitable for hardware acceleration. Use the `-dut_prof` option to make the Xcelium Profiler aware of the DUT in the design environment, and generate a run-time profile file, which, by default, is named `ncprof.out`. This file includes a table called `Design Unit Time Summary` that reports the percentage of time spent simulating a specified design under test (DUT) and the amount of time spent in the rest of the design.

```
xrun -dut_prof <dut_prof_spec> -profile
```

Format of the DUT Specification Information File

At run time, you need to provide a DUT specification information file, `<dut_prof_spec>` to the Xcelium profiler. This file must specify the name of at least one design unit considered to be top level of the DUT RTL/Gate hierarchy, and is assumed to be acceleratable on the Palladium Z1 system.

The format of the DUT specification in this file is:

```
# comments can be also be added
DUT: <lib>.<cell>:<view>
OTHER: <lib>.<cell>:<view>
```

A sample `<dut_spec_file>` is provided below:

```
-----------------------------------------------------
# This is a sample dut specification file
```


© 2021 All Rights Reserved.

```
DUT: LIB1.E1:A1
OTHER: LIB2.E2:A2
DUT: LIB3.E3:A3
-----------------------------------------------------
```

In the <*dut_spec_file*>, the `DUT` and `OTHER` keywords have the following meaning:

- `DUT`: Implies that all the specified instances of `L.C:V` (<*lib*>.<*cell*>:<*view*>) should be considered as DUT `top`. All the instantiation below DUT `top` (defined recursively) including the top-level module of the DUT, is considered as a part of DUT, and the rest is considered as a part of the testbench.
- `OTHER`: None of the instances of the specified `L.C:V` should be considered as a part of DUT. If an instance of `L.C:V`, which is specified in `OTHER`, is under the DUT hierarchy of DUT, the whole instance hierarchy below it (`OTHER: L.C:V`) is not considered as DUT. All embedded modules must be specified in the `OTHER` category.

Specify the required module and all its sub-hierarchies to mark these for profiling as DUT design units. You can specify multiple DUT design units, and their sub-hierarchies. For example:

```
# multiple top level DUT modules
dut: sontop.sontop:v
dut: worklib.bidi:v
dut: chip.data_grab:v
OTHER: worklib.mc28m256x:v
```

Note: It is mandatory to specify the <*library*> and <*view*> names in Xcelium format. To find out the names of the library and view of the module, use the `xmls` utility. The `xmls` utility lists the compiled objects stored in the library system and displays various attributes and information about those objects. For example:

```
% xmls -64 bidi
xmls: 06.20-s004: (c) Copyright 1995-2008 Cadence Design Systems, Inc.
        module worklib.bidi:v (VST)
        module worklib.bidi:v (SIG) <0x661d9fc9>
```

In the above example, the `bidi` design unit resides in `worklib` library with a single Verilog view `v`.

With the `xmls` command, the `-64` option is required because the default setting for the `xmls` utility is 32 bit, while the IXCOM compiler uses 64 bit by default.

If a module exists in multiple libraries or has multiple views, choose the required library and view.

All HDL scopes outside of those marked as DUT are implicitly considered in the `OTHER` category to be interpreted as the testbench or verification-related HDL, which cannot be run on an HDL accelerator like Palladium Z1.

© 2021 All Rights Reserved.

In addition to the implicit `OTHER` scopes, any particular design unit can be marked explicitly as `OTHER`. This is useful when a DUT hierarchy marked with the top-level DUT module contains a non-synthesizable design unit (including the sub-hierarchy below it), and, therefore, those scopes should not be counted as acceleratable design unit in the profiling information. In such cases, the units at the top of the non-synthesizable scopes should be added to `dut_spec_file` using the `OTHER` marking.

In the above case, the Xcelium profiler includes CPU time for `mc28m256x` also under the `OTHER` category.

Tip

Profiling information should be generated out of several simulation runs with different test conditions, each of which should run for a duration of at least 5-10 minutes. Very short simulation runs are likely to give inaccurate profile numbers because the design cannot be stimulated enough.

For more information, see *Using the Profiler to Identify and Eliminate Simulation Bottlenecks* in the *Maximizing Simulation Performance* document available in the Xcelium documentation set.

Using the +profile Option of the ixcom Command

You can use the `+profile` option of the `ixcom` command to generate profiling statistics for a design using the basic or advanced Xcelium profilers, as shown below:

```
% ixcom {+profile=[ncprof|xprof]} [other_ixcom_options] <source_files>
```

To invoke the basic Xcelium profiler, specify the `ncprof` argument to the `+profile` option. To invoke the advanced Xcelium profiler, specify the `xprof` argument to the `+profile` option. If you do not specify any argument with the `+profile` option, the default argument is `ncprof`.

For more information, see *Using the Advanced Profiler* and *Using the Basic Profiler* sections in the *Maximizing Simulation Performance* document in the Xcelium documentation set.

Using the `+profile` option implies the following:

- At elaboration time, IXCOM invokes the *xmelab* utility is invoked with the `-dut_prof_elab` option. The IXCOM compiler automatically generates the DUT profile specification file in the `xc_work` directory, and specifies this file as an argument to the `-dut_prof` option when invoking *xrun*. If you have specified the `+profile` option, IXCOM generates this file even if you have not specified the `-ua` option with the `ixcom` command.

© 2021 All Rights Reserved.

- The IXCOM compiler enables Xcelium profiling by adding `-profile -dut_prof <generated_dut_spec_file>` with the `XMSIMOPTS` option in the `xc_work/hdl.var` file. The IXCOM compiler also enables hardware profiling by adding the `+xcprof` argument with the `XMSIMOPTS` option in the `xc_work/hdl.var` file. The hardware profiling is enabled only when the design is run on hardware, otherwise, it is not invoked.

Using the Tcl profile Command

The `profile` command provides more control over the profiler behavior than the `-profile` option.

With the `profile` command, you can:

- Start or stop profiling at any time (`-on` and `-off`)
- Dump the profiled data to a file at any time (`-dump`)
- Clear the currently collected profiled data (`-clear`)

If profiling is enabled with the `profile` command, it remains on even after you issue a `reset` command. The `reset` command clears the profile data, but does not disable profiling.

Use one of the following options on Xcelium Tcl command line:

```
xmsim> help profile
profile.......................Profile Command
   [-clear]...................Clear profile data.
   [-dump] [-overwrite].......Dump profile data to file. Use the overwrite
                              option to replace an existing file.
   [-off].....................Disable profiling of user designs.
   [-on]......................Enable Profiling of user designs.
```

For example:

```
xmsim> profile -dump myprof.out -clear
```

Note: The `-dut_prof` option is not supported with the Tcl `profile` command.

For more information about the `profile` command, see the *profile* section in the *Using the Tcl Command-line Interface* chapter of the *Xcelium Simulator Tcl Command Reference* document in the Xcelium documentation set.

Analyzing and Interpreting the Xcelium Profiler Output in ncprof.out

The profiler output file, `ncprof.out` contains the following sections:

- Design Unit Time Summary

© 2021 All Rights Reserved.

- Most Active Modules (Other Units)
- Stream Counts
- Most Active Modules (Behavioral)
- IXCOM Run-Time Profiler Report (hardware-run only)
- Stream Type Summary Counts

Design Unit Time Summary

In an `ncprof.out` file generated using the `-dut_prof` option, the Design Unit Time Summary section summarizes the % of hits and number of hits for the following categories:

- DUT
- HDL corresponding to non-DUT hierarchy.
- HDL not attributed to either 1 or 2 above, under the Aggregated Simulation Overhead category
- HDL C interface
- User Code
- IXCOM (Emulator + Overhead), if design profiled in IXCOM hardware mode
- Gate Evaluation Methods
- Support for VPI callbacks (or UI)
- Engine support
- Outside engine
- File I/O

```
------------------------------------------------------------
Design Unit Time Summary (473684 hits total)
------------------------------------------------------------
%hits        #hits        Set
 66.3        314075       Design Unit
  2.6         12545       Other  Units
  2.5         11731       Aggregated Simulation Overhead
 12.9         60985       Specman
  2.1          9739       HDL C interface
  2.4         11321       User Code
 10.0         47835       Support for VPI callbacks (or UI)
  0.1           235       Engine support
  0.9          4608       Outside engine
  0.2           608       File I/O
```


© 2021 All Rights Reserved.

Potential Acceleration Estimation in the SA Run Mode

This section provides a range of achievable acceleration in the SA run mode estimated on the basis of the various factors that affect the improvement.

This section includes the following sub-sections:

- Best-Case Performance Estimate
- Average-Case Performance Estimate

Following is a brief description of these sub-sections.

Best-Case Performance Estimate

This section provides the performance estimate for the best case, and is based on following equation:

Estimated speedup = 100/(100-(%DUT CPU time))

This is the best-case performance estimate of the SA run mode, and is based on the following assumptions:

- The time spent in hardware-software synchronization and in executing the DUT on Palladium Z1 is assumed to be negligible as compared to the time spent in testbench.
- The time taken by the testbench (software) portion of the design to execute on the simulator remains identical in the co-simulation run and the present pure-simulation run.

In this section, the following data is generated:

```
--- Time spent in "Design Unit (DUT)" = 66.3%
--- Time spent in "Aggregated Simulation Overhead (ASO)" = 2.5%
```

Dividing the `ASO` in proportion between the `DUT` and `OTHER` categories for a more realistic estimate:

```
--- Estimated Time spent in DUT = 66.3% + 66.3% of 2.5 (ASO %) = 68.0%
--- Estimated Best-Case Speedup = (100/(100-(DUT time%))) = 3.12X
```

The <*`dut_spec_file`*> specifies the DUT hierarchy and the profiler adds the time for that design hierarchy into the time spent in DUT.

The `-dut_prof_elab` option must be included with the `xrun` command for the above-mentioned information to be generated. Otherwise, the following message is displayed at the top of the section in `ncprof.out` file:

© 2021 All Rights Reserved.

```
### Please elaborate the design snapshot using '-dut_prof_elab' XMELAB/xrun
elaboration option for improved accuracy in Time spent in DUT ####
```

Average-Case Performance Estimate

While calculating the best-case performance estimate, the time spent in hardware-software synchronization and executing the DUT on Palladium Z1 is assumed to be negligible as compared to the time spent in the testbench. However, this assumption is not completely valid. Therefore, to calculate a reasonable average-case performance estimate of the speed increase, it is necessary to include the time spent in hardware-software synchronization and executing the DUT on Palladium Z1. Generally, each synchronization incurs a penalty in the range of 10 us to 15 us. So, a typical value of 13 us penalty per hardware-software synchronization is assumed. The average-case performance estimate can be determined by estimating the number of hardware-software synchronizations required in the co-simulation run.

To enable this calculation, you need to use the `-cycle_prof` and `-dut_prof_elab` options of the `xrun` command. When you use these options, the following data is dumped in the `ncprof.out` file.

```
--- SW simulator executes @ 29487.78 cycles/sec.
--- SW simulation runtime = 463.65 sec.
--- Time spent in SW SIM of non-DUT portion = (100 - %DUT)% of SW simulation
    runtime = 148.37 sec.
--- Estimated number of HW-SW synchronizations required in COSIM run = 4391604
--- Estimated COSIM runtime = DUT runtime on HW (incl. HW-SW Sync Overhead) +
    non-DUT runtime on SW = 57.09  + 148.37 = 205.46 sec.
--- Estimated Average Case speedup = (SW SIM runtime)/(COSIM runtime) =
    463.65/205.46 = 2.26 X.
```

This sub-section is displayed only when you specify the `-cycle_prof` option while elaborating the design hierarchy. Otherwise, a message like the following appears in the `ncprof.out` file.

```
### Please elaborate the design snapshot using '-cycle_prof' XMELAB/xrun
    elaboration option for Average Case Speedup Estimation ####
```

You should use the following `xrun` command options for Xcelium design profiling if you need the average-case performance estimate section in the `ncprof.out` file:

```
% xrun -dut_prof_elab \
       -cycle_prof -profile \
       -dut_prof <dut_specification_file> \
       [other_xrun_options] <source_files>
```

Sample Report File with the Average-Case Performance Estimate Section

Following is an extract of the header part of a sample report file, when both the `-dut_prof_elab` and `-cycle_prof` elaboration options have been specified.

© 2021 All Rights Reserved.

```
-------------------------------------------------------------
Design Unit Time Summary (473684hits total)
-------------------------------------------------------------
%hits      #hits              Set
66.3       314075         Design Unit
 2.6        12545         Other  Units
 2.5        11731         Aggregated Simulation Overhead
12.9        60985         Specman
 2.1         9739         HDL C interface
 2.4        11321         User Code
10.0        47835         Support for VPI callbacks (or UI)
 0.1          235         Engine support
 0.9         4608         Outside engine
 0.2          608         File I/O
-------------------------------------------------------------
Potential Speed-up Estimation in IXCOM 'Run Mode'
-------------------------------------------------------------
Best Case Performance Estimate
------------------------------
  --- Time spent in "Design Unit (DUT)" =  66.3%
  --- Time spent in "Aggregated Simulation Overhead (ASO)" =  2.5%
  Dividing the ASO in proportion between DUT and OTHER to get more realistic
  estimate
  --- Estimated Time spent in DUT = 66.3% + 66.3% of 2.5 (ASO%) = 68.0%
  --- Estimated Best-Case Speedup = (100/(100-(DUT time%))) = 3.12X
Average Case Performance Estimate
---------------------------------
  --- SW simulator executes @ 29487.78 cycles/sec.
  --- SW simulation runtime = 463.65 sec.
  --- Time spent in SW SIM of non-DUT portion = (100 - %DUT)% of SW simulation
      runtime = 148.37 sec.
  --- Estimated number of HW-SW synchronizations required in COSIM run = 4391604
  --- Estimated COSIM runtime = DUT runtime on HW (incl. HW-SW Sync Overhead) +
      non-DUT runtime on SW = 57.09  + 148.37 = 205.46 sec.
  --- Estimated Average Case speedup = (SW SIM runtime)/(COSIM runtime) =
      463.65/205.46 = 2.26 X.
```

Most Active Modules (Other Units)

The `ncprof.out` file also provides the top CPU-time consumers in `OTHER` scopes in the `Most Active Modules` section. This section carries detailed information about `OTHER` time consumers. This is useful for identifying and prioritizing the HDL-verification modules that can be targeted for improving performance of acceleration by mapping them to hardware or less CPU-time-consuming methodologies.

```
Most Active Modules (Other Units)
-------------------------------------------------------------
%hits #hits  #inst  name
0.8  5131 [    1] worklib.SonTopTB:v (file: ./son/verification/chip/SonTopTB.v
line: 21)
....
```


© 2021 All Rights Reserved.

Stream Counts

This section indicates the proportion of simulation time spent in different parts of the design (processes, subprograms), in the kernel, in housekeeping, in user C code, and in interacting with the operating system.

The following example shows a part of a Stream Counts section from a profile.

```
------------------------------------------------------------
Stream Counts (90056 hits total)
------------------------------------------------------------
%hits #hits  #inst  name
36.2 32568  [    1] Always stmt (file: ./hdl/hspll.v, line: 54 in worklib.hspll
[module])
 32.9 29654 [     ] specman engine
  8.2  7372 [     ] Method SSS_KM_CL2TA (method)
  7.8  7067 [     ] Method SSS_KM_FINDRFT (method)
  6.1  5462 [     ] /<specman_install>/specman/6.20.s3/components/sn/linux/
libxmvlog_sn_adapter.so
  1.3  1165 [     ] Logic primitive 'timing output', Module 'BUFX20', Port 'Y', 1
Timing Inputs (process)
```

The Specman verification engine is a common case of an external application, and in the above example, it appears in the Stream Counts section as two separate entries - one for the Specman engine and the other for Specman interface (`libxmvlog_sn_adapter`) to the simulator. Here, the Specman engine is consuming *32.9%* time and the adapter and interface to simulator *6.1%*. This separation of engine time and interface time in Stream Counts is only done for Specman case. For a general case of C application, all of the time will be reported in Stream Counts, and will be associated with the `.so` file.

Most Active Modules (Behavioral)

This section summarizes the contribution from each of the design units in descending order of the simulation time. This can be useful when a design unit is taking a lot of time due to the combined time in streams below the threshold for being listed in the Stream Counts section.

Following is an example of the *Most Active Modules* section.

```
------------------------------------------------------------
Most Active Modules (behavioral)
------------------------------------------------------------
%hits #hits #inst name
7.3   57    [1] xc_ncwork.xc_top:module (file:./xc_work/v/xc_top.v line: 7)
0.9   7     [1] xc_ncwork.nw_driver_xcm23:module (file:./xc_work/v/23n.v95 line:
4)
0.9   7     [1] xc_ncwork.nw_driver_xcm25:module (file:./xc_work/v/25n.v95 line:
4)
0.6   5     [1] xc_ncwork.nw_monitor_xcm20:module (file:./xc_work/v/20n.v95 line:
4)
0.6   5     [1] xc_ncwork.nw_monitor_xcm26:module (file:./xc_work/v/26n.v95 line:
4)
0.5   4     [1] xc_ncwork.clock:module (file:./xc_work/v/2n.v95 line: 4)
```


© 2021 All Rights Reserved.

```
0.5   4     [1] xc_ncwork.nw_driver_xcm19:module (file:./xc_work/v/19n.v95 line:
4)
0.5   4     [1] xc_ncwork.nw_monitor_xcm24:module (file:./xc_work/v/24n.v95 line:
4)
```

IXCOM Run-Time Profiler Report

This section presents the IXCOM run-time profiling information for the entire design. this section is generated only when the design is run on the hardware. For more information, see Profiling Design Run-Time Performance on Hardware with XC Profiler on page 1147.

```
------------------------------------------------------------
IXCOM Runtime Profiler Report (760 hits,  97.4% of total)
------------------------------------------------------------
%hits   #hits  %xcRun        name            comments
  2.8    22     2.9       XCRT-OH      IXCOM runtime Control (Overhead).
 24.2   189    24.9       SW-SIM       TB and TBCallexecution in sw simulator
  2.2    17     2.2       DECODE       HW O/P change decode (Overhead).
 68.2   532    70.0       HW-EMU       Time spent in HW evaluation & Sync Latency
```

Note: This section is reported in `ncprof.out` only if the design is also run on the hardware. For a pure software simulation, the `ncprof.out` file does not include the `IXCOM Run-Time Profiler Report` section.

Stream Type Summary Counts

This final section of the `ncprof.out` file summarizes the overall contribution by different activity units toward simulation time. This section includes streams with counts too low to appear separately in the first section and gives a general idea of where simulation time is being spent.

The summary makes it easier to identify widespread inefficiencies in the simulation.

Following is an example of *Stream Type Summary Counts*.

```
------------------------------------------------------------
Stream    Type    Summary Counts (780 hits total)
------------------------------------------------------------
%hits     #hits       #inst            name
 73.2       571       [    ]           IXCOM (Emulator + Overhead)
  6.7        52       [6029]           Always statements
  4.2        33       [ 179]           Verilog tasks
  4.1        32       [    ]           Simulation Engine Time
  2.7        21       [    ]           User defined system tasks/functions
  1.9        15       [2199]           Initial statements
  1.7        13       [  66]           Verilog functions
  1.3        10       [    ]           Engine support
  1.3        10       [    ]           HDL C interface
  1.2         9       [    ]           VCD/SHM variable dumping
  0.8         6       [6603]           Non-blocking assignments
  0.4         3       [4926]           Continuous assignments
```


© 2021 All Rights Reserved.

```
0.4         3       [    ]          CAG stream
0.3         2       [ 831]          Anonymous continuous assignments
```

For more information, see *Using the Profiler to Identify and Eliminate Simulation Bottlenecks* in the *Maximizing Simulation Performance* document available in the Xcelium documentation set.

Assertions Reporting in ncprof.out

For a design compiled and run in pure software simulation (using Xcelium only), assertions are profiled separately. The profiling information related to assertions is reported as follows:

- An `Assertions` category is reported in the `Design Unit Time Summary` section.
- A separate `Assertion Summary Counts` section is reported in the `ncprof.out` file.

The IXCOM compiler, however, converts assertions to HDL. Therefore, for a design compiled using the IXCOM compiler, the assertions are not profiled separately and reported in either of the above two sections in `ncprof.out`.

Improving Profiling Accuracy and Caveats

Following sections provide tips and caveats for improving the profiler accuracy related to:

- Using the -dut_prof_elab Option in Pure Software Simulation
- Using -cycle_prof to Improve the Potential Acceleration Estimation
- Updating Embedded Modules Information in <dut_spec_file>
- Using Xcelium Run-Time Profile Command to Control Profiling

Using the -dut_prof_elab Option in Pure Software Simulation

To ensure accuracy in the statistics reported in the `Design Unit Time Summary` section of `ncprof.out` for a design compiled and run using Xcelium, use the `-dut_prof_elab` option during design elaboration.

Using -cycle_prof to Improve the Potential Acceleration Estimation

Use of the `-cycle_prof` option is costly, and can incur 5-10% of simulation-run-time penalty. Use this option only when profiling the design for a potential acceleration estimation.

© 2021 All Rights Reserved.

Updating Embedded Modules Information in <dut_spec_file>

In a typical profiling exercise, it is customary to declare just the top design units of the DUT hierarchy in <*dut_spec_file*>. This makes an assumption that the entire hierarchies are synthesizable and is a good first order assumption to get an estimate of acceleratability of the environment.

However, for a more precise profile, you must check if there are any significant non-synthesizable blocks under the DUT hierarchy that need to run under Xcelium and, therefore, must be explicitly marked as `OTHER` in the <*dut_spec_file*>. Examples of such design blocks include: embedded C model of IPs, analog-mixed signal blocks, and complex behavioral CAMs.

To update the `OTHER` list in the <*dut_spec_file*>, first generate a list of embedded software modules from the compiler's RTL mapping phase in `ixcom.log`. For this step, the design should be run through IXCOM compiler with the `-ua` option.

This produces the `ixcom.log` file containing the list of non-synthesizable embedded modules. For the above list, add the following line in the <*dut_spec_file*> (after verifying library and view name in Xcelium database as explained in Format of the DUT Specification Information File) as follows:

```
other: worklib. FXPLL110HC0H_APGD:v
```

Multiple modules in the form of multiple `OTHER` specification lines can be added.

Using Xcelium Run-Time Profile Command to Control Profiling

To start profiling from a non-zero time or to reset profile data during simulation, instead of using the `-profile` option at the start of simulation, use the Tcl `profile` command. For more information, see Using the Tcl profile Command.

© 2021 All Rights Reserved.

Profiling Design Run-Time Performance on Hardware with XC Profiler

In SA mode, to analyze the performance of the design run on the Palladium Z1 emulator, and identify the bottlenecks to be removed to improve performance, use the run-time profile information and design statistics generated at the end of a simulation run. For this, you need to enable the XC profiler.

Enabling the XC Profiler

Use one of the following ways to enable the XC profiler:

- Specify the `+xcprof` option of the `xrun` or the `xeDebug --xmsim` command.

  Generating Hardware-Run Profiling Information Using +xcprof Option

- Use the TCL based `xc profon` command on the `xmsim>` prompt when using the xeDebug tool. (Generating Hardware-Run Profiling Information Using xc profon Command)

Note: Using the `+xcprof` option supersedes the `xc profon` command. When you use the `+xcprof` option, the hardware-run-time profiler is enabled for the entire session, and subsequent `xc profon` and `xc profoff` commands do not have any effect.

Generating Hardware-Run Profiling Information Using +xcprof Option

Specify the `+xcprof` option with any of the following commands to generate hardware-run-time profiling information for SA mode:

```
xrun -R [other_options] +xcprof
```

When you use the `-profile` option of the `xrun` command to enable Xcelium profiling, the verbose hardware-run-time profiler (`+xcprof` option) is not enabled by default. You need to explicitly specify the `+xcprof` option to enable the hardware-run-time profiling.

Or,

```
xeDebug --xmsim +xcprof [other_options]
```

The use of the `+xcprof` option does not incur a run-time penalty of more than 1-2% of the total run time of the design.

© 2021 All Rights Reserved.

The output of this option is generated in the `xcprof.out` file in the current working directory. It indicates the time spent in running the design on the hardware and software sides during the previous run.

When multi-channel GFIFO/SFIFO is enabled by using the compile-time option `+mc_gsfifo_config+<gsfifo_channel_config_file>`, the performance statistics for each OTB channel is reported in the `xcprof.out.<channel_name>` file. For more information, refer to the Running GFIFO-SFIFO Pairs in Parallel section.

Categories of Profiling Information in the XC Profiler Report (+xcprof Option)

The hardware-run-time profiling information for the design running on Palladium Z1 is generated in a detailed profiling report named `xcprof.out` in the current working directory (by default) when you enable the XC profiler.

You can modify the location and name of the profiler-output file by using the `+xcprofoutput=<filename>` option when you specify the `+xcprof` option as follows:

```
% xrun -R [other_xrun_options] +xcprof +xcprofoutput=prof_work/xcprof_1.out
```

The profiling information in the `xcprof.out` file is sorted into various categories of statistics as listed in the following table.

Each category is reported only if the count in the given run is non-zero.

Profiling Category	Profiling Information
Evals	Reports the number of simulation time steps for which the design is evaluated on emulator.
Behavioral evals	Reports the number of behavioral evaluations done per module. These per-module behavioral-evaluations occur concurrently. So, it should not be aggregated by summation to match with the global behavioral evaluations done during the simulation of the design.
cfclks (ECM controlled fclks)	Reports total number of FCLK (fast-clock) required for evaluation of the design.
gfifo-bwait evals (Gfifo behavioral wait cycles)	Reports total behavioral cycles spent in flushing gfifo.
gftbsync (Gfifo tbsync bwaits)	Reports total number of behavioral cycles required to drain the gfifo when tbsync happens.

© 2021 All Rights Reserved.

Profiling Category	Profiling Information
gbfull (Gfifo gb full bwaits)	Reports total number of behavioral cycles required to drain global buffer when the buffer gets full. A large percentage of execution time spent in the gbfull category means that the performance of your design is being constrained by the time it takes to execute the code in your GFIFO functions. To improve performance, you can try one and/or any of the following methods: ■ Modify your GFIFO functions to run faster. ■ Reduce the number of calls. ■ Use Parallel GFIFO/SFIFO as described in the Running GFIFO-SFIFO Pairs in Parallel and Improving Performance of Parallel GFIFO-SFIFO by Using Multiple Processes sections.
lbfull (Gfifo lb full bwaits)	Reports total number of behavioral cycles required to drain local buffer to global buffer when the local buffer gets full. A large percentage of execution time spent in the lbfull category means that you are invoking GFIFO functions much faster than the IXCOM run-time tool can collect and sort these GFIFO calls in hardware. To improve performance, you can try one and/or any of the following methods: ■ Reduce the number of calls. ■ Increase the local buffer size using the `+gfifo_lbsize` option described in the Options of the GFIFO Function section. ■ Use Parallel GFIFO/SFIFO as described in the Running GFIFO-SFIFO Pairs in Parallel and Improving Performance of Parallel GFIFO-SFIFO by Using Multiple Processes sections.
Sync Export Function Cycles (Sfifo asyncCall idle cycles)	Reports total number of cycles required to execute sfifo function calls during a tbsync.

© 2021 All Rights Reserved.

Profiling Category	Profiling Information
wall clock time	It is the elapsed time spent in `run` command or design evaluation after design swaps into hardware until it finishes execution.
command-line session time	It is end to end elapsed time since the design swapped into HW and finished execution. It includes time spend on interactive shell, idle time, TCL commands processing and execution.
emulator busy time	Time taken to execute the required number of FCLK for evaluating the design.
tbsyncs	Reports total number of testbench synchronizations happened during this session.
tbcall syncs	Reports number of synchronizations due to tbcalls (includes all synchronizations due to DPI calls, SVA action blocks, systask calls).
TB/DPI/SVA/SysTask calls	Reports the number of TBCalls, DPI-Calls, SVA, and System Task calls that happened during the run along with the line and file information for these calls. Also reports the time taken in executing the DPI calls, including the time required to pass the parameters between the software and hardware portions. You can use this information to determine if: ■ The parameters to the DPI calls should be passed using PIO or MEM ■ The DPI calls should be mapped to execute in only hardware or software. **Note:** Time-based profiling is not supported for VHDL TBCalls, HMPI, and those TBCalls that have not been processed by interface compilation.
Output event statistics	Reports the number of events on the DUT outputs
Input event statistics	Reports the number of events on the DUT Inputs
Force Release statistics	Reports the number of force-release operations on the DUT signals during the hardware run.

© 2021 All Rights Reserved.

Profiling Category	Profiling Information
Deposit statistics	Reports the number of deposits on DUT signals during the hardware run
Memory Read statistics	Reports the number of memory-read transactions issued to the hardware
Memory Write statistics	Reports the number of memory-write transactions issued to the hardware.

DPI-Profiling

The `Synchronous task and function call profile` section reports the details of the blocking `tb_import` and `tb_export` task and function calls during simulation.

```
-----------------------------------------------------------------------
              Synchronous task and function call profile
-----------------------------------------------------------------------
    Count Scope
-----------------------------------------------------------------------
    45454 top.s1.exptbc (Task, tb_export:tb_pio, iArgW=64, oArgW=64)
              Total life-time = 1.07 sec,  Loc:(test.sv, 32)
    166667 top.s1.imptbc (Task, tb_import:tb_pio, iArgW=64, oArgW=64)
              Total life-time = 0.08 sec,  Loc:(test.sv, 16)
```

The non-blocking GFIFO and SFIFO function calls are reported separately in their respective sections.

```
         GFIFO (import) profile: Total Execution time = 0.05 sec
-----------------------------------------------------------------------
               Count Scope
-----------------------------------------------------------------------
                1024 top.idut.sendData
         SFIFO (export) profile: Total Execution time = 0.10 sec
-----------------------------------------------------------------------
               Count Scope
-----------------------------------------------------------------------
                1024 top.idut.getData
```

Analysis and Interpretation of Report Generated with the +xcprof Option

This section provides example reports followed by the analysis of the information provided in these reports.

Example 1:

```
xcprof.out for (RUN Mode)
```


© 2021 All Rights Reserved.

```
==========================================================================
IXCOM Performance and Design Profile
--------------------------------------------------------------------------
--- Maximum HW operating speed (compiler fclk freq): 3906.25 KHz
--- Clocking Mode: Default (2X).

--- HW execution command line session time: 84.45 sec (83.97 CPU sec)
--- HW execution wall clock time          : 84.45 sec (83.97 CPU sec)
--- HW execution emulator busy time       : 2.10 sec (2.49%)

--- Simulation acceleration speed achieved: 97.24 KHz (97241.39 evals/sec)

--- HW executed
    --    8211920 ECM controlled fclk cycles      @ 97.24 KHz (97241.39 cfclks/sec)
    --    8211920 Eval (or sim-timestep) cycles  @ 97.24 KHz (97241.39 evals/sec)

--- Total number of HW-SW Synchronizations(tbSyncs) : 8211920 (1.00 evals/tbSync)
-- Synchronizations due to tbcalls(tbcallSyncs) : 0

        --- xc Profile: (%)
            82.21     HW-EMU-RUN (Elapsed: 69.43 sec)
            16.28     SW-SIM (Elapsed: 13.74 sec)
             1.51     SYNC-OH (Elapsed: 1.28 sec)

        --- HW-EMU : HW evaluations and Synchronization Latency.
        --- SW-SIM : TB, TBCalls, DPICalls, VPI/VHPI/PLI/E/SystemC.
        --- SYNC-OH: Synchronization Overhead of IXCOM Runtime.

--- Primary Inputs  (bits): INPUT(235) FORCE(0)DEPOSIT(0 :dut=0 + dpi=0 + ixcom=0).
--- Primary Outputs (bits): OUTPUT(65 :dut=65  dpi=0 + ixcom=0).
==========================================================================
==========================================================================
Output event statistics: 410565 output events
--------------------------------------------------------------------------
     Count Name
--------------------------------------------------------------------------
    410564 top.PD0.clrOut
         1 top.PD0.vldOut
==========================================================================

==========================================================================
Input event statistics: 18043191 Input events
--------------------------------------------------------------------------
     Count Name
--------------------------------------------------------------------------
   8211919 top.PD0.clk (SA clock)
    821191 top.PD0.slowclk (SA clock)
    205120 top.PD0.clrIn(42)
    205045 top.PD0.clrIn(2)
```

Analysis of the above report for Example 1 is as follows:

- The maximum operating FCLK frequency for this design is ~3.9 MHz. However, the acceleration achieved in this run is ~97 KHz.
- The time of `84.45` sec in the above report is the elapsed time during the simulation run after the design is swapped in to the hardware until the end of the particular hardware-run session. This could be at the end of simulation or at the time of swapping out of hardware. Out of this `84.45` sec, `82.21`% has been spent on the emulator, `16.28`% on the software simulator, that is for executing the TBCalls or while running the testbench portion, and `1.51`% in transferring data between hardware and software sides.

© 2021 All Rights Reserved.

- Channel latency per synchronization or the I/O latency is the latency for sending input values to hardware, evaluation, and getting output values from hardware. I/O latency excludes any time used for output decoding and TB-run-time communication through DPI calls.

  You can estimate the I/O latency by analyzing the elapsed emulation time (elapsed time in the `HW-EMU` category), and the emulator busy time.

  Channel latency per tbsync = (elapsed emulation time - emulator busy time) / #tbsyncs

- Based on the above results, the break-up of time spent on the hardware can be displayed.
  - Time spent in hardware_software synchronization = Channel Latency * #synchronizations = (elapsed emulation time - emulator busy time) = 67.33 s.
  - Time spent on hardware-Emu = DUT execution + hardware_software synchronization.
  - Time spent in execution of DUT on emu = emulator busy time = 2.1 s.
- In *RUN MODE,* the number of synchronizations is the primary contributor to performance.
- In *RUN MODE,* the hardware stops at all the `tb_delays` to synchronize the DUT inputs, deposit, force, and memory write operations.
  - From the input-output event section, it can be inferred that the hardware has to stop to synchronize on the clock signals.
- The primary cause of synchronization is the testbench fast clock (`top.PD0.clk`).

Example 2:

```
xcprof.out for (Optimized SA -TBRUN Mode)
=========================================================================
IXCOM Performance and Design Profile
-------------------------------------------------------------------------
--- Maximum HW operating speed (compiler fclk freq): 3571.43 KHz
--- Clocking Mode: Default (2X).

--- HW execution command line session time: 26.44 sec (26.00 CPU sec)

--- HW execution wall clock time          : 26.44 sec (26.00 CPU sec)
--- HW execution emulator busy time       : 3.22 sec (12.17%)

--- Simulation acceleration speed achieved: 310.58 KHz (310584.16 evals/sec)

--- HW executed
   --   11496688 ECM controlled fclk cycles     @ 434.82 KHz (434817.87 cfclks/sec)
   --    8211919 Eval (or sim-timestep) cycles @ 310.58 KHz (310584.16 evals/sec)
```


© 2021 All Rights Reserved.

```
--- Total number of HW-SW Synchronizations(tbSyncs) : 1642384 (5.00 evals/tbSync)
   -- Synchronizations due to tbcalls(tbcallSyncs) : 0

--- xc Profile: (%)
            70.63     HW-EMU-TBRUN (Elapsed: 18.67 sec)
            27.26     SW-SIM (Elapsed: 7.21 sec)
             2.11     SYNC-OH (Elapsed: 0.56 sec)

        --- HW-EMU : HW evaluations and Synchronization Latency.
        --- SW-SIM : TB, TBCalls, DPICalls, VPI/VHPI/PLI/E/SystemC.
        --- SYNC-OH: Synchronization Overhead of IXCOM Runtime.

--- Primary Inputs  (bits): INPUT(234) FORCE(0)DEPOSIT(0 :dut=0 + dpi=0 + ixcom=0).
--- Primary Outputs (bits): OUTPUT(129 :dut=65  dpi=64 + ixcom=0).
==========================================================================

==========================================================================
        Output event statistics: 410565 output events
--------------------------------------------------------------------------
     Count Name
--------------------------------------------------------------------------
    410564 top.PD0.clrOut
         1 top.PD0.vldOut
==========================================================================

==========================================================================
        Input event statistics: 9831272 Input events
--------------------------------------------------------------------------
         Count Name
--------------------------------------------------------------------------
    821191 top.PD0.slowclk (SA clock)
    205120 top.PD0.clrIn(42)
```

You can can change the testbench and the DUT in SA mode so that instead of the testbench sending the faster clock (clk) to DUT, the DUT generates the faster clock, thereby improving the number of required synchronizations from 8211920 to 1642384.

Analysis of the above report for Example 2 is as follows:

- The synchronizations happen only due to slowclock (821191 `top.PD0.slowclk` input events).
- The acceleration achieved in this run is ~310.58 KHz, due to the following:
  - Only 70.63% of total run time is spent on hardware (18.67 sec out of 26.44 sec).
  - Of this 70.63% of the time spent on hardware, the hardware needed to execute 1642384 synchronizations with each synchronization having a latency of 9.4 us.
- Following is the break up of time spent on hardware:
    - Time spent in hardware_software synchronization = Channel Latency * #synchronizations = (elapsed emulation time - emulator busy time) = (18.67-3.22) = 15.45 s.
    - Time spend on hardware-Emu = DUT execution + hardware_software synchronization.

© 2021 All Rights Reserved.

- Time spent in execution of DUT on emu = emulator busy time = 3.22 s.

As can be seen, the bottleneck is still the number of hardware_software synchronizations. So, to further improve the performance, the DUT-testbench communication can be reduced by using TBA as shown in the following example.

Example 3:

```
xcprof.out for (TBA using SCEMI pipes)
=========================================================================
IXCOM Performance and Design Profile
-------------------------------------------------------------------------
--- Maximum HW operating speed (compiler fclk freq): 4166.67 KHz
--- Clocking Mode: Default (2X).

--- HW execution command line session time: 5.26 sec (4.67 CPU sec)
--- HW execution wall clock time          : 5.26 sec (4.67 CPU sec)
--- HW execution emulator busy time       : 1.97 sec (37.46%)

--- Simulation acceleration speed achieved: 1560.62 KHz (1560619.28 evals/sec)

--- HW executed
    --   8212071 ECM controlled fclk cycles       @ 1560.65 KHz (1560649.88 cfclks/
sec)
    --   8211910 Eval (or sim-timestep) cycles   @ 1560.62 KHz (1560619.28 evals/
sec)
   --- Total number of HW-SW Synchronizations(tbSyncs) : 54 (152072.41 evals/
tbSync)
       -- Synchronizations due to tbcalls(tbcallSyncs) : 53 (154941.70 evals/
tbcallSync)

--- xc Profile: (%)
            57.46     SW-SIM (Elapsed: 3.02 sec)
            40.78     HW-EMU (Elapsed: 2.15 sec)
             1.54     HW-MEM (Elapsed: 0.08 sec)
             0.21     SYNC-OH (Elapsed: 0.01 sec)

        --- SW-SIM : TB, TBCalls, DPICalls, VPI/VHPI/PLI/E/SystemC.
        --- HW-EMU : HW evaluations and Synchronization Latency.
        --- HW-MEM : Memory Access (read/write) from/to HW.
        --- SYNC-OH: Synchronization Overhead of IXCOM Runtime.

--- Primary Inputs  (bits): INPUT(0) FORCE(0)DEPOSIT(69 :dut=69 + dpi=0 + ixcom=0).
--- Primary Outputs (bits): OUTPUT(101 :dut=0  dpi=99 + ixcom=2).
=========================================================================

=========================================================================
--- Synchronizations due to TB/DPI/SVA/SysTask calls (tbcallSyncs): 53.
-------------------------------------------------------------------------
        TBCALL statistics: 53 tb-calls
-------------------------------------------------------------------------
     Count Scope
-------------------------------------------------------------------------
        27 top.PD0.bfm_ins.my_input_pipe.<protected>
              Loc: (/lan/cva_rel/uxe151/15.1.0.178/tools.lnx86/etc/tba/
scemi_pipe21.svp, 206)

        26 top.PD0.bfm_ins.my_output_pipe.<protected>
              Loc: (/lan/cva_rel/uxe151/15.1.0.178/tools.lnx86/etc/tba/
scemi_pipe21.svp, 705)
```


© 2021 All Rights Reserved.

```
=========================================================================
        Memory Read statistics: 26 mem reads
-------------------------------------------------------------------------
   Count     Bytes         sec     MB/sec          Name
-------------------------------------------------------------------------
     26      4926996      0.0343  143.540  top.PD0.bfm_ins.my_output_pipe.buffer
-------------------------------------------------------------------------
     26      4926996      0.0343  143.540  Total
-------------------------------------------------------------------------

        Memory Writes statistics: 26 mem writes
-------------------------------------------------------------------------
   Count     Bytes         sec     MB/sec          Name
-------------------------------------------------------------------------
     26      4926996      0.0373  132.109  top.PD0.bfm_ins.my_input_pipe.buffer
-------------------------------------------------------------------------
     26      4926996      0.0373   132.109  Total
-------------------------------------------------------------------------

--- Memory Access during Swapin 1 times, transferred 196608 bytes in 0.0074 sec
(26.547 MB/sec)
```

Analysis of the above report for Example 3 is as follows:

- Maximum hardware-operating speed (fclk freq): 4166.67 KHz.
- The acceleration achieved in this run is ~1560.62 KHz, due to the following:
  - Only 40.78% of total run time is spent on hardware (2.15 sec out of 5.26 sec).
  - Of this 40.78% of the time spent on hardware, the hardware had to execute 53 synchronizations.
- The breakup of time spent on hardware is as follows:
    - Time spent in hardware_software synchronization = Channel Latency * #synchronizations = (elapsed emulation time - emulator busy time) = 2.15 s sec - 1.97 sec = 0.18 sec.
    - Time spend on hardware-Emu = DUT execution + hardware_software synchronization.
    - Time spent in execution of DUT on emu = emulator busy time = 1.97 sec.

As can be seen in this TBA testcase, the synchronization is no longer the bottleneck in performance. The bottleneck is that 57.46% of total run time is in software simulation to service the 53 tbcalls. You can identify these tbcalls from the report and optimize them to further improve the performance.

© 2021 All Rights Reserved.

Generating Hardware-Run Profiling Information Using xc profon Command

The xc profon command profiles the simulation time during the hardware run. You can use the xc profon command at any time before executing the xc on -run and xc on -tbrun commands. The hardware run-time profiling process begins only after the design states have been swapped into the hardware.

Here is a comparison of the use of xc profon command with the use of the +xcprof option:

- The xc profon command is a Tcl command that enables you to switch the profiler on or off using the TCL interface ***during*** the run-time session. This is different from the use of the +xcprof option, which must be specified ***before*** the run to enable the profiler.
- The xc profon command provides a basic-level profiling of ***smaller sessions*** within the whole simulation, and generates a basic report in the **xrun.log** file to indicate the time spent in various categories. This is unlike the +xcprof option that works as a more advanced profiler for the ***complete run-time session***, and generates a comprehensive performance report in a separate **xcprof.out** file.

To end the profiling session, use the xc profoff command. If you do not issue the xc profoff command, the profiler automatically stops when the simulation finishes.

Analyzing and Interpreting the Profiling Output of the xc profon Command

When the profiler stops, results like the following are displayed in the xrun.log file:

```
--- HW execs 8211919 evals 0 bevals 11496688 cfclks 1642384 tbsyncs in 30.67
sec (3.73 us/eval; 267758.84 eval/sec; 30.12 CPU sec)
--- HW execs emulator busy 2.68 sec (8.72%)
--- HW execs 0 force/deposit 9831272 input events
--- HW execs 410565 output events
--- HW execs emulator command line session time 30.67 sec (30.12 CPU sec)
--- xc Profile: (%)
    75.05                 HW-EMU-TBRUN   (Elapsed: 23.02 sec)
    22.70                 SW-SIM         (Elapsed: 6.96 sec)
     1.53                 XCRT-OH        (Elapsed: 0.47 sec)
     0.72                 DECODE         (Elapsed: 0.22 sec)
```

Here is a brief description of the values in this report:

- The HW-EMU-<MODE> value is the percentage of time spent in hardware evaluation. This includes the time spent in sending the DMA input to the hardware, initiating hardware evaluation, polling the hardware for stop conditions, and sending the DMA output back to the software simulator.
- The SW-SIM value is the percentage of time spent in Xcelium space during the profile time. This includes the simulation events in testbench modules, the tbcall tasks triggered

© 2021 All Rights Reserved.

by hardware tbcall calls, and any overheads imposed by the VPI, VHPI, and DPI calls to the Xcelium simulator.

- The `DECODE` value is the percentage of time spent in decoding the hardware output changes.

  If many DUT outputs change very frequently, this time can be significant.

- The `XCRT-OH` value is the percentage of time spent in the rest of the run-time execution.

When a design is profiled using the `-profile` option of the `xrun` command, the profiling information is automatically reported in `ncprof.out`, as shown below.

The categorization of information in the above-mentioned buckets indicates that:

- The time reported under the `HW-EMU` category is the time spent when the design was running on the hardware. It includes the time spent in executing the DUT on hardware and the I/O latency delays in the hardware-software synchronizations.
- The times reported in the `xc_runtime` and `decode` categories are the typical overhead times for accessing and transferring the DUT between the Palladium Z1 emulator and the Xcelium simulator.
- The time reported in the `IUS` category is the time spent in the non-DUT portion of the design like the testbench or the tbcall tasks, and IXCOM-instrumentation overhead (like the xc_top directory).

Example: Sample output of XC Profiler in `ncprof.out`

```
------------------------------------------------------------
IXCOM Runtime Profiler Report (985 hits,  97.5% of total)
------------------------------------------------------------
%hits   #hits      %xcRun      name             comments
  3.2      32        3.2       XCRT-OH          IXCOM runtime Control (Overhead).
 28.8     291       29.5       SW-SIM           TB and TBCall execution in
                                                sw simulator
  3.0      30        3.0       DECODE           HW O/P change decode (Overhead).
 62.6     632       64.2       HW-EMU           Time spent in HW evaluation & Sync
                                                Latency
```


© 2021 All Rights Reserved.

Integration with Advanced Profiler - xprof

This section describes the integration of IXCOM with the Advanced Profiler - xprof. For details about xprof, refer to the *Advanced Profiler* document available in the *Xcelium Documentation* set.

Design profiling in IXCOM can be broadly categorized into *Software Simulation Mode Profile* that is used to predict the performance of design on Palladium Z1, given a pure Xcelium simulation run, and *Hardware Mode Profile* that is used to explain the performance of the given hardware run on Palladium Z1, and identify and analyze bottlenecks to improve performance further.

The xprof performance profiling process is divided into two phases:

- Activation and Collection: This involves collection of run-time data.
- Analysis (offline): This involves the GUI and batch mode commands that help visualize the collected data and enable smart analysis of performance.

The run-time information is collected along the run. IXCOM communicates with xprof and provides the run-time information (CPU usage) for collection. This raw information is read and dumped into a unified database for further analysis.

Collecting Data for Software Simulation Mode

For Software simulation mode, if Speed-up information (sub-section `Average Case Performance Estimate`) is required, specify the -`hwperf_estimate` elaboration option:

```
xrun -xprof [-hwperf_estimate] \
     -DUT_PROF <dut_specification_file> \
     [other xrun options] <source_files>
```

The `-xprof` option is an `xmelab` argument; not an `ixcom` argument. So, you must tell `ixcom` or `xrun` to pass on the argument to `xmelab`. Choose any one of the following ways to do this:

```
xrun -hw -ua -xmelabargs "-xprof"
xrun -hw -ua -xmelabargs "-xprof -hwperf_estimate"
```

OR

```
xrun -hw -ua +profile=xprof
```

OR

Create `uxc_hdl.var` file in the working directory and include the following line in this file:

```
DEFINE XMELABOPTS ?xprof
```


© 2021 All Rights Reserved.

Note that the `-DUT_PROF` option is the same as for the Xcelium profiler tool (Commands for Design Profiling Using the Xcelium Profiler). Use the <*dut_specification_file*> file to specify the DUT portion of the design environment to Xcelium. This file must include the name of at least one design unit that is considered to be top level of the DUT RTL/Gate hierarchy, and is assumed to be acceleratable on the Palladium Z1 system (that is, corresponding to `+dut+` option of the `ixcom` command).

Use the following format to specify the DUT in the <*dut_specification_file*> file is:

```
# comments can also be added
dut: <lib>.<module>:<view>
```

For the specified module, all its sub-hierarchies are marked in profiling as DUT design units. Multiple DUT design units and their sub-hierarchies can be specified.

For example:

```
# multiple top level DUT modules
dut: worklib.sontop:v
dut: worklib.bidi:v
dut: chip.data_grab:arch_grab
```

Note: It is mandatory to specify the library and view name in Xcelium format. To find out a the library and view name for a module, use the `xmls` utility. The `xmls` utility lists compiled objects stored in the library system and displays various attributes and information about those objects.

Collecting Data for Hardware Mode

For Hardware mode, specify only the `-xprof` option at elaboration time; the `-hwperf_estimate` option is not needed.

At run time, specify the `-dutProf` option with the `--xmsim` option of the `xeDebug` command:

```
xeDebug --xmsim -dut_prof <file> -- <run_script>
```

Optionally, include the `-dut_prof` option at run time with the `xmsim` command (multi-step mode) or the `xrun` command (single-step mode) to specify the DUT portion of the design environment.

Using the xprof GUI to View Profile Reports

The GUI mode is the default mode for performance run analysis. To invoke the GUI for Unified Profiler, use the `xprof` command. Then, open the database by clicking the *Open* button in the xprof GUI.

© 2021 All Rights Reserved.

In the xprof GUI, for Software Mode profiling, the `Design Unit Time Summary` and `Potential Speedup Estimation` sections are displayed in the DUT profiling report file. This information is displayed only when the `-dut_prof` option is included with `xmsim` or `xrun` command.

Following figure displays the *General Information* window of the xprof GUI. Note that the last row displays `IXCOM Speedup Estimation Information` and the path of the DUT profiling report file:

If the `-dut_prof` option was specified with the `xmsim` or `xrun` command, the following report file is displayed when the `IXCOM Speedup Estimation Information` tab is

© 2021 All Rights Reserved.

double-clicked in the `General Information` window on the `Welcome Page` of the xprof GUI:

In addition, the xprof GUI displays the `DUT marking (DUT/OTHER)` information based on the <*`dut_specification_file`*> specified with the `-dut_prof` option. The data can be filtered based on the `DUT/OTHER` information so that it is possible to analyze and determine the `DUT/OTHER` instances that are causing the performance bottleneck.

© 2021 All Rights Reserved.

Following figure displays a sample of the DUT marking information:

Environment Hierarchy

Instances Types Categories

Name	Self	Self+	DUT Type
Instances	n/a	73.76%	n/a
testbench	7.21%	73.58%	OTHER
top_i1	0.11%	64.31%	DUT
ext_source	0.32%	0.32%	OTHER
adc3_sim	0.24%	0.24%	OTHER
panel_if	1.5%	1.5%	OTHER
xc_top	0.18%	0.18%	OTHER
pomod_82	0.0%	0.0%	OTHER
Others	n/a	26.24%	n/a
Simu Name: Others he	0.0%	25.37%	n/a
IUS Setup	0.0%	0.33%	n/a
Support for VPI callbacks (or UI)	0.0%	0.31%	n/a
File IO	0.0%	0.16%	n/a
Engine support	0.0%	0.02%	n/a

For hardware mode profiling, the `IXCOM Runtime Profile` report is displayed in `Run Analysis Page` under `Others`.

The `IXCOM Runtime Data` category has the following IXCOM run-time specific CPU usages buckets:

- **`SYNC-OH`**: Synchronization Overhead
- **`HW-EMU`**: Hardware evaluations and Synchronization Latency
- **`HW-MEM`**: Memory Access from/to hardware
- **`DEPOSIT`**: Deposit signals in hardware
- **`HMPI`**: HMPI Tasks and Functions

Caution

The HMI/HMPI feature is being deprecated in VXE 20.05 release. It will not be supported in future releases.

© 2021 All Rights Reserved.

The following figure displays a sample `IXCOM Runtime Profile` report:

Using xprof in Batch Mode

The batch mode (CLI) for xprof can be invoked for analysis by using the `xprof -batch` command after the performance simulation run is complete.

Using the `xprof -batch` command displays the `xprof>` prompt where the following commands can be specified to view the profile data in the software simulation and hardware modes:

- Software Simulation mode:
  - To view estimation report, use the following command:

    ```
    xprof> show -dut_prof_report
    ```

  - To view `DUT/OTHER` marking information and determine if an instance lies on DUT:

    ```
    xprof> summary -inst
    ```

- Hardware Mode:

  To check the hardware evaluations and delays caused due to synchronization, use the following command:

  ```
  xprof> summary -category -filter HW-EMU
  ```

The above-mentioned commands are specific to IXCOM. For details of batch mode reporting and other commands of the Advanced Profiler, refer to the *Advanced Profiler* document available in the *Xcelium documentation set*.

© 2021 All Rights Reserved.

Following are some examples of using the above commands.

For Software Mode Profiling, the `show -dut_prof_report` command displays the data for `Design Unit Time Summary` and `Potential Speedup Estimation` for software mode run.

For example, the `show -dut_prof_report` command displays the following information:

```
-------------------------------------------------------------------------------
-----------------------------------------------
DUT Time Summary
=================
%Hits                Set

----------------------------------------
99.44% (1427/1435)  Design Unit
0.00% (0/1435)      Other Unit
0.49% (7/1435)      IUS Setup
0.07% (1/1435)      Outside engine
Best Case Performance Speedup Estimate in IXCOM 'Run Mode'
==========================================================
Time spent in DUT = 99.44%
Estimated Best case speedup = 179.37X
Average Case Performance Speedup Estimate in IXCOM 'Run Mode'
=============================================================
Simulation runtime Data
-----------------------
SW simulator executes @cyctes/sec = 1000
Estimated number of HW-SW I/O sync = 0
Performance Estimation
----------------------
SW simulation runtime (sec) = 5.83
Time spent in SW sim of non-DUT portion (sec) = 0.03
Estimated COSIM runtime (sec) = 0.03
Estimated Average case speedup = 179.37X
```

The `Potential Speed-up Estimation in IXCOM 'Run Mode'` section is meaningful only for Software Mode Profiling. Therefore, this section should not be dumped for hardware mode profiling.

The `Best Case Performance Speedup Estimate in IXCOM 'Run Mode'` is based on the following assumptions:

- Time spent in hardware-software synchronizations and executing the DUT on Palladium Z1 is negligible compared to time spent in Testbench.
- Time taken by the Testbench (software) portion of the design to execute on simulator remains identical in COSIM run and the present pure-simulation run.

In the sub-section `Best Case Performance Speedup Estimate in IXCOM 'Run Mode'`, the `Aggregated Simulation Overhead (ASO)` is divided between `DUT` and `DUT/Other`. This division is done only when `ASO` is greater than `0`.

© 2021 All Rights Reserved.

The `Average Case Performance Estimate` sub-section is reported only when the `-hwperf_estimate` option is enabled at elaboration.

`DUT runtime on HW (incl. HW-SW I/O Sync Overhead)` is calculated assuming a typically value of 13 us penalty per hardware-software synchronization.

For example, if the estimated number of HW-SW I/O Sync is 40179, then, the calculation is as follows:

`DUT runtime on HW (incl. HW-SW I/O Sync Overhead) = 40179 * (13 * 0.000001) sec = 0.52 sec.`

The `DUT marking information` for instance with the `summary -inst` command is reported by default, if the `-dut_prof` option is specified during the simulation run. Use the `-noduttype` option to disable the DUT marking information for the `report` command.

For example, the `summary -inst` command displays the following information:

```
xprof> summary -inst
Self+ %   Self %   DUT Type   Instance's Hierarchical name
--------  -------  ---------  ------------------------------
65.85 %   65.85 %  DUT        top.i1
33.59 %   33.59 %  DUT        top.i2
```

With the `-noduttype` option, the DUT marking information for instance is not shown.

```
xprof> summary -inst -nodutytpe
```

Following is a sample output of the above command:

```
Self+ %   Self %   Instance's Hierarchical name
--------  -------  ------------------------------
65.85 %   65.85 %  top.i1
33.59 %   33.59 %  top.i2
```

Similarly, using the `detail -inst` option displays `(DUT)/(OTHER)` with reported instance names.

For hardware-mode profiling, to find the `HW evaluations and Synchronization Latency` CPU usage, use the following command:

```
xprof> summary -category -filter HW-EMU
```

Following is a sample output of the above command:

```
Self+ %   Self %   Name
--------  -------  -----
69.17 %   69.17 %  HW-EMU: HW evaluations and Synchronization Latency
```

To determine the `Synchronization Overhead` CPU usage, use the following command:

```
xprof> summary -category -filter SYNC-OH
```


© 2021 All Rights Reserved.

Following is a sample output of the above command:

```
Self+ %   Self %   Name
--------  -------  -----
4.91 %    4.91 %   SYNC-OH: Synchronization Overhead
```

If required, create an alias for the command to reduce the characters to be typed. For example:

```
xprof> alias HW-EMU-REPORT "summary -category -filter HW-EMU"
xprof> HW-EMU-REPORT
Self+ %    Self %  Name
--------  -------  -----
69.17 %   69.17 %  HW-EMU: HW evaluations and Synchronization Latency
```


© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

24

Guidelines for Writing Testbenches

The simulation process consists of two components that together allow you to simulate almost any Verilog design:

- The emulator, which accelerates *synthesizable* Verilog code
- The Xcelium software simulator, the co-simulation component, which simulates the *non-synthesizable* parts of your design, including C/C++ code linked through the Programming Language Interface (PLI)

In a typical design, the *chip*, or Device Under Test (DUT), contains mostly or entirely synthesizable code, while the *testbench* contains non-synthesizable, behavioral-style code that cannot be accelerated with the emulator.

This chapter provides you with guidelines for writing testbenches that can be accelerated by the emulator.

Important

The guidelines provided in this chapter are applicable only to SA mode.

Accelerating even a small portion of the testbench can provide significant overall performance improvement. The following aspects of testbench development are covered:

- Following a Synthesizable Coding Style
- Using Initial Blocks (Verilog Only)
- Specifying Clocks
- Applying Stimulus to Your Design
- Avoiding Testbench Race Conditions (Verilog Only)
- Using Hierarchical References in Testbench (Verilog Only)
- Accessing Memories Using Hierarchical References
- Using Verilog PLI in the Testbench

© 2021 All Rights Reserved.

- Dealing with Testbench Monitoring Code

Note: While it might be a good idea to change your testbench to improve performance, Cadence does not recommend that you make changes to the chip itself for this purpose. You should always follow a methodology that will allow you to best realize the chip into actual hardware.

Following a Synthesizable Coding Style

To maximize simulation performance, you should write as much of your testbench as possible in the *synthesizable* coding style.

In addition to the Verilog and VHDL constructs allowed by the synthesis subset defined by Synopsys, IXCOM also lets you make use of Verilog cross-module references (XMRs) or VHDL global signals to read registers and wires inside the Device Under Test. This feature is especially useful for accelerating *monitor* code, which is contained by many testbenches. When writing the testbench, try to keep all non-synthesizable code in separate modules; these will not be accelerated by the emulator.

Using Initial Blocks (Verilog Only)

IXCOM supports the use of Verilog `initial` blocks anywhere in your design, even in the Device Under Test (DUT). The DUT is automatically run in software simulation mode for at least one simulation time-tick to execute the `initial` block statements.

`initial` blocks are especially useful for initializing registers and memories and for applying basic stimulus to the design. If you are using an `initial` block in a module running in the software simulator only, then you can use any construct that Verilog allows to add to `initial` blocks.

However, if the `initial` block is used in a module in the DUT that can be accelerated, certain constructs will delay the time when you can swap your design to Simulation Acceleration mode or prevent it from happening altogether. This is because all events generated by `initial` blocks must be completed while in software simulation mode before swapping can occur. These constructs are described in the following section.

This section contains the following:

- Initial Block Statements That Delay Swapping to Hardware
- Initial Block Statements That Prevent Swapping to Simulation Acceleration Mode

© 2021 All Rights Reserved.

Initial Block Statements That Delay Swapping to Hardware

This section tells you about `initial` block statements that *delay* swapping to Simulation Acceleration mode.

- *Delay specifications*: The design cannot be swapped to Simulation Acceleration mode until the delay period is over. If there are different delays in various `initial` blocks, the design cannot be swapped until the longest delay period is over, for example:

```
module dut(..);
reg a;
initial
begin
    #10;            // cannot swap design to Simulation Acceleration
                    // or emulation mode until after #10
    a = 1'b0;
end
endmodule
```

  Note: Inserting delays in `initial` blocks does not typically reduce simulation performance if reasonable delay values are used.

- *Event control statements*: The design cannot be swapped to Simulation Acceleration mode until after the event is simulated, for example:

```
module dut(..);
reg clk;
initial
begin
    @(posedge clk);    // cannot swap to Simulation Acceleration mode until
first posedge on clk
    a = 1'b1;
end
endmodule
```

- *Wait statements*: The design cannot be swapped to Simulation Acceleration mode until all events are simulated.

Initial Block Statements That Prevent Swapping to Simulation Acceleration Mode

This section tells you about `initial` block statements that *prevent* swapping to Simulation Acceleration mode.

- *Forever statements*: Since events are generated continuously, the design can *never* be swapped to Simulation Acceleration mode .

```
module dut(..);
reg clk;
initial
begin
    clk = 1'b0;
forever #10 clk = ~clk;    // generates events forever--cannot swap
```


© 2021 All Rights Reserved.

```
                                // design to Simulation Acceleration mode
    end
    endmodule
```

Refer to Specifying Clocks on page 1172 for details on using `forever` statements for this reason.

- *PLI (Programming Language Interface) calls*: These calls persist throughout simulation. So, the design cannot be swapped to Simulation Acceleration mode.

```
module dut(..);
reg a;
initial
    $ARM7TDMI(...); // This PLI call persists throughout simulation;
                    // cannot swap design to Simulation Acceleration modes
endmodule
```

Note: The preceding `initial` block statements affect swapping to Simulation Acceleration mode *only* if used in a DUT module. Modules running exclusively in the software simulator (that cannot be accelerated) do not affect the swapping capability.

If you need to use any of the preceding constructs, move the `initial` block to a module not in the DUT.

Specifying Clocks

There are two common methods for specifying design clocks.

Method 1

```
initial
begin
    clk = 1'b0;
    forever #10 clk = ~clk;
end
```

Method 2

```
initial
    clk = 1'b0;
always
begin
    #10;
    clk = ~clk;
end
```

To accelerate the code for the above methods, Cadence recommends that a separate clock-generation module be added to the design. Then, specify this module with the `+iscDelay` option of the `ixcom` command.

© 2021 All Rights Reserved.

Here is an example of a separate clock-generation module:

```
module synthesizable_testb ();
wire sys_clk;
clk_gen clk_gen1(sys_clk);
chip chip1 (sys_clk, ...);
..
<synthesizable code>
..
endmodule
module clk_gen(clk);
output clk;
initial
begin
    clk = 1'b0;
    forever #10 clk = ~clk;
end
endmodule
```

Here is the way you would run the compiler to exclude this clock-generation module from the DUT:

For a mixed-HDL or VHDL Design:

```
ixcom -top synthesizable_testb -ua +dut+synthesizable_testb +iscDelay+clk_gen
```

For a Verilog-only Design:

```
ixcom testbench.v chip.v -ua +dut+synthesizable_testb +iscDelay+clk_gen
```

Alternatively, you can include the `$ixc_ctrl` system task to ensure that the clock-generating module is mapped to the hardware.

```
initial $ixc_ctrl("map_delays");
```

Then, the module need not be specified with the `+iscDelay` option of the `ixcom` command.

Applying Stimulus to Your Design

This section describes how to apply stimulus to your design and contains the following sections:

- Synchronous Testbenches
- Applying Test Vectors Using Memory ($random)

Synchronous Testbenches

Many testbenches use delay specifications for applying stimulus to the design at specific intervals of time, for example:

© 2021 All Rights Reserved.

```
initial
begin
    reset = 1'b0;
    #10 reset = 1'b1;
    #10 start_mem_cycle = 1'b1;
end
```

Since delay specifications are non-synthesizable, this coding style cannot be accelerated. Instead, use the edge of a clock to apply stimulus to the design. In the following synthesizable example, stimulus is applied on each negative edge of the clock. Since the design uses the positive edge, this ensures the timely application of stimulus before it is actually used by the design.

```
initial
    reset = 1'b0;
always @(negedge clock)
begin
    reset = 1'b1;
    @(negedge clock);   // Note: Must use same clock to be synthesizable
    start_mem_cycle = 1'b1;
    @(negedge clock);
         ...
end
```

Even if the intent is not to write a synthesizable testbench, Cadence recommends that you make use of clocks, instead of delays, to apply stimulus. The emulator simulates with zero delay and consequently might not exhibit the delay relationships between signals that you would expect. Applying stimulus on clock-edges results in much more predictable behavior.

Applying Test Vectors Using Memory ($random)

A common method for applying stimulus is by using random test vectors generated with the `$random` system task (Verilog only). The test vectors are then applied to nets and registers in the design at regular intervals. However, the `$random` system task is non-synthesizable, so any module containing this task will not be accelerated.

An alternate method of applying randomly generated test vectors (or any other stimulus) that can be accelerated involves the use of special memories. Here is how this technique works:

1. Generate the test vectors outside of simulation, using the `rand` C function in a C program, or using a `perl` script, and write to an ASCII-formatted file.
2. Create a read-only memory structure.
3. Load the test vectors into the ROM using an `initial` block.
4. Read the ROM at specified intervals to extract the desired vectors.

© 2021 All Rights Reserved.

Example

In this example, the 32-bit randomly generated vectors are in a file, `random.hex`, and are loaded into the ROM at initialization, then read into the circuit one vector at a time on the posedge of the system clock:

```
module testbench ();
wire sys_ clk;
reg [31:0] address;
reg [31:0] data;
clk_gen clk_gen1(sys_clk);            // clock-generator in above example
reg [31:0] vmem [0:65535];
always @(posedge sys_clk)
       data <= vmem[addr];

initial
begin
address = 0;                          // initialize ROM address counter
$readmemh("random.hex",testbench.vmem);   // load test-vectors into ROM
end
always @(posedge sys_clk)
address <= address + 1;            // Increments ROM address counter
endmodule
```

Avoiding Testbench Race Conditions (Verilog Only)

Some designers make use of `#delays` of various values to establish certain delay relationships between various signals, such as clock and data, in their designs. The emulator simulates with zero delay. Therefore, this strategy might not yield the desired results when using the emulator.

An alternate method that works as well as other simulators is to make use of blocking versus nonblocking assignment statements to establish relationships between signals. The most common race condition occurs between the data and clock signals to a flip-flop. In most cases, you would want the data signal to change value only after the clock edge used by the flip-flop, that is, the flip-flop would store the old value of the data signal. In this case, use blocking assignment statements when generating the clock signal and nonblocking assignment statements when assigning the data signal, for example:

```
module synthesizable_testb ();
wire sys_clk;
clk_gen clk_gen1(sys_clk);
always @(posedge sys_clk)
 q <= d;                      // nonblocking - old value of d assigned to q
endmodule
module clk_gen(clk);
output clk;
initial
begin
 clk = 1'b0; // blocking assignment on clock - evaluates first
 forever #10 clk = ~clk;
end
endmodule
```


© 2021 All Rights Reserved.

Using Hierarchical References in Testbench (Verilog Only)

Many testbenches make use of hierarchical references (also known as cross-module references or XMRs) to reference nets and registers in the Device Under Test (DUT), as well as in other parts of your design. Hierarchical references can be used to initialize memories or to probe the values of signals during simulation.

When writing your testbenches, keep the following in mind, especially if you are attempting to accelerate parts of your testbench by adding them to the DUT:

- A special I/O signal is created for each DUT net or register bit referenced in a testbench module or other module running in the software simulator. Such I/Os incur a performance penalty for uploading and downloading their values through the PCI bus. To maximize simulation performance, keep hierarchical references of this type to a minimum.
- You can reference nets and registers in the DUT from your testbench only if they are being read. You can write to registers using system tasks, although this routine is not synthesizable and reduces simulation performance. Similarly, for a VHDL design, you can write to registers *that are edge-triggered flip-flops* in the DUT using system tasks. You cannot write to DUT nets.
- You can add testbench modules that are otherwise synthesizable, except for the usage XMRs, to the DUT to be accelerated. However, keep the following rules in mind:
  - You can reference nets and registers from one DUT module to another for both reading and writing.
  - You cannot reference nets and registers in the testbench or other modules running in the software simulator, for example:
    - For a mixed-HDL or VHDL Design:

```
ixcom -top top_level_design_unit other_options +dut+chip+testb_in_dut
```

    - For a Verilog-only Design:

```
ixcom Verilog_sources other_options +dut+chip+testb_in_dut
module test ();
reg sys_clk;                // testbench clock signal
chip chip1 (..);
testb_in_dut dut2 (..);
endmodule
module testb_in_dut(..);
reg int_clk;
initial
begin
int_clk = test.sys_clk; // reference to a testbench signal not
             // allowed since module testb_in_dut is in the DUT
```


© 2021 All Rights Reserved.

```
    end
    endmodule
```

There is a workaround for this issue, assuming you are referencing the DUT net or register in an `initial` block, as in the previous example. The workaround is to use system tasks. This task, which has no parameters, returns a boolean `true` if the design is currently running in the software simulator or `false` if running in Simulation Acceleration mode. Since all designs are run at least 1 timestep in software simulation mode before swapping to Simulation Acceleration mode, `initial` blocks are always executed. For the previous example, make the following change:

```
module test ();
    reg sys_clk;                 // testbench clock signal
    chip chip1 (..);
    testb_in_dut dut2 (..);
endmodule

module testb_in_dut(..);
    reg int_clk;
    initial
    begin
      `ifdef IXCOM_UXE
        if ($axis_rccoff)        // code enabled in software simulation mode
        int_clk = test.sys_clk;  // XMR simulated as expected
        `else
      `endif
    end
endmodule
```

The `` `ifdef IXCOM_UXE `` compiler directive automatically instructs the simulator to include the `$axis_rccoff` system task, but it will be ignored by other simulators.

Some testbenches use Verilog `force` and `release` statements to set reference nets and registers to desired values. These statements are non-synthesizable, and modules containing them will not be accelerated.

Accessing Memories Using Hierarchical References

Many testbenches access embedded memories in the design for the purposes of monitoring the simulation for expected results, for example:

```
always @(packet_complete)
 if (testbench.chip.sram.mem[0] == 1'b0)   // Using XMR to read memory
  $display("Invalid header bit");
```

If you want to access your memories in this fashion, there are two things to keep in mind:

1. Cross-module references (XMRs) to memories are not synthesizable; modules containing this type of XMR will not be accelerated by the emulator.
2. Memory fetches in a hardware-based component, such as the emulator, can be expensive, especially if done frequently during simulation.

© 2021 All Rights Reserved.

If you want to frequently access multiple addresses to the same memory, it is generally more efficient to upload the entire memory to a locally declared memory, perform the operation, then download the changed memory back to the -mapped memory. Use the `$readmemb` and `$readmemh` system tasks for this purpose.

For example, if your code is as follows:

```
reg [0:63] count;
always @(packet_complete)
begin
 for (i=0;i<64;i=i+1)
  begin
   testbench.chip.sram.mem[i] = count;   // Using XMR to write memory
   count = count + 1;
  end
end
```

You could change it to:

```
reg [0:31] local_mem [0:63];
reg [0:63] count;
always @(packet_complete)
begin
 $readmemh(testbench.chip.sram.mem, local_mem, 0, 63, 0, 63);
   // download 64 locations from -mapped memory to local memory
  for (i=0;i<64;i=i+1)
   begin
    local_mem[i] = count;
    count = count + 1;
   end
 $readmemh(local_mem, testbench.chip.sram.mem, 0, 63, 0, 63);
   // upload 64 locations from local memory to -mapped memory
end
```

Using Verilog PLI in the Testbench

Programs written using the Verilog Programming Language Interface (PLI) are run in software simulation. Since they are not accelerated, it is important to write them as efficiently as possible. Use the following guidelines when writing PLI calls.

Task/Function Routines Instead of Access Routines

Whenever possible, use task/function routines instead of access routines in your code. These simulate more efficiently in the simulator, as well as in other compiled-code simulators.

© 2021 All Rights Reserved.

Reading and Writing Signals in DUT Modules

Palladium Z1 lets you cross-reference nets and registers in DUT modules from your testbench for the purpose of using them in PLI calls, assuming the following conditions are met:

- DUT nets and registers can be read only; you cannot write to these objects.
- DUT nets and registers can be read at run time, only if they are *exported* at compile-time. Exporting a signal from the DUT allows you to read its value during simulation. Signals are automatically exported if:
  - You reference the DUT net or register in your testbench or other module running in software simulation mode, for example:

    ```
    always @(posedge testbench.chip.sys_clk)
                                // read reference to DUT signal
    ```

  - For Verilog, you explicitly export the signal using either the `$export_read` or `$export_frcrel` Verilog system tasks from a testbench module. Refer to *Command and System Tasks for SA Mode* chapter of the *VXE Command Reference Manual* for detailed information on these tasks.
  - For VHDL, you explicitly export the signal using either the `export_read` or `export_frcrel` VHDL procedures from a testbench module. Refer to *Command and System Tasks for SA Mode* chapter of the *VXE Command Reference Manual* for detailed information on these procedures.
- Do not use access routines in your PLI code that attempt to access signals that were not exported.
- If you use **tf** routines, and you attempt to pass a reference to a DUT signal as a parameter to the PLI call, you will get an error during compile-time by default, for example:

  ```
  initial
      $my_pli(testbench.chip.my_reg);
  --
  Error: XMR defined in DUT cannot be assigned in testbench
  ```

  To address this issue, you must enable read access to the PLI routine. This is done with the `+tfReadOnly+`*`pli_routine`* compiler option. To enable read access for several PLI routines, append their names to this option:

  `+tfReadOnly+`*`pli_routine1`*`+`*`pli_routine2`*`+...`

  If you do not specify any routine names, read access will be enabled for all routines in the design.

  Specify the `+tfReadOnly` option with the `ixcom` command.

© 2021 All Rights Reserved.

For a mixed-HDL Design:

`ixcom -top` *`top_level_design_unit`* *other_options* **+tfReadOnly+my_pli**

For a Verilog-only Design:

`ixcom` *Verilog_sources other_options* **+tfReadOnly+my_pli**

Dealing with Testbench Monitoring Code

The majority of typical testbench code involves the monitoring and manipulation of signals in the Device Under Test (DUT) to direct simulation or to detect error conditions occurring during simulation. This code is usually non-synthesizable and typically represents the largest performance bottleneck in simulation. There are several options to deal with this type of monitor code:

- Exclude the code completely from your simulation. This option provides the fastest performance, but sacrifices some debugging capabilities.
- Use the Testbench Callback technology to automatically execute the non-synthesizable code in the emulator. Refer to Executing Non-Synthesizable Code in the Emulator for more detail.
- Partition the monitor code into a synthesizable error-condition check and a non-synthesizable code segment that executes when the error is detected. This technique is described later in this section.
- Set the monitor code to be active in software simulation mode only. The design is swapped to software simulation time at specific time intervals, or after specific error conditions are detected. This technique is described later in this section.

This section contains the following:

- Activating Monitor Code Only in Software Simulation
- Monitoring Signals and Displaying Messages

Activating Monitor Code Only in Software Simulation

There are two techniques for making monitor code active in software simulation mode only:

- Marking Monitor Code to Simulate Only in Software Simulation
- Swapping to Software Simulation Mode at the Appropriate Time

© 2021 All Rights Reserved.

Marking Monitor Code to Simulate Only in Software Simulation

At run time, you can *mark* either a whole module in the DUT, or just a portion of it, to be active in software simulation mode only; in Simulation Acceleration mode the code is totally inactive and has no affect. After you swap to software simulation mode, the code becomes active again from the next simulation time-tick. This feature is especially useful for minimizing the impact of monitor code in testbenches. There are two ways to mark modules in this fashion:

- To *mark* an entire module to simulate while in software simulation mode only, specify the `+dutignore+`*module_name* compiler option. Note that all instances of the module *and its sub-hierarchy* will not simulate at all while running in Simulation Acceleration mode. To *mark* just a portion of a module to be simulated while in software simulation mode, use the `$axis_rccoff` system task and the `//axis rtl_off` pragma. The `$axis_rccoff` task takes no arguments, returning a boolean `true` if the design is currently running in software simulation mode or `false` if running in Simulation Acceleration mode.

  Here is an example:

  ```
  always @(posedge error_flag)
  begin
      if ($axis_rccoff)
     // code executed only if design running in software simulation mode
  end
  endmodule
  ```

  Note: This system task is non-synthesizable and thus cannot be used in a DUT module.

Swapping to Software Simulation Mode at the Appropriate Time

To simulate the monitor code, you must initiate the swap from Simulation Acceleration mode to software simulation mode, in either of two places:

- xeDebug *CLI (Command-Line Interface)*: You can control the swapping feature at the command line, using the commands listed in the *Hot Swap (SW to HW) Options* sub-section of the *SA Run-Time and Debug Commands* chapter of the *VXE Commands Reference Manual*. If you want to break on a signal in Simulation Acceleration mode, that signal must be exported at compile time, as previously explained.

- *Verilog testbench*: You can use the `$xc_cmd` system task to execute an `xc` command directly from your testbench, for example:

  ```
  module testbench ();
    ...
    always @(posedge sys_clk)
    begin
      if (packet_status==3'b0 && !packet_received)
      $xc_cmd("verbose 2");     // enable xc verbose messages
  ```


© 2021 All Rights Reserved.

```
  end
endmodule
```

For more information on `$xc_cmd`, refer to the *System Tasks and Procedures* section of the *Commands and System Tasks for SA Mode* chapter in *VXE Command Reference Manual*.

Note: The following commands are not supported through the `$xc_cmd` system task if you are using the xeDebug tool:
`on`, `off`, `stop`, `run`, `tbrun`, `autorun`, `nbrun`, `match`, `bmatch`

Monitoring Signals and Displaying Messages

The `$display` and `$monitor` system tasks, which can be used in the testbench to display message strings and signal values during simulation, are fully supported in the software simulator and produce expected results if used in non-accelerated modules.

However, if used in a DUT module, by default, these tasks will not work as expected while running the design in Simulation Acceleration modes, because they are not synthesizable and are thus ignored by the emulator. If you are trying to add a testbench module to the DUT so it can be accelerated, this can be a problem, for example:

```
module dut (..);
always @(posedge sys_clk)
begin
    if (start_mem_cycle && write_en && read_en)
    $display("Invalid memory cycle at time %t", $time);
end
```

In this example, the `$display` system task is ignored if running the design in Simulation Acceleration mode.

Any of the preceding techniques for speeding up monitor code can also be used to display these messages. If using the partitioning technique, you would modify the previous example, as follows:

```
module dut (..);
reg dummy_reg;
always @(posedge sys_clk)
begin
    if (start_mem_cycle && write_en && read_en)
        dummy_reg = 1'b1; //register toggles when error condition detected
    else
        dummy_reg = 1'b0;
end
```

Now add the following code to a testbench module:

```
always @(posedge testbench.dut.dummy_reg)
    $display("Invalid memory cycle at time %t", $time);
```


© 2021 All Rights Reserved.

In the previous example, only the `$display` message itself is simulated in software simulation; actual error checking is accelerated by the emulator.

Exercising Caution While Forcing Signals

The results of the `force` statement are not completely identical to Xcelium and VCS, and might also vary for ICE and SA modes in Palladium. The `force` statement in Xcelium depends on the type of driver. For example, the type of driver is `wire` as shown in the following code below and you issue `force U1.in1 = 1'b0`, then `U1.in1` and `U2.in2` will be forced to `1'b0` in Xcelium and VCS.

```
wire u0_out;
wire b;
assign b=u0_out;
A U0(
.in1(),
.out1(u0_out));
A U1(
.in1(b)
.out());

A U2(
.in1(b)
.out());
```

If you change the type of signal `b` to `reg`, then only `U1.in1` is forced in Xcelium and VCS.

```
reg b;
always @(posedge clk)
b<=u0_out;
```

However, in Palladium, all nets are forced in the SA mode and only nets in the `force` statement are forced in ICE mode irrespective of the type of the driver.

The following table lists the differences in the results of the `force` statement:

Table 24-1 Results of the Force Statement

	Type `reg`	**Type** `wire`
All nets are forced	SA	SA, Xcelium, and VCS
Only the nets in the `force` statement are forced	ICE, Xcelium, and VCE	ICE

If the `force` statement described above does not meet your requirements, consider using the `hierAssign` command described in the *User Data Commands* chapter of the *VXE Command Reference Manual*.

© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

25

Compiling and Running Simulation Acceleration Designs with *xrun*

The *xrun* utility enables compilation and simulation of designs using the Xcelium simulator by specifying all the input files and options on a single command line. *xrun* simplifies the invocation process by invoking multiple tools using a single command.

The traditional use models for simulation acceleration (SA) mode using Palladium Z1 require a mix of different tools to compile and simulate a design. The *xrun* use model for simulation has been designed to invoke the IXCOM tools (vlan, vhan, and ixcom) to compile a design for simulation acceleration, and simulate the hardware-acceleratable snapshot. Thus, *xrun* facilitates a smooth transition from the pure Xcelium-based software simulation mode to SA mode using Palladium Z1.

This chapter covers the following topics:

- Overview
- SA-Specific Options of the xrun Command
- Compiling Designs
- Simulating Designs
- Limitations

Important

The content of this chapter pertains only to SA mode.

Overview

The *xrun* interface for compiling and running designs in SA mode is similar to the *xrun* interface used in Xcelium-based software simulation mode.

© 2021 All Rights Reserved.

For SA mode, in addition to using the typical software-simulation options of the `xrun` command, you need to use the following command-line options with the `xrun` command:

- Command options for enabling SA mode
- Compilation options for SA mode that need to be passed to IXCOM tools
- Simulation options that need to be passed to Xcelium tools or xeDebug

xrun internally invokes the Xcelium and IXCOM tools to compile the source files, generate the snapshot for SA mode, and invoke the simulator. *xrun* dispatches the specified command-line options to the appropriate Xcelium and IXCOM tools. At run time, the Xcelium simulator, *xmsim*, is invoked as the default simulation tool. Alternatively, invoke xeDebug as the run-time and debug environment by using the `-xedebug` option with the `xrun` command. For more information about the new command options, see SA-Specific Options of the xrun Command on page 1187.

Setting Up the Environment to Use *xrun* for Simulation Acceleration

You must set the required environment variables to use *xrun*. See the *Configuring Your Environment for the 64-Bit Version* section in the *Setting Up Your Environment* document in the Xcelium documentation set for more information.

Additionally, set the required environment variables for IXCOM before you compile and run the design in SA mode. See Setting Up the Environment for Palladium Z1 chapter for more information.

Compiling and Simulating Designs

Use *xrun* to compile and simulate designs for simulation acceleration either in a single step or multiple steps.

Single-Step Compilation and Simulation

In the single-step compilation and simulation process, use *xrun* to:

- Compile and elaborate using IXCOM and Xcelium tools, and simulate using *xrun*:

  ```
  xrun -hw [ixcom_options] [ies_options] <source_files>
  ```

- Compile and elaborate using IXCOM and Xcelium tools, and simulate using xeDebug:

  ```
  xrun -hw -xedebug [ixcom_options] [ies_options] <source_files>
  ```


© 2021 All Rights Reserved.

Multi-Step Compilation and Simulation

In the multi-step compilation and simulation process, use *xrun* to:

1. Compile and elaborate the design in multiple steps:

   ```
   1 xrun -hw -compile [ixcom_options] [ies_options] <source_files>
   ```

   Note: This step can be invoked multiple times to compile different sets of input source files.

   ```
   2 xrun -hw -elaborate [ixcom_options] [ies_options]
   ```

2. Simulate and debug the design:

   - Simulate the design using *xmsim*:

     ```
     xrun -R
     ```

   - Simulate and debug the design using xeDebug:

     ```
     xrun -R -xedebug
     ```

The `-hw` option enables compilation of the design for SA mode. *xrun* internally invokes the IXCOM tools (`vlan`, `vhan`, and `ixcom`) to compile the design files with the IXCOM options specified on the command line. The IXCOM tools work according to their existing specifications. By default, xmsim is invoked for simulation. If the `-xedebug` option is specified on the `xrun` command line, the xeDebug tool is invoked for simulation. The run-time environment using xeDebug is discussed in Simulating Designs on page 1202.

Important

xrun has been enhanced for SA mode to facilitate the transition from pure Xcelium-based simulation to hardware-accelerated simulation using Palladium Z1. You can still directly use the IXCOM tools (vlan, vhan, and ixcom) for design compilation. Unlike the Xcelium tools, IXCOM tools do not support incremental compilation. When using *xrun* for SA mode, incremental compilation is supported only for the portion of the design compiled using Xcelium tools.

Note: Using `xrun -hw` command to compile the design for SA mode so that it can be run on the Palladium Z1 emulator is a Beta-level feature. Beta-level features are those that have been introduced in one of the recent releases but have not yet reached the production stage. The stability and usability of these features is not well established. Beta-level features are released to you to obtain your feedback before the features reach the production stage.

SA-Specific Options of the xrun Command

To use *xrun* for compiling and running the design for SA mode, specify the following categories of options with the `xrun` command:

© 2021 All Rights Reserved.

- Compilation Options to Enable SA Mode
- Run-Time Options for Using xeDebug

Compilation Options to Enable SA Mode

This section lists the options of the `xrun` command to compile the design files for creating a hardware-acceleratable simulation snapshot.

Table 25-1 Compilation Options for Simulation Acceleration

Compilation Option	Description
`-hw`	**Note:** Specifies that the IXCOM tools should be used to compile and process the design for simulation acceleration. All the `ixcom` command options specified with the `xrun` command, without specifying the `-hw` option are ignored.
`-compile_sw <filename1> <filename2> ... -end`	Specifies the files that should be compiled using the Xcelium tools, irrespective of the HDL used to write these files. Use this option for using *xrun* to compile code written in Hardware Verification Languages (HVLs) that cannot be mapped to the emulator.
`-ixcomargs <string>`	Directly dispatches options indicated by `<string>` to IXCOM (without any parsing by *xrun*). The `-<ixcom_tool>args <string>` options are provided to make the use of *xrun* for SA mode compatible with the use of Xcelium tools in the pure simulation invocation using *xrun*.
`-vlanargs <string>`	Directly dispatches options indicated by `<string>` to the `vlan` tool (without any parsing by *xrun*).
`-vhanargs <string>`	Directly dispatches options indicated by `<string>` to the `vhan` tool (without any parsing by *xrun*).
`-log_ixcom <logfile>`	Displays output from the `ixcom` command into the specified `logfile`. The `-log_<ixcom_tool> <logfile>` options are provided to make the use of *xrun* for SA mode compatible with the use of Xcelium tools in the pure simulation invocation using *xrun*.

© 2021 All Rights Reserved.

`-log_vhan <logfile>`	Displays output from the `vhan` command into the specified *`logfile`*.
`-log_vlan <logfile>`	Displays output from the `vlan` command into the specified *`logfile`*.
IXCOM tools command options	All the options for IXCOM tools (`vlan`, `vhan`, and `ixcom`) are recognized by the `xrun` command and passed to the appropriate tools. Refer to the *Commands and System Tasks for SA Mode* chapter in the *VXE Command Reference Manual* for a list of all these command options.

Run-Time Options for Using xeDebug

This section lists the options of the `xrun` command used for running and debugging the simulation by using the xeDebug tool. By default, *xrun* is used at run time in the absence of these options.

Table 25-2 Options to Use xeDebug at Run Time

Option	Description
`-xedebug`	Specifies that xeDebug should be used at run time.
`-host <name>`	Specifies the host name for the emulator. By default, the current machine is identified as the host machine.
`-xedebugargs “ [options] “`	Specifies the arguments to be passed on to xeDebug at run time. For example, the following command enables you to use dynamic probes when running xeDebug from *xrun*. This option is particularly useful because when you invoke xeDebug from *xrun*, you do not have any control over the emulator. `xrun -R -xedebug -xedebugargs "-dynp"`
`-disxedebug`	Specifies that *xrun* should be used at run time. This is the default option.

© 2021 All Rights Reserved.

Accessing Help Information for IXCOM Tools from xrun

The help infrastructure of *xrun* provides usage information for the options for IXCOM tools.

- To display help for options specific to a tool, use the following command syntax:

  `xrun -helpsubject <topic>`

  For example:

  - `xrun -helpsubject vlan` - Displays a list of `vlan` command options and their descriptions.
  - `xrun -helpsubject vhan` - Displays a list of `vhan` command options and their descriptions.
  - `xrun -helpsubject ixcom` - Displays a list of `ixcom` command options and their descriptions.
  - `xrun -helpsubject hwaccel` - Displays a list of all the supported command options for the IXCOM tools (***vlan***, ***vhan***, and ***ixcom***) and their usage descriptions.

- To display help for a specific option of an IXCOM tool, use the following command syntax:

  `xrun -helpargs <option>`

  - Executing the `xrun -helpargs <ixcom_option>` command displays the description for the specified option of the `ixcom` command.
  - Executing the `xrun -helpargs -ua` command displays the following output:

    ```
    xrun(64): 10.20-s040: (c) Copyright 1995-2011 Cadence Design Systems, Inc.

    -ua                        Compile design to UA hardware engine
    ```

xrun provides additional help options for getting verbose descriptions for command options and retrieving a list of tools to which the specified option belongs. For more information, see the *xrun User Guide* in the Xcelium documentation set.

Compiling Designs

xrun invokes the IXCOM and Xcelium tools to compile input source files when the `-hw` option is specified on the command line. IXCOM and Xcelium tool invocation mechanism is described below:

- Verilog (Verilog-1995, Verilog-2001, SystemVerilog) and VHDL (87 and 93) source files are compiled using the **IXCOM** tools (vlan, vhan, and ixcom). The Xcelium tools support

© 2021 All Rights Reserved.

all these languages supported by the IXCOM tools as well. But, priority is given to using IXCOM tools for these languages.

- All the source files written in languages other than the languages specified above, such as SystemC, Specman ***e***, C and C++ (for PLI, VPI, VHPI, and DPI) are compiled using Xcelium tools.

Exception to the above points: Source files specified within the `-compile_sw` and `-end` options are always compiled using Xcelium tools irrespective of the HDL used to write these files.

xrun dispatches the command-line options to the tools it invokes. For more information, see Parsing IXCOM and xrun Command Options with the xrun Tool on page 1196.

The following sub-sections describe how *xrun* supports designs using different languages, in simulation acceleration mode.

- Verilog-Only Designs
- Mixed-Language (Verilog and VHDL) Designs
- Designs with PLI/VPI and VHPI
- Designs with SystemVerilog DPI
- Designs with SystemC
- Designs Using Specman e Testbenches

Note: For the use model examples in this chapter, only the required options are listed with the `xrun` command. You can include additional *xrun* and/or IXCOM options for each of the following use models.

Other compilation aspects are described in the following sub-sections:

- Parsing IXCOM and xrun Command Options with the xrun Tool
- User-Partitioned Testbench and DUT Compilation
- Logical Library Support

Verilog-Only Designs

Verilog-only design files are compiled using the `xrun` command as shown below:

```
xrun -hw [ixcom_options] [ies_options] <source_files>
```


© 2021 All Rights Reserved.

If the `-top` option is not specified on the `xrun` command line, single-step compilation using `ixcom` is invoked. IXCOM automatically determines the top-level design unit names. For example, the following command invokes `ixcom` to analyze the source files and elaborate the design in a single step:

```
xrun -hw -ua +dut+Dut Tb.sv Dut.v
```

If you specify the top-level design unit names using the `-top` option on the `xrun` command line, multi-step IXCOM compilation using `vlan` and `ixcom` is invoked. For example, the following command invokes `vlan` to analyze the source files and `ixcom` to elaborate the design.

```
xrun -hw -ua +dut+Dut Tb.sv Dut.v -top Tb
```

Using the <tool> args Options

In single-step invocation using the `ixcom` command, *xrun* ignores the `-vlanargs` option specified on the command line. `-ixcomargs` must also have context-sensitive (single-step or multi-step) options for correct compilation of the design.

Mixed-Language (Verilog and VHDL) Designs

Mixed-language (Verilog and VHDL) design files are compiled using `xrun` command as shown below:

```
xrun -hw [ixcom_options] [ies_options] <verilog_and_vhdl_source_files>
```

For mixed-language designs, *xrun* invokes IXCOM tools to compile the design in multiple steps. First, design files are analyzed using language specific analyzers, `vlan` for Verilog files and `vhan` for VHDL files. In the final step, analyzed design units are elaborated using `ixcom` to generate the simulation snapshot. For example, the following command invokes `vlan` and `vhan` to analyze source files and `ixcom` to elaborate the design.

```
xrun -hw -ua +dut+Dut Tb.sv Dut.vhd -top Tb
```

Important

To compile mixed-language designs that include VHDL source files, you must specify the top-level design unit names using the `-top` option. This requirement is applicable even for designs in which a Verilog top-level design unit instantiates a VHDL design unit.

© 2021 All Rights Reserved.

Designs with PLI/VPI and VHPI

PLI/VPI and VHPI applications and design files are compiled using the xrun command, as shown below:

```
xrun -hw [ixcom_options] [ies_options] \
     <design_files_&_PLI_VPI_VHPI_application_source_files_library>
```

For example:

```
xrun -hw -ua +dut+Dut +sv test.v test.c
```

The PLI/VPI/VHPI application files are compiled using `ncsc`. *xrun* invokes IXCOM tools to compile the design files using either a single step or multiple steps, as described in Verilog-Only Designs and Mixed-Language (Verilog and VHDL) Designs. The design is elaborated using `ixcom` to generate the simulation snapshot with the application code library.

Only the `-afile`, `-loadpli1`, and `-loadvpi` options of ***xrun*** are mapped to the corresponding options of the ixcom command.

- Use the `-afile` option to specify the PLI map file. The `-p|P` options of `ixcom` can also be used, depending on the map file format.

  ```
  xrun -hw -ua +dut+Dut +sv -afile PLI_map_file test.v test.c
  ```

  Note: If the PLI map file is not specified, *xrun* generates this mapping file using the `ncpli2tab` utility. Support for the `-afile` option is limited to PLI/VPI specific content in the access file. Other access specifications in the file result in IXCOM tool error.

- The support for PLI/VPI and VHPI is similar to the way *xrun* reads the PLI/VPI-specific content in the file specified using the `-afile` option. Refer to Xcelium documentation on "*Using a PLI map file*" for details on PLI/VPI-specific-content-allowed access file. The `-afile` option is mapped to the `ixcom` command option. IXCOM reports an error for non-PLI/PVI specific content in the access file. Following is an example of an xrun command using the `loadpli1` option:

  ```
  xrun -hw -ua +dut+Dut +sv Tb.sv Dut.sv +loadpli1=".<path>/<library_name>.so"
  ```

- Use the `+loadvpi` option for VPI applications.

VHPI support is handled through attributes in the source language, and does not require any explicit command-line option to load the application library.

Designs with SystemVerilog DPI

The SystemVerilog DPI application code and design files are compiled using the xrun command, as shown below:

© 2021 All Rights Reserved.

```
xrun -hw [ixcom_options] [ies_options]
     <design_files_&_DPI_application_source_files_library>
```

SystemVerilog DPI application source files are compiled using `ncsc`. *xrun* invokes IXCOM tools to compile the design files using single or multiple steps, as described in Verilog-Only Designs and Mixed-Language (Verilog and VHDL) Designs. The design is elaborated using `ixcom` to generate the simulation snapshot with the application code library.

For example, the following commands are used for multi-step compilation with pre-compiled application DPI application library:

```
xrun -hw -ua +dut+Dut Tb.sv Dut.sv Dpi.cpp
xrun -hw -ua +dut+Dut Tb.sv Dut.sv -sv_lib ./<path>/<library_name>.so
```

xrun provides the `-cpost` and `-end` options to specify the order of compilation of DPI C/C++ libraries. *xrun* orders internal invocations of IXCOM tools to correctly handle the functional requirements of these options, as it is handled in pure software simulation mode. See the *xrun User Guide* in the Xcelium documentation set for more information about these options and their functional requirements, and other DPI-specific-elaboration options.

Designs with SystemC

SystemC application code and design files are compiled using `xrun` command, as shown below:

```
xrun -hw [ixcom_options] [ies_options] \
         <design_files_and_SystemC_application_source_files_or_library>
```

SystemC application source files are compiled using `ncsc`.The *xrun* tool invokes the IXCOM tools to compile the design files using single or multiple steps, as described in Verilog-Only Designs and Mixed-Language (Verilog and VHDL) Designs. The design is elaborated by using the `ixcom` command to generate the simulation snapshot with the application code library.

Important

To compile SystemC designs, you must specify the top-level design unit names using the `-top` option. Otherwise, an error is reported.

For example:

- One-step compilation and simulation using the shell interface:

  ```
  xrun -hw -sysc Tb.v Dut.sv Dut_shell.sv Sysc.cpp
  ```

- One-step compilation and simulation using the auto-generated shell file:

  ```
  xrun -hw Tb.v Dut.sv -loadsc ./<path>/<library_name>.so
  ```


© 2021 All Rights Reserved.

Automatic generation of SystemC shell file support is enabled for Verilog and VHDL with the `-scautoshell` option. An unsupported error message is reported for "`-scautoshell SystemC`" invocation with "`-hw`" options.

The *xrun* tool passes the SystemC library (created by `ncsc_run`) to IXCOM elaboration invocation with the `-loadsc` option.

Designs Using Specman *e* Testbenches

Specman ***e*** testbench and design files are compiled using `xrun` command, as shown below:

```
xrun -hw [ixcom_options] [ies_options] <design_files_&_testbench_e_files>
```

For example:

```
xrun -hw -ua +dut+Dut Dut.sv Tb.e
```

Keep the following considerations in mind when using *xrun* to compile Specman ***e*** designs for hardware using the `-hw` option of the `xrun` command:

- The Specman reference files for VHDL are not compiled automatically (unlike the pure-simulation flow). You must specify the reference files explicitly in the `xrun` command line to enable VHDL design files to be compiled using the IXCOM tools.
- The Specman package files for SystemVerilog are not compiled automatically (unlike the pure-simulation flow). You must specify the package files explicitly in the `xrun` command line to import package objects into SystemVerilog modules compiled using IXCOM tools. IXCOM tool limitations apply for the package file.
- Combination of Specman ***e***, SystemC, and Verilog/VHDL compilation is not supported.
- The timescale information must be specified for Xcelium elaboration.

  Note: This limitation exists for the IXCOM-based compilation flow as well.

Important

Use the `-sntimescale <arg>` option on the `xrun` command line to specify the timescale for Specman delays. In a pure software-simulation run, even if the timescale for Specman is not specified using the `-sntimescale` option, for a design with a consistent timescale definition, the timescale from *xrun* is automatically used for Specman, along with a warning. When using *xrun* for simulation acceleration, you need to explicitly use the `-sntimescale <arg>` option to specify the timescale.

© 2021 All Rights Reserved.

Parsing IXCOM and xrun Command Options with the *xrun* Tool

xrun passes the options specified with the `xrun` command to the relevant IXCOM and Xcelium tools. The functionalality of the option is defined by the tool invoked by *xrun*. Thus, all the options of the `ixcom` command work in the same manner when using *xrun* to compile and simulate designs in the simulation acceleration mode.

If you specify an Xcelium-tool option on the `xrun` command line that has the same syntax and functionally equivalent behavior as an existing `ixcom` option, the Xcelium-tool option is mapped to the identical `ixcom` option. See Table 26-1 on page 1207 in the *Xcelium - IXCOM Option Mapping* Appendix for a list of such options.

For example, both the `xrun` and `ixcom` commands have the `-clean` option to remove the compiler-generated database files.

- If the `-hw` option is included, as in the following `xrun` command, the database files generated by both the Xcelium and IXCOM tools are cleaned up:

  ```
  xrun -hw -clean
  ```

- If the `-hw` option is not included, only the Xcelium database files, `INCA_libs`, are cleaned up.

If you specify an Xcelium tool option on the `xrun` command line, for which the syntax is different but the functional behavior is equivalent to an option of the `ixcom` command, the Xcelium option is mapped to the corresponding `ixcom` command option. See Table 26-1 on page 1207 in the *Xcelium - IXCOM Option Mapping* Appendix for a list of such options.

You can use the `-debugscript` <*`filename`*> option of the `xrun` command to generate a script for capturing the environment variables and commands related to the executables invoked by xrun. This script helps to identify how each of the Xcelium and IXCOM tools was invoked by *xrun* and debug the problems encountered during the execution of the `xrun` command.

Using the `-debugscript` option with the `xrun` command, generates two files:

- A script with the specified <*`filename`*> that contains the commands you can use to run the executables invoked by *xrun*. Each executable is invoked with the `-f` <*`argument_file`*> option, which specifies the argument file created by *xrun* when that executable was invoked. The argument file contains the command-line arguments passed to the executable.
- A file called <*`filename`*>.`env` that you can source to reproduce the same environment as the *xrun* run.

More information about the `-debugscript` <*`filename`*> option is available in the *xrun User Guide* in the *xrun-Specific Command-line Options* section.

© 2021 All Rights Reserved.

Handling Unsupported Options Ignored by xrun

As stated above, when *xrun* is invoked with the `-hw` option to compile designs for SA mode, all the `ixcom` command options are supported on the `xrun` command line. However, all Xcelium options cannot be used in the same manner. Only some Xcelium options are mapped to IXCOM options, and are passed to the appropriate IXCOM tools at the time of invocation.

The rest of the Xcelium options that are not mapped to any of the IXCOM options are ignored. However, you need to be careful about using such options because this might result in incorrect functionality when compared to pure-Xcelium simulation. To determine the ignored or unmapped Xcelium options, use the `-checkargs` option of the `xrun` command. When this option is used, the list of ignored options is reported as per the invocation of the appropriate IXCOM tool as follows:

- All ignored options of the `xmvlog` tool (Xcelium) are reported at the time of invocation of `vlan` tool (IXCOM).
- All ignored options of the `xmvhdl` tool (Xcelium) are reported at the time of invocation of `vhan` tool (IXCOM).
- All ignored options of the `xmvlog` and `xmelab` tools (Xcelium) are reported at the time of single-step invocation of `ixcom` tool (IXCOM) for compiling and elaborating Verilog files.
- All ignored options of the `xmelab` tool (Xcelium) are reported at the time of multi-step invocation of `ixcom` tool (IXCOM).

To see the list of all the options that are either ignored or processed, use the following command:

```
xrun -hw -checkargs
```

Xcelium-Compatible Support for Testbench Compilation

In SA mode, the verification environment, typically, includes a non-synthesizable testbench and synthesizable design units. The *xrun* utility invokes the Xcelium tools to compile testbench source languages, which are not supported by the IXCOM tools (HVLs such as SystemC, ***e*** and Vera). The SystemVerilog and VHDL portions can also contain non-synthesizable constructs intended for testbench. The `xrun` command options `-compile_sw` and `-end` allow you to manually partition the tesbench portion of the design.

Use the `-compile_sw` and `-end region` option to enable a pure-Xcelium-like compilation for the testbench. However, this compilation is not exactly like the Xcelium compilation, and

© 2021 All Rights Reserved.

has the limitation that you must specify the `-top` <*du*> option to compile the testbench portion of the design.

Refer to User-Partitioned Testbench and DUT Compilation on page 1198 and Designs Using TBA and UVCs on page 1199 for details about supported testbench compilation use models.

User-Partitioned Testbench and DUT Compilation

You can specify explicit partitioning information to improve compilation efficiency. You can specify explicit Xcelium compilation for testbench files (non-synthesizable HVLs and unsupported constructs in IXCOM). Use the `-compile_sw` and `-end` options with the `xrun` command to specify these files. *xrun* invokes Xcelium tools to process the testbench files specified within the `-compile_sw` and `-end` command options (the testbench marker region). The design files are compiled using IXCOM tools. The final design is elaborated using Xcelium with the IXCOM-compiled design units. The following `xrun` command is used.

```
xrun -hw -ua +dut+Dut +xcDesignTop+Tb.dut=Dut [ixcom_options] [ies_options]
     <design_source_files> -compile_sw <testbench_source_files> -end
```

Specify the top-level design units explicitly within the `-compile_sw` and `-end command-line` regions. The top-level design units specified using the `-top` <*du*> option within the `-compile_sw` and `-end` regions are compiled by Xcelium. To specify top-level design units for IXCOM compilation, the `-top` <*du*> option must be specified outside of the `-compile_sw` and `-end` regions.

Example 1: Xcelium testbench compilation and elaboration of pre-compiled design units using IXCOM (one-step compilation)

```
xrun -hw -ua +dut+Dut +xcDesignTop+Tb.dut=Dut \
             Dut.sv Chk.vhd -compile_sw Tb.v -end
```

Example 2: Xcelium testbench compilation and elaboration of pre-compiled design units using IXCOM (multi-step compilation)

```
xrun -hw -compile dut.sv
xrun -hw -elaborate -top Dut +xcDesignTop=Tb.dut=Dut ...
xrun -hw -compile_sw tb.sv -top Tb -end ...
```

Limitations

Xcelium limitations apply for options specified within the `-compile_sw` and `-end` option marker region on the command line. For more information, see *xrun User Guide* in the Xcelium documentation set.

© 2021 All Rights Reserved.

Designs Using TBA and UVCs

The *xrun* support for these features is similar to use model described in User-Partitioned Testbench and DUT Compilation. Automation support to include header file and linking use-model-specific libraries is not available. Synthesizable design units are compiled using IXCOM tools and non-synthesizable constructs are compiled using Xcelium tools. The `-compile_sw` and `-end` options in the command line are used to manually partition testbench and design source compilation. The following examples illustrate the compilation of TBA designs with SystemVerilog testbenches.

Example 1: Xcelium testbench compilation and elaboration of pre-compiled design units using IXCOM (one-step compilation):

```
xrun -hw dut.sv -ncompile ... -compile_sw tb.sv -top Tb ... -end
```

Example 2: Xcelium testbench compilation and elaboration of pre-compiled design units using IXCOM (multi-step compilation):

```
xrun -hw -compile dut.sv
xrun -hw -elaborate -xmcompile -top Dut  ...
xrun -hw -compile_sw tb.sv -top Tb -end ...
```

Note: You can compile non-sythesizable testbenches written in HVLs such as SystemC, ***e***, and Vera by using the same use model.

Refer to Xcelium-Compatible Support for Testbench Compilation on page 1197 for details.

Logical Library Support

With the `xrun` command, you can choose to analyze the design units into multiple logical libraries by using the `-makelib ... -endlib` option set as follows:

```
xrun -hw [ixcom_options] [ies_options] \
     -makelib <path_to_library>[:<logical_name>] <source_files> -endlib |\
     -end ...
```

Multiple logical libraries are supported in the same manner as pure-Xcelium. Use the `-makeLib` and `-endLib` options to specify the libraries.

Refer to Handling cds.lib and hdl.var Files on page 1200 for details about logical library specification.

© 2021 All Rights Reserved.

Handling cds.lib and hdl.var Files

You can use the `-work` option of the xrun command to analyze a given set of files into a single logical library, as follows:

```
xrun -hw -compile -work <library> [ixcom_options] <source_files>
```

For Xcelium tools, the library mapping information specified with the above commands is recorded in the `cds.lib` file. The IXCOM tools, on the other hand, use the `libmaprc` file for this library-mapping information. The syntax of the `libmaprc` file used by IXCOM tools is similar, but not the same as the `cds.lib` file used by the Xcelium tools. When you use *xrun* for SA mode, this `libmaprc` file is automatically generated, based on the following:

- Information in the `cds.lib` file
- Arguments specified with the `-makeLib` option of the xrun command

Note: The content in the `hdl.var` file is not considered for the generation of the `libmaprc` file. Also, any other specification in the `cds.lib` file other than the logical-library mapping using the `DEFINE` statement is ignored.

With **Xcelium**, the `cds.lib` file has entries like the following:

```
define L1 ./Lib1
```

Here, `L1` is the logical name and `./Lib1` is the physical directory. It is essential to specify the logical name with the `xrun` command as follows:

```
xrun -work L1
```

With **IXCOM**, only the `libmaprc` file is used, and its syntax is as follows:

```
L1 => ./Lib1
```

Note: The *xrun* tool creates physical directories mapped to the specified logical libraries inside the `INCA_libs` directory. The IXCOM tools create the physical directories in a directory named `ixcom` within the `INCA_libs` directory. When you use the `-clean` option with the `xrun` command to remove the Xcelium-generated directories, the IXCOM-generated directories are also removed.

To handle such automatic generation of the `libmaprc` file, and/or to force the IXCOM-style-library-mapping method of using the `libmaprc` file, you can disable the automatic conversion of `cds.lib` file by using the `-libmaprc <file>` option with the `xrun` command.

For more information about the `libmaprc` files, see Associating Logical Libraries to Physical Directories Using libmaprc File on page 617. For details on handling `cds.lib` file, see Handling cds.lib and hdl.var Files.

© 2021 All Rights Reserved.

The following table presents the search order for `cds.lib` and `libmaprc` files and the `-cdslib` and `-libmaprc` options.

Table 25-3 Search Order for cds.lib and libmaprc Files and Options

-cdslib Option	-libmaprc Option	cds.lib File	libmaprc File	used	Comment
Present	Present	Present	Present	`-libmaprc`	If both the `-cdslib` and `-libmaprc` options are present and the `cds.lib` and `libmaprc` files are present, the `-libmaprc` option takes precedence.
Present	-	Present	Present	`-cdslib`	The `-cdslib` option takes precedence over the `cds.lib` file in the current directory.
-	-	Present	`-libmaprc`	`cds.lib`	The `cds.lib` file takes precedence, and it is searched in the order as specified above.
-	-	-	-	-	The `libmaprc` file is never used; instead the `cds.lib` file, which is internally generated by *xrun*, is used.

Analyzing into Multiple Logical Libraries Using xrun in Single- and Multi-Step Modes

xrun supports design compilation into multiple logical libraries with both single-step and multi-step compilation processes.

© 2021 All Rights Reserved.

Single-Step Compilation Process

Following are the examples of single-step compilation:

- One step compilation and simulation (with compilation into a logical library)

  ```
  xrun -hw -makelib Lib1 Dut.v -endlib Tb.sv
  ```

- One step compilation and simulation (with compilation into a logical library with explicit logical name)

  ```
  xrun -hw -makelib ./Chip/Dut:Lib1 Dut.v -endlib Tb.sv
  ```

Multi-Step Compilation Process

In the multi-step process, specify the logical library name using the `-reflib` option in the elaboration step. This is consistent with Xcelium and IXCOM compilation use models for elaborating design units compiled into different logical libraries. Following is an example of compilation and elaboration using multiple steps:

```
xrun -hw -compile -makelib ./Chip/Dut:Lib1 Dut.v -endlib
xrun -hw -compile -makelib ./Chip/Sub:Lib2 Sub.v -endlib
xrun -hw -elaborate Tb.sv -reflib Lib1 -reflib Lib2 -top Tb
```

Cadence recommends that you use the `-makelib` and `-reflib` options on the `xrun` command line to compile and elaborate the design.

Note: Xcelium limitations apply for options specified within the `-makelib` and `-endlib` options on the *xrun* command line. For Xcelium limitations and/or unsupported options when using this use model, use the multi-step compilation process as described above.

Simulating Designs

This section provides information on simulating a hardware-acceleratable design using *xrun*. A prerequisite for simulating such a design is compilation using the `-hw` option with the `xrun` command.

Simulating Using xmsim

By default, *xrun* invokes xmsim to simulate the IXCOM-generated snapshot. This behavior is similar to pure Xcelium-based simulation. When using *xrun* for simulation acceleration, use the following steps to invoke xmsim as the run-time and debug environment:

1. Compile the design using `-hw` option with the `xrun` command:

   ```
   xrun -hw [ixcom_options] [ies_options] <source_files>
   ```


© 2021 All Rights Reserved.

Alternatively, use two-step compilation, as described in Multi-Step Compilation and Simulation on page 1187.

2. Simulate the design:

```
xrun -R [-gui]
```

For more information, see Invoking ***xrun*** Tool (for SA Mode) on page 77.

Simulating Using xeDebug

xeDebug provides a unified access to testbench in software and design mapped to the hardware. A unified interface eases transitioning from software simulation to the simulation acceleration mode. Minor modifications are required to the native simulation scripts, mostly to handle swap in and swap out of the design to and from emulator.

To ensure *xrun* invokes xeDebug as the run-time and debug environment, specify the `-xeDebug` option with the `xrun` run-time command. For example:

1. To compile the design, use the following command:

```
xrun -hw [ixcom_options] [ies_options] <source_files>
```

2. To simulate the design, use the following command:

```
xrun -R -xeDebug [-gui]
```

xeDebug is invoked as the run time. When the `-gui` option is specified, two sets of windows are displayed, one for SimVision and the other for xeDebug.

Note: You can still use the standard method of invoking the xeDebug tool for running and debugging a simulation snapshot. For more information, see Invoking xeDebug Tool on page 76. For general information about xeDebug, see Design Debugging Processes on page 1697. Also see Limitations.

Compilation and Run-Time Logs

xrun collects standard output from the IXCOM tools and xeDebug to the default `xrun.log` file. Alternatively, you can specify a log file of your own using the `-l <logfile>` option.

You can redirect the output from `vlan`, `vhan`, and `ixcom` to a separate log file using the `-log_<tool_name> <file_name>` option.

To append the output to an existing log file, use the existing *xrun* option `-append_log`.

© 2021 All Rights Reserved.

Limitations

- Incremental compilation support is limited to the Xcelium portion of the *xrun* compilation. Thus, *xrun* does not support incremental compilation of design files (or database) compiled using `ixcom`.
- IXCOM supports only `-top <design_unit_name>` and `-top <library>.<design_unit_name>` for the `-top` option. Xcelium tools support more sub-options. See Xcelium documentation set for more information.
- xeDebug user sessions are not supported. In the offline debugging mode, you need to start xeDebug directly.
- The Universal Verification Methodology (UVM) use model is not supported.

© 2021 All Rights Reserved.

26

xrun - IXCOM Option Mapping

When *xrun* is used for simulation acceleration, it internally invokes Xcelium and IXCOM tools to compile the source files, generate the snapshot for simulation acceleration, and invoke the simulator. *xrun* dispatches the specified command-line options to Xcelium and IXCOM tools. If you specify `xrun` command options that have functionally equivalent behavior and either identical or non-identical syntax as IXCOM options, these `xrun`-command options are mapped to IXCOM. All other unused options (or not mapped to IXCOM tools) on the ***xrun*** command-line are ignored with a warning.

An option is considered as unused if both of the following conditions are true:

- The option is a recognized Xcelium tool (`xmvlog`, `xmvhdl` or `xmelab`) option.
- The option has no mapping specified in Xcelium-to-IXCOM option-mapping tables listed in this appendix.

This appendix lists the Xcelium-tool options mapped to the IXCOM options when using *xrun* for simulation acceleration as follows:

- IXCOM Options with the Same Syntax and Behavior as xrun Options
- IXCOM Options with Different Syntax but the Same Behavior as xrun Options

© 2021 All Rights Reserved.

IXCOM Options with the Same Syntax and Behavior as xrun Options

Following is the list of the Xcelium tool options that are mapped to IXCOM options with identical syntax and functionally equivalent behavior.

```
-append_log
-assert
-assert_vlog
-assert_vhdl
-clean
-coverage <string>
-default_ext <arg>
+define+<macro>
-dpiheader
-dpi_void_task
-f <file>
-F <file>
-help
-incdir <path>
+incdir+<path>
-l <file>
+libext+<extension>
+liborder
+librescan
+libverbose
+loadpli1+<library>
+loadvpi+<library>
-lps_cpf <file>
-neverwarn
-noassert
-override_precision
-override_timescale
-q
-sysv_ext [+]<extns>
-timescale <arg>
-top <du>
-v <file>
-verbose
-version
-vhdl_time_precision <arg>
-vlog95_ext [+]<extns>
-vlog_ext [+]<extns>
-vtimescale <arg>
-work <library>
-y <directory>
```

IXCOM Options with Different Syntax but the Same Behavior as xrun Options

The following table lists the Xcelium-tool options that have different syntax but functionally equivalent behavior as an IXCOM option.

© 2021 All Rights Reserved.

Table 26-1 xrun and IXCOM Options with Different Syntax but Equivalent Functional Behavior

xrun Option	IXCOM Option
`-32bit`	`-32` or `-32bit`
`-64bit`	`-64` or `-64bit`
`-define <macro>`	`+define+<macro>`
`-loadpli1 <library>`	`+loadpli1+<library>`
`-loadvpi <library>`	`+loadvpi+<library>`
`-propfile_vlog <file>`	`-propfile <file>`
`-propfile_vhdl <file>`	`-propfile <file>`
`-sv`	`+sv`
`-v1995`	`+v1995`
`-v200x`	`+v2000`
`-lexpragma`	`+rtlCommentPragma` Supports: ■ `cadence/synopsys/pragma translate_off/on` ■ `cadence/synopsys/pragma synthesis_off/on` Does not support `-rtl_synthesis off/on`
`-makelib <arg>`	`-work <lib>`
`-reflib <arg>`	`-L <lib>`
`-smartorder`	`-sort`
`-v93`	`-93`

See *Using xrun for Simulation Acceleration* guide for more information on the `xrun` comamnd options.

© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

27

Compiling and Running Designs with Assertions

Assertions represent the executable specification of a design and are primarily used to validate the behavior of a design. VXE supports assertions written in SystemVerilog and PSL, in addition to the Open Verification Library (OVL) components.

For a complete list of the SVA supported constructs, refer to Support for Specific SVA Constructs on page 1267.

For a complete list of the PSL supported constructs, refer to Support for Specific PSL Constructs in a Mixed-HDL Design on page 1282 and Support for Specific PSL Constructs in a Verilog Design on page 1298.

The OVL components are listed in Support for Open Verification Library Components on page 1309.

Important

The content of this chapter pertains only to SA mode.

Assertions can be mapped to the Palladium Z1 engine for Verilog and VHDL designs by using the IXCOM compiler. This mapping enables powerful debugging features, including hot-swapping, to be extended to assertions.

Caution

The performance of the software simulator is adversely affected as the number of assertions in the design increases. Therefore, mapping the assertions into the emulator provides an order of magnitude performance improvement over the software simulator.

Assertions can be included either as a part of the design (as embedded PSL/SVA) or by using external files (in verification units or vunits). Using vunits enables verification engineers to add assertions without modifying the design. Also, in this manner, assertions can be reused. For

© 2021 All Rights Reserved.

example, assertions written for one version of the design unit can be linked to another version of the design.

Methodology guidelines for compiling and running assertions for hardware are provided in Appendix 28, "Methodology Guidelines for Compiling Assertions with the IXCOM Compiler."

Assertion Examples

Examples of assertions are located in the following directory:

`<IXCOM_INSTALL_HOME/share/vxe/gift/tutorial/assertions`

The `assertions` directory has the following two sub-directories:

- `PSL` - This directory includes examples of PSL assertions written within VHDL and Verilog modules.
- `sva` - This directory includes examples of SystemVerilog assertions written within Verilog modules.

Each example has an associated README file that describes the example, lists the related files in the directory, and provides instructions on how to compile and run the design.

Enabling the Use of SVA

SystemVerilog Assertions (SVA) are part of SystemVerilog; therefore, to compile assertions, you must use the `+sv` (SystemVerilog) option during compilation with the `ixcom` and `vlan` commands. (For more about compilation, refer to Chapter 16, "Compiling Designs for Simulation Acceleration Using IXCOM.")

The `+sv` option indicates that the SystemVerilog constructs and keywords should be honored. The assertions under the DUT will be mapped to the emulator, and the assertions in the testbench will run in software simulator. No additional option is required to map SVA constructs to the emulator. All assertion-containing modules under the hierarchy of the module specified with the `+dut` option will be mapped to the emulator.

For more information on the `+sv` option, refer to the option description in the *Commands and System Tasks for SA Mode* chapter in *VXE Command Reference Manual*.

Enabling the Use of PSL Assertions

To ensure that the PSL assertions are recognized, specific options need to be passed to the `vhan`, `vlan`, and the `ixcom` commands.

© 2021 All Rights Reserved.

- `-assert` option enables parsing of PSL assertions in the DUT. Parsing of PSL assertions is *not enabled* by default. You can specify the `-assert` option with `vhan`, `vlan,` and `ixcom` commands.
- `-propfile` *`<file>`* option ensures that PSL assertions specified in external files (other than the design file) are recognized and parsed. The *`<file>`* option refers to the external files containing PSL assertions, such as vunits. This option can be specified with `vhan`, `vlan`, and `ixcom` commands.
- The `-noassert` and `+no_psl` option ensure that the PSL assertions are not elaborated in the design. Also, you can specify selected modules with these options to ensure that PSL assertions in the specified modules are not elaborated. These options can be specified *only* with the `ixcom` command. (See Disabling PSL Assertions in Selected Modules.)

Both the VHDL and Verilog flavors of PSL assertions are supported. Therefore, depending on the language, PSL assertions can be parsed and elaborated in following two modes:

- **Two-step compile mode**

  In this mode, `-assert` option is specified with the `vlan` command to parse PSL assertions in a Verilog design, and with the `vhan` command to parse PSL assertions in a VHDL design. After the PSL assertions are parsed, the `ixcom` command elaborates the PSL assertions at compile time.

- **Single-step compile mode**

  In this mode, `-assert` option is passed to the `ixcom` command that parses and elaborates the PSL assertions in a design. The single-step compile mode is applicable to Verilog designs only.

Disabling PSL Assertions in Selected Modules

You can disable elaboration of PSL assertions that have been analyzed or parsed using the -psl option by using the feature-file `psl disable` rule or the `+no_psl` option of the `ixcom` command.

No additional option is required to map PSL constructs to the emulator. All the modules that have been specified with the `+dut` option, and have assertions will be mapped to the emulator.

© 2021 All Rights Reserved.

The following table illustrates the feature-file rules and options for disabling PSL assertions.

Feature-File Rule	Description
`psl disable du=<modName1>`	Disables PSL assertion elaboration in the `<modName1>` module.
`psl disable instance=<tb_top.inst.inst1>`	Disables PSL assertion elaboration in the `<tb_top.inst.inst1>` module instance.

Labeling PSL Assertions

For unique identification, Cadence recommends that each PSL assertion is appropriately labeled. A label enables easy identification of an assertion in all forms and reports. If you do not provide a label to an assertion, a system-generated label is used.

The syntax for labeling PSL assertions is:

```
[ Label : ] Verification_Directive   (IEEE Std. 1850-2005, Section 7.1)
```

For example:

For PSL assertions embedded in comments in the HDL, use:

PSL assertions in Verilog HDL:

```
// psl  my_label: assert <assertion>
```

Or, PSL assertions in VHDL:

```
-- psl my_label: assert <assertion>
```

For PSL assertions defined within a vunit, use:

```
my_label: assert <assertion>
```


© 2021 All Rights Reserved.

Command Options for Compiling Designs with Assertions

The following table lists the options for compiling designs with assertions. These options are syntactically and functionally the same for both `ixcom` and `xrun -hw` commands.

Option	Description
`+sv`	Enables SystemVerilog assertions.
`-assert`	Enables PSL features in HDL files.
`-assert_vlog`	Enables PSL features in Verilog files.
`-assert_vhdl`	Enables PSL features in VHDL files.
`-noassert`	Disables PSL and SystemVerilog assertions.
`-propfile_vlog`	Enables all the PSL assertions found in the file specified with the `-propfile_vlog` option, and all the assertions in any other HDL files specified on the command line.
`-propfile_vhdl`	Enables all the PSL assertions found in the file specified with the `-propfile_vhdl` option, and all the assertions in any other HDL files specified on the command line.

Enabling and Disabling Assertions at Compile Time

Use the `-featurefile` option of the `ixcom` command to enable or disable assertions as a feature at compile time. You need to specify the following keywords in the feature file to describe the assertions.

- `sva` - Enables or disables the SystemVerilog Assertions
- `psl` - Enables or disables the PSL assertions.
- `assert` - Enables or disables the SVA and PSL assertions.

The `-featurefile` option can be used only with the `ixcom` command. It cannot be used with the `vlan` and `vhan` commands. Using the `-featurefile` option controls the assertion synthesis. Therefore, only the assertions available after analysis can be disabled or enabled by using the `-featurefile` option.

Following is a brief description of how the assertion-related options can be specified in conjunction with the `-featureFile` option:

© 2021 All Rights Reserved.

Analysis-specific Assertion Options: The `-assert`, `-psl`, and `+no_sva` options of the `vhan` and `vlan` commands control the parsing or analysis of assertions. The assertions that are analyzed are available during elaboration for synthesis, and these can be controlled by using the `-featureFile` option.

Elaboration-specific Assertion Options: The `-featureFile` option can be used during elaboration for disabling or enabling synthesis of assertions. By default, all the analyzed assertions are synthesized.

You can override this behavior by using the `-featureFile` option. Note that you can achieve the same objective by using the `-noassert`, `+no_psl`, and `+no_sva` options, and override the default behavior; But you cannot use these options together with the `-featureFile` option to enable or disable SystemVerilog assertions. Use only one of these options, and recompile.

Note that PSL assertions written in comments are not enabled by default and the `-psl` option is needed with the `ixcom` command to enable the PSL assertions.

For a detailed description of the usage of the `-featureFile` option, refer to Enabling and Disabling Features at Compilation Time.

The following table lists the frequently used feature file configuration options.

Feature Name	Description
`sva`	Feature to enable or disable SVA during elaboration.
`psl`	Feature to enable or disable PSL assertions during elaboration.
`assert_user_action_block`	Feature to control keeping or ignoring user specified action block of SVA.
`assert_status`	Feature to enable or disable evaluation of assertions at run time.
`assert_counter`	Feature to enable or disable configuration of counters that are mapped to hardware
`assert_cover`	Feature to enable or disable creation of coverage counters for assert and assume directives.

© 2021 All Rights Reserved.

Functional Coverage Support

The PSL and SystemVerilog Assertion (SVA) languages provide functional coverage information about the designs and are integrated with the Integrated Metrics Center Tool.

This section covers the following topics:

- Generating Assertion-Based Functional Coverage Using VXE
- Concurrent Cover Statement
- Integration with Integrated Metrics Center Tool

Generating Assertion-Based Functional Coverage Using VXE

There are two types of functional coverage, namely covergroup-based functional coverage and assertion-based functional coverage. This section describes the generation of assertion-based functional coverage using VXE.

Generating the assertion-based functional coverage database using VXE includes assertion-based coverage information in the VXE-generated coverage database in the VXE domain.

To enable creation of VXE Unicov database for assertion coverage, compile the design using the `-coverage functional` option of the `ixcom` or `xrun -hw` command:

Example:

```
ixcom -ua +dut+dut -assert +sv -coverage functional test.v
```

The following ***xrun*** run-time options are supported for assertion coverage:

- `-covmodeldir` <*arg*>
- `-covnomodeldump`
- `-covoverwrite`
- `-covscope` <*arg*>
- `-covtest` <*arg*>
- `-covworkdir` <*arg*>

VXE creates the coverage database for all the assertions processed by IXCOM compiler:

- In the testbench and DUT partitions

© 2021 All Rights Reserved.

Inside exported netlist hierarchy

- In the modules specified with the `+xcDesignTop` option (This essentially implies that the coverage database for assertions in the testbench hierarchy that is passed directly to simulator is written by Xcelium in the ICC domain.)

VXE creates the coverage database for assertions at the end of simulation which can further be loaded and analyzed using IMC. Refer to the Integrated Coverage User Guide available in the *Xcelium documentation set* for further details of analysis and report generation from VXE-generated coverage database. Xcelium tools and Integrated Comprehensive Coverage (ICC) must be installed to create and interact with the coverage data.

Integration with Integrated Metrics Center Tool

You can view functional coverage information using the Integrated Comprehensive Coverage Metrics Center tool. IXCOM writes out the checked, failed, and finished counts for each assertion in the coverage database, at run time. To enable this option, specify the following option with the `ixcom` command:

```
-coverage functional
```

The `-coverage functional` option creates the functional coverage database to store the functional coverage information. IXCOM records the coverage data in `cov_work/scope/test/uxe_<design-checksum>.ucd` file, at run time. The `uxe_<design-checksum>.ucd` file contains the functional coverage information that can be viewed by using the ICC tool.

The Integrated Metrics Center viewer is located in the Xcelium release tree.

Concurrent Cover Statement

IXCOM uses a `concurrent cover` statement to give you the following counts:

- `Checked count` is the number of times the antecedent of the implication becomes true.
- `Finished count` is the number of times the property succeeded.
- `Failed count` is the number of times the property failed.

By default, IXCOM keeps these coverage counters only for SVA cover statements. If you need coverage counters for all concurrent assertion statements, that is for SVA `assert` and `assume` statements, specify the `assert_cover` feature.

© 2021 All Rights Reserved.

By default, all of these counters are one-bit counters; but you can increase their size using the `assert_counter` feature and its variants. The larger the size of the counters, the larger the resources required to track the counters in hardware. The value of these counters is printed at the end of the simulation as the ASSERTION COVERAGE SUMMARY.

SystemVerilog Bind Support

There are two ways to use SystemVerilog assertions in designs:

- Instantiate the module that models the assertions. This requires you to make a change to your design. The advantage of this method is that the assertions travel with the design and can also be used by other tools, for example, formal verification tools.
- Bind the assertion modules to the design modules or the instances of the design module using the SystemVerilog `bind` construct. The biggest advantage of this approach is that verification teams do not have to make any modification to the design RTL.

The SV `bind` construct is fully supported. Using the `bind` construct, you can, in effect, instantiate a module containing SVA in the design RTL. This construct also enables the flexibility of instantiating the assertion module inside only the selected instances of the target design module.

Using Assertion Control Tasks (ACTs) in HDL

IXCOM supports the following three assertion control tasks:

- `$asserton`

  Re-enables the execution of all specified assertions.

- `$assertoff`

  Stops checking of all specified assertions until a subsequent `$asserton`.

- `$assertkill`

  Aborts execution of any currently executing specified assertions and then stops the checking of all specified assertions until a subsequent `$asserton`.

For IXCOM to be able to parse, elaborate, and dump these assertion control tasks, specify the `+assertStatus` option with the IXCOM command. If the `+assertStatus` option is not specified, the assertion control tasks are ignored.

© 2021 All Rights Reserved.

Usage of Formal Arguments of Property and Sequence Declarations in Constant Expressions

The actual argument corresponding to formal argument of property and sequence declaration can be a non-constant expression as well as a constant expression. IXCOM facilitates semantic checks to allow reference of formal argument in a context where only constant expression is expected, and a constant expression is passed as a corresponding actual argument.

Consider the following example in which the `A1` assertion is legal and the `A2` assertion is illegal.

```
module test(clk);
   input clk;
   localparam NMB_LANES=2;
   bit enable;
   bit [NMB_LANES:0] lane_1;
   bit [NMB_LANES:0] lane_2;
   bit kk;
   default clocking @posedge clk);endclocking
   property P1(idx);
      enable
      && lane_1[idx]
      && ~|lane_1[idx:0]
      |-> lane_2[idx] == lane_2[0];
   endproperty
   A1: assert property(P1(5));
   A2: assert property(P1(kk));
endmodule
```

Here, **`idx`** is a Formal argument and **`5`** in `P1(5)` and **`kk`** in **`P1(kk)`** are actual arguments.

The actual argument **`kk`** is a non-constant in assertion `A2` that effectively makes the part-select range expression `[idx:0]` a non-constant in `lane_1[idx:0]`. Note that non-constant part-select range expressions are not legal and an error is generated if found in the design.

Assertion Coverage of Assertions with Constant Property

You can write property expressions for an assertion that either always passes or always fails. By default, IXCOM evaluates such assertions during elaboration and reports the results in the log file. This removes the capacity impact of these constant assertions.

In the following example, the cover property `C1` has property `1'b1` as an expression that is a constant and always remains true. The cover property `C1` is evaluated at every clocking event `@(posedge clk)` and passes as the property expression always remains true.

```
C1: cover property (@ (posedge clk) disable iff (~reset_n) (1'b1));
```


© 2021 All Rights Reserved.

Feature-File Configuration for Optimization of Static Assertions

You can configure to evaluate those assertion during run time by stopping evaluation during elaboration. This is useful in procedural context with cover directives. The assertions with constant property expressions are identified and evaluated during elaboration by default. The following feature-file configuration can be used for changing default behavior:

- `assert_optimize_const_property enable`: Enables identification and ignoring of static assertion during elaboration in the design. This rule is also is the default rule for a design.
- `assert_optimize_const_property disable`: Disables identification and ignoring of static assertions during elaboration in the design.

Specifying Varying Widths for Assertion Counters

You can specify the widths of the assertion counters according to your requirements and control the resource usage on the emulator. The IXCOM compiler enables you to specify different widths for the Checked, Finished, and Failed counters.

You can enable the following features in a feature file <*feature_file*>, and specify the name of the <*feature_file*> with the `ixcom` command to specify the different widths:

- `assert_counter_check` - Configure this feature to set the width of the checked counter to the value <*Nbits*>, as follows:

  ```
  assert_counter_check <Nbits> du=<dut_ip5+0>
  ```

- `assert_counter_finish` - Configure this feature to set the minimum finished counter width to <*Nbits*> for assertion for the `dut_top.dut_ip2.dut_ip3` instance and up to two levels of its sub-hierarchy.

  ```
  assert_counter_finish <Nbits> instance=<dut_top.dut_ip2.dut_ip3+2>
  ```

- `assert_counter_fail` - Configure this feature to set the minimum failed counter width to <*Nbits*> for assertion for the `dut_top` module and its sub-hierarchy.

  ```
  assert_counter_fail <Nbits> du=<dut_top+0>
  ```

Running and Debugging Designs with Assertions

This section describes the procedure for exercising and debugging designs with assertions on Palladium Z1 systems in SA mode. For compiling assertions in any of the other supported modes, for example, Dynamic Target or Synthesizable TestBench (STB), you need to compile the design with assertions using IXCOM.

© 2021 All Rights Reserved.

You can use the emulator to test the assertions expected to be exercised at the end of long verification runs, within reasonable '*real*' time, which would not be so achievable in traditional simulation. Running designs with assertions does not require any emulator resources in addition to those used during a normal run, except for the additional hardware gates used for assertion instrumentation. This solution enables you to see the assertion activity on the emulator as they do during simulation.

Examples of SVA Usage on page 1237 provides examples of compiling designs with SVA in various modes and debugging them using xeDebug tool.

Examples of PSL Assertion Usage on page 1244 provides examples of compiling designs with PSL in various modes and debugging them using xeDebug tool.

The *Assertions* tab allows you to perform the following tasks:

- Browsing Assertions

  Browse the assertion hierarchy and enable and/or disable assertions at the module level by specifying the depth. The *Path* field specifies the current scope. You can filter the current displayed hierarchy by specifying the filtering condition in the *Filter* field. Select the assertions to be displayed with their depth, and click *To Display*.

- Selecting Assertions

  View the assertions in the selected module and enable and/or disable assertions individually. The selected assertions are then displayed. Additionally, filter the assertion list - *by type* and *by state* using the *Name Filter* field. The options in the *Name Filter* field are used to provide rapid specification of the assertions listed in the assertion table, based on the directive type, current state, and name patterns.

  To operate on a subset of the compiled assertions, directly select one or more adjacent assertions from the Selection Filters.

- Displaying Assertions

  The selected list of assertions are displayed in the assertion table. The selected assertions can be removed from the display and you can update states or choose to do so automatically after every run. Anytime the emulator is run, you must click the *Update States* button to see accurate assertion state status. The assertion states displayed in the assertion table read *n/a* when they are first entered to the *Assertion* tab, and whenever a `run` command is executed. They show n/a status until the *Update States* button is selected. The *Update States* button is also highlighted to indicate that it is required to refresh the state values.

© 2021 All Rights Reserved.

Figure 27-1 Assertions Tab

To get a summary, click *Get Summary* button. The *Get Summary* button launches a window displaying the Assertion Summary report. To get a summary or schedule a final summary, click the *Schedule Final Summary* button. The *Schedule Final Summary* button specifies that the summary report should not be generated immediately; instead the report is generated only on exiting from the xeDebug tool. Probe the selected

© 2021
All Rights Reserved.

assertions from the assertion table by clicking the *Probe* button. Save/import the list of displayed assertions by clicking the *Save* and *Import* buttons, respectively.

Status bars above and below the assertion table provide status information. The status bar above the assertion table provides information on the number of unique assertion instances currently compiled into the Palladium Z1 DUT, the number of valid cover points in the compiled assertion set, the number of assertions currently enabled, and the number of assertions filtered to be displayed in the assertion table. The assertion table itself presents the status for each assertion displayed with respect to directive type, current status with respect to enabled/disabled and coverage database selection, and current state. The status bar below the assertion table reflects how many assertions are currently in each state. For this status to be accurate you should be sure to use the *Update States* button after each run to force updated state retrieval. Additionally, the console window provides visibility to the stdout stream and, thus, to any command success and error status.

Debugging Assertion Failures

At CLI prompts, you can disable or enable assertions (by default, assertions are enabled), request an assertion summary, or specify other assertion-related commands. At the time of running the `ixcom` compilation command, you can exclude instrumentation of selected assertions. Additionally, due to implementation restrictions, it might not be feasible to instrument some assertions and those will be ignored. There are no other means of including or excluding assertions.

All assertion failures are reported by the instrumentation. Assertion passes (finishes) are reported only when the SVA assertion has a pass-related action block. Pass-related action blocks are processed on the emulator.

© 2021 All Rights Reserved.

If assertion failures are reported, by default, emulation continues until the specified run interval has been completed.

By default, the instrumented assertion model is tuned to use low or moderate capacity. If you need to debug assertions, specify the `+assertStatus` option with the `ixcom` command at the time of compiling the design. This debug choice of using the `+assertStatus` option entails the use of additional hardware resources and sacrifices some run-time performance.

If required, capacity and performance can be sacrificed to get more debug features by specifying the `assert_status` feature in the feature file. This includes:

- Querying the status of the assertion and inspecting the fanins of the assertion
- Debugging using SimVision or Verdi
- Enabling/disabling assertions
- Other assertion run-time features supported by Xcelium

When the `assert_status` feature is enabled, the assertions can be turned off and on at any point of time during simulation.

Xcelium Simulator Required for Exercising Assertions

To enable the exercising of assertions on Palladium Z1 system, it is necessary to have the Xcelium simulator running concurrently with the emulator for all the supported use models. **An Xcelium license is always required to run designs with assertions.** Assertions are not supported in a non-simulator (ICE) mode. Therefore, for designs with assertions, even the emulation you need to compile using IXCOM and load the design into the simulator. A distinct advantage of the presence of the Xcelium simulator is that it enables support for enhanced Hot Swap capabilities when running designs with assertions. Identical assertion support is provided for both the DUT design partitions in the emulator and simulator. The run-time engine transparently determines the active engine and makes the required adjustments accordingly.

The run-time functionality is defined by the compile-time instrumentation decisions, such as presence of counters and their widths. These decisions are specified by using the relevant `ixcom` command options.

For details of compiling designs with IXCOM, corresponding assertion compilation options, and the supported assertions set, refer to Chapter 16, “Compiling Designs for Simulation Acceleration Using IXCOM.”

© 2021 All Rights Reserved.

The supported verification packages (OVL, IAL, VIP), languages (PSL, SVA) and various constructs are listed in the following sections:

- Support for Specific SVA Constructs
- Support for PSL Constructs
  - Support for Specific PSL Constructs in a Mixed-HDL Design
  - Support for Specific PSL Constructs in a Verilog Design
- Support for Open Verification Library Components

Support for Assertions on Emulator

When assertions fail or pass in these modes, the emulator is stopped, and the control is passed to the simulator to process the event. When done, the changes are updated in the emulator and emulation continues.

Assertions are disabled or enabled by invoking the corresponding Xcelium Tcl assertion commands. These will handle the setting/clearing of the value of the assertion-specific disable signal.

In the waveforms, the assertion state values and counters in the simulator databases or hardware-instrumented counters in the emulator databases can be traced. Use the `assertion -get` command, to access the assertion state value. Use the `assertion -summary` or `assertion -counter` command to access the counter values.

Note: In SA mode, the testbench assertions reside only in the simulator partition.

The `reset` command in xeDebug can generate the following warning if assertions are disabled before specifying the `reset` command:

```
xmsim: *W,FRCACC: Potentially forced object, object does not have read and
connectivity access
```

When this warning is issued, assertions are not re-enabled as expected. However, this warning is not generated when using *xrun* tool.

Using Assertions with InfiniTrace

In the `infiniTrace -prepare` sessions, complete assertion support is provided. In the `infiniTrace -observe` session, the assertion support is limited to providing waveform support of assertion states, hardware-instrumented counters, and assertion state values. Tracing is supported only for assertion state values and hardware-instrumented counters.

© 2021 All Rights Reserved.

Also, assertion run-time and `assertion -summary` commands are not supported in the `infiniTrace -observe` mode.

The following assertion-related commands are supported in the `infiniTrace -observe` mode:

- `assertion -get`
- `assertion -list`
- `assertion -check`
- `assertion -counter`
- `probe`

The following assertion-related commands are not supported in the `infiniTrace -observe` mode:

- `assertion -disable`
- `assertion -enable`
- `assertion -summary`

SDL Support for Assertions

SDL supports conditional expressions that reference assertions.

Specifying the assertion name implies a reference to its state value. For example, the following statement results in an SDL trigger if the `DUT.AssertAlwaysReady` assertion enters a `finished` state.

```
If ( DUT.AssertAlwaysReady == "finished" )
```

Note: All five legal assertion states are supported. Refer to Assertion States on page 1233.

SDL expressions referencing assertion counters are not supported.

Invalid assertion names are treated the same as the signal names that cannot be found and result in the following kind of error messages when the `sdl -verify` command is executed:

```
Error in expression 'assert1: signal 'assert1' does not exist, file <sdlFilename>,
line <sdlFileLineNum>.
```

The predefined signal, `ASSERTION.FAILURE` enables triggering an action on an assertion failure.

For example:

© 2021 All Rights Reserved.

```
if (ASSERTION.FAILURE)
{
exec “puts \“Assertion Failure Detected\"”;
trigger;
}
```

This condition evaluates to true when any assertion fails. You can specify an action when this condition is met. For example,

- Capture waveforms related to the context of assertion failure.
- Use the `assertion -list -failed` command to identify failed assertions. The returned list is not limited to the assertions that caused the trigger, but all the failing assertions from time `0`.

End-of-Sim Assertion Checks

Some assertions, such as a `strong eventually!`, if in the `active` state, are required to complete before the end of the session. If not satisfied, these are considered as failures, and must be reported. These end-of-sim evaluation checks are automatically performed when the session terminates.

These checks are not initiated when the entire design is swapped to the simulator using Hot Swap feature. When the design is swapped to the simulator, the emulator is released, but the session is not terminated.

The end-of-sim failure counts are included only in the `-final` assertion summaries.

probe Command

The `probe` command can be used with the following options to include assertions in the tracelist. This command can be used to trace both testbench and DUT assertions in SST2 format. For more details of this command, refer to the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

```
probe -create {<object> | <scopeName>}
      -depth { n | all}
      -assertions -state -signals
```

Pertaining to assertions, the above options perform the following actions:

`-create`	Creates a probe for the specified assertion(s) or the assertions within the specified scopes.

© 2021 All Rights Reserved.

`-depth`	Selects assertions contained in instances within <*n*> or all levels of the specified scope.
`-assertions`	Specifies that only assertion objects should be selected from specified scopes. **Note:** Specific assertion paths can be probed without the `-assertion` switch.
`-state`	Captures and displays assertion state values. This is the default option.
`-signals`	The `-signals` option adds the fanin signals for the specified assertions (that you have either enumerated or included in the specified scope).

Breakpointing Assertions

Assertion breakpointing is supported using the existing native Xcelium `stop` command and `assert_stop_level` and `assert_output_stop_level` Tcl variables. These variables are described in the *Xcelium Assertion Checking in Simulation* manual.

Support for Assertion Fanin Signals

To debug assertion failures, it is usually necessary to identify the assertion inputs and their values leading up to the failure. In addition to knowing the assertion state and counters, knowledge of the fanin signals is required to probe these without probing the entire scope.

To address this requirement, the `-signals` option is provided with the `probe` command and the `assertion -list` command. The syntax of the command options to enable support for assertion fanin signals browsing support is consistent with the syntax used in *xrun*.

In general, the options of the `probe` and `assertion` commands apply to all those objects that have been named in the command and for which these options make sense. For example, the `-depth` option applies to all the scopes and the `-signals` option applies to all assertions, and so on, regardless of their position in the command. These options do not apply only to the immediately preceding or following objects.

When you use the `-signals` option with either of the `probe` or `assertion -list` commands, the assertion fanin signals are automatically selected, provided any assertions are specified. Use this option with any other options of the `probe` and `assertions -list` commands, and even with the Dynamic Target mode.

© 2021 All Rights Reserved.

The `-signals` option of the `probe` command adds the fanin signals only for the specified assertions (that you have either enumerated or included in the specified scope):

```
probe [[-assertions] [-signals]]... <pathnames> ...
```

Specify *locator* arguments including multiple paths and scopes with the `-signals` option.

```
assertion -list -signals [-depth <level>] \
                         [-directive <directiveList>] <pathnames>...
```

The list of fanins is returned as a Tcl output in the form of a Tcl list of Tcl lists. Each top-level list identifies one selected assertion, and its corresponding list of fanins:

```
{{<assertionName1> {<faninSig1> <faninSig2> ... <faninSigN>}
 {<assertionName2> {<faninSig1> <faninSig2> ... <faninSigN>}
        ...
        {<assertionNameN> {<faninSig1> <faninSig2> ... <faninSigN>}}
```

The paths of all the assertions and fanin signals include the full hierarchy.

You can, for example, use the following code snippet to see a listing of all the assertions with fanin signals:

```
foreach assertEntry [assertion -list -signals -depth all top] {
        puts "Assertion:  [lindex $assertEntry 0] "
        foreach faninSig [lindex $assertEntry 1] {
                                                    puts "$faninSig"
                                                  }
}
```

When you use the `-signals` option with other options of the `assertions -list` command, the Tcl output uses the above-mentioned nested-list formatting.

Keep the following points in mind when probing fanins of the assertions:

- The `-signals` option is not applicable to non-assertion objects. If you include only non-assertion paths, and have not specified any scope arguments, a warning is generated. So, if you use the `-signals` option with the `probe` command, you must specify the paths of the assertions.
- The `-assertions` option is applicable only for scopes.If you include the `-assertions` option, but no assertions are included in the specified scope(s), a warning is generated. If an assertion path is explicitly specified, the assertion and its fanin signals are probed. The `-all` option also implies the `-assertions` option. In the above cases, the probe action will still be processed.

Examples of Using the -signals Option

Here are a few examples of using the `-signals` option:

© 2021 All Rights Reserved.

The following command probes all the assertions in the DUT hierarchy and all of their fanin signals. It also probes the `top.Assert1` assertion and its fanin signals.

```
probe -create -depth all -assertions -signals top.DUT top.Assert1
```

The following command probes all objects (signals and assertions) in the `top.DUT.Inst1` instance. It also probes the `top.Assert1` assertion and its fanin signals. The `top.Assert1` is not a scope. So, the `-depth` and `-all` options do not apply to it. In this case, the fanin signals for any assertions within the `top.DUT.Inst1` instance are already probed because of the use of the `-all` option. These will not be duplicated.

```
probe -create -all -signals -depth 1 top.DUT.Inst1 top.Assert1
```

The following command probes the `top.DUT.Assert_Always` assertion and its fanin signals, and the `top.clk` signal. Because the assertion path is already specified, the `-assertion` option is not required. Here, `top.clk` is not an assertion. So, the `-signals` option does not apply to it.

```
probe -create -signals top.DUT.Assert_Always top.clk
```

Using the Assertions Tab of the xeDebug GUI to View Fanin Signals

To view the values of the assertion fanin signals, in addition to the `probe` and `assertion` XEL commands, you can use the *Probe with Fanin* button on the *Assertions* tab of the xeDebug GUI as shown below. You do not explicitly specify the assertions in the *Probe* tab of the xeDebug GUI. You only select from the Assertions tab by selecting their parent scope. Then, the fanin signals will be probed through the scope, without needing any other action.

These fanin signals are included in the probe list, but these are not directly associated with the assertion.

© 2021 All Rights Reserved.

This means that unlike the assertion counters, these signals are not displayed *within* the assertion object in the waveforms. Also, like other objects, even if the same fanin signals are probed for multiple assertions, these are displayed only once.

Assertions within Protected IP

Assertions within protected IP are supported. If a protected assertion fails, the failure message will only indicate the time at which the assertion failed, and not provide any other details, such as assertion name. Assertion disabling will not be supported for these protected assertions.

Protected assertions are not included in the assertion summary. However, when these exist in the design, the summary will be followed by this message:

```
Part of the design was protected.  Affected assertions are not shown.
```

For details on handling of protected assertions, refer to the *"Writing Assertions for Protected IP"* in the "*Assertion Writing Guide*" available in the Xcelium documentation set.

© 2021 All Rights Reserved.

Coverage Support

If the design has been compiled with the `-coverage functional` option of the `ixcom` command, assertion coverage is supported at run time.

Assertion Reporting

Assertion status information is provided to you in three forms:

- Waveforms
- Messages provided for failed and passed assertions
- Assertion summary report

This information is written to stdout, the simulator log file (`xmsim.log`), and the emulator run-time log file (`xeDebug.log`).

The following table summarizes the assertion-related output.

Table 27-1 Assertion Reporting Output

	stdout	xeDebug.log	xe.msg	xmsim.log
Failure/Pass	Yes	Yes	Yes	Yes
Summary Report	Yes	Yes	No	Yes
Echoing of Commands	No	No	Yes	No

Assertion Summary Report

The assertion summary report is of the following form:

```
Checked?  Finish   Failed  Assertion Name
31        17       14       test.TB_A1
31        22       5        test.inst1.A1
0         0        0        test.inst1.ABORT1 (./dut.vp:45)
24        24       0        test.inst1.Cover2 (./dut.vp:40)
52        0        0        test.inst1.EVENT1 (./dut.vp:42)
28        22       6        test.inst1.MP_PROT_ASSERT
```


© 2021 All Rights Reserved.

```
1          0          1          test.inst1.StaticAlways (./dut.vp:20)

Total Assertions = 7, Failing Assertions = 3, Unchecked Assertions = 1
Assertion summary at time 200 NS + 0
```

Contents of the Assertion Summary Report

Each entry displays the checked, finished, and failed counts (or subset specified by the `-show` option) with full hierarchical assertion path.

The count for any of the assertion counters listed in the summary report freezes on the maximum value for the counter bit width after that value has been achieved. There is no way to differentiate between a counter which has attained or surpassed the maximum value.

A footnote summary briefly indicates the total numbers of assertions — those that have failed at least once and those never exercised.

In the summary, the protected assertions that share the same 'visible' hierarchical path are consolidated into a single entry.

Both testbench and DUT assertions are presented in the same summary report.

Debugging Assertions Using Waveforms

You can view the assertion waveforms using either SimVision or Verdi. Accordingly, specify the database format as SHM (default) for SimVision or FSDB for Verdi.

SimVision integration with the xeDebug tool enables display of assertion waveforms and the Assertion Browser data from both the emulator and simulator engines.

VXE provides a loose integration with Verdi, in terms of compatible database formats for debugging capabilities. However, appropriate licenses are required to use Verdi.

This support does not include the Property Result Table feature (Verdi Assertion Browser). However, if you independently license the Property Result Table, it will display correct results.For information on integrating Novas Tools (Verdi or Siloti), refer to Chapter 47, "Using Novas Tools."

The VCD trace format does not support assertion objects and using this format will result in only the raw instrumentation signals.

© 2021 All Rights Reserved.

Important

To be able to debug assertions, it is essential that at compilation time, the `assert_status` feature should be enabled.

Assertions are supported in both testbench (software simulator) and DUT (emulator). Emulator assertions can be viewed using Verdi and SimVision. For the simulator assertions, waveforms are dumped only in the SST2 format and only the SimVision browsers are supported. Simulator assertions include the testbench assertions and all assertions after Hot Swap to simulator.

When a portion of the trace interval is simulated after a Hot Swap, the simulator waveforms will be dumped to a separate SST2 database. At viewing, as with other traceable objects, the assertion waveforms can be included in both a simulator and emulator trace database. The relevant waves will be in the database for which that assertion was processed during the desired interval.

When the Assertion Browser window is opened, the displayed values are associated with the waveform time selected. This functionality is analogous to the Xcelium-only simulation. All of the assertions and related data are presented in the same Assertion Browser window regardless of which engine is currently processing the assertion.

No assertions or corresponding fanin signals are automatically added to the tracelists – you must add these signals explicitly.

Assertions can be added to tracelists and included in subsequent uploads analogous to other trace objects. The assertion trace-related attributes include:

- Assertion state value
- Tracing of assertion counters, if instrumented with the `assert_cover` feature enabled

Adding assertions to the tracelist is accomplished using the `probe` command. If assertions are explicitly selected or contained within the selected scope, these are included in the tracelist.

Assertion States

The assertion strobe provides visibility into the state of the assertions, which include:

`off`	Indicates that the instrumentation for the particular assertion has been made non-active by an `assertion -disable` command.

© 2021 All Rights Reserved.

`failed`	Indicates that the fulfilling condition of the assertion has evaluated to false and represents an assertion failure.
`finished`	Indicates that the fulfilling condition of the assertion has evaluated to true or has discharged without failing. In the case of a cover directive, this signals a coverage completion.
`active`	Indicates that the assertion is enabled. The first sub-element of the enabling condition is satisfied and the assertion has not finished or failed.
`inactive`	Indicates that the assertion is enabled. Currently, there are no matches with the sub-elements of the enabling condition for the assertion.

The above list also indicates the priority of the assertion states. This means that **OFF** has the highest priority and **INACTIVE** has the lowest priority. If multiple attempts of a given assertion are running simultaneously, the state probe indicates the state with the highest precedence. For example, if an assertion simultaneously passes and fails, the state probe will display FAILED.

Assertions-Related Features Required at Compile Time

Certain `ixcom` command options related to assertions are mandatory at compile time for correct run-time assertion behavior.

- If the `assert_status` feature has not been enabled, assertion values and waveforms will not be available at run time. A warning message will be produced and the command for getting the assertion state values will be ignored.
- If the `assert_cover` option has been enabled, the coverage information (passed, failed, and finished coverage counters) is added for all `assert` and `assume` statements in the assertion summary report. If this option is not specified, counters are provided only for `cover` statements.
- If the `-coverage functional` option is used, assertion coverage will be supported.

Assertion Run-Time Commands

At run time, use run-time interface commands to:

- Obtain detailed assertion summary reports
- Debug any unexpected results or assertion failures using assertion logs and waveforms
- Modify the testbench based on coverage results

© 2021 All Rights Reserved.

- Control assertions by enabling or disabling assertions
- Manage probes and uploads
- Terminate or interrupt conditional emulation based on assertion failure

The assertion command set is primarily used to configure the assertion environment and produce the required analysis data. These XEL commands only affect the state of the assertion instrumentation and its settings in the XEL environment. These commands do not cause any simulation time to be run.

Following is the list of assertion run-time commands:

```
assertion -check
assertion -counter
assertion -get
assertion -off
assertion -list
assertion -on
assertion -summary
```

For a description of the assertion run-time commands, refer to *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

These commands are available at any time in the CLI after the design has been loaded. These are case-insensitive and can be abbreviated to minimum unique characters. These command invocations are echoed to the `xe.msg` log file.

Most of the assertions-related behavior is the same irrespective of the DUT run in the emulator or simulator.

All assertions are initially active. Use `assert_stop_level` variable of *xrun* to specify the stop severity level.

Assertion Tcl Variables

The following table identifies the Xcelium assertion-related Tcl variables. Only some of these are supported in xeDebug and some of those only partially. These variables can be set or queried in the xeDebug Tcl environment as well as the nested Xcelium Tcl interpreter using the `xmsim` prefix.

Refer to the Xcelium documentation for further description of these variables.

© 2021 All Rights Reserved.

Table 27-2 Assertions-Related Tcl Variables

Variable	Supported (Y/N)	Comments
`assert_1164_warnings`	Y	Full support
`pack_assert_off`	Y	Full support
`severity_pack_assert_off`	Y	Full support
`assert_stop_level`	Y	Supported in all modes except Dynamic Targets
`assert_output_stop_level`	Y	Supported in all modes except Dynamic Targets
`assert_report_level`	N	
`assert_stop_reason`	Y	Supported in all modes except Dynamic Targets
`assert_count_attempts`	N	Limited compile-time support
`assert_count_traces`	N	Limited compile-time support
`assert_report_incompletes`	N	Limited compile-time support

© 2021 All Rights Reserved.

Examples of SVA Usage

This section contains different examples of SystemVerilog Assertion (SVA) usage:

- Example 1: Using Concurrent Assert Statements
- Example 2: Using Concurrent Cover Statements
- Example 3: Using SV Bind
- Example 4: Functional Coverage Information

Example 1: Using Concurrent Assert Statements

In this example of how to use a concurrent assert statement, the property `P1` tests for signal *a* to be high in the first clock cycle, followed by signal *b* to be high, followed by *c* to be high in the next cycle.

```
`timescale 1ns/100ps

module top;
  reg [3:0] in1;
  reg [3:0] in2;
  reg clk;
  wire [4:0] out1;
  initial
  begin
    clk = 0;
    #100 $finish;
  end

  always
  begin
    #10 clk  = ~clk;
  end

  always
  begin
    #9;
    in1 = $random;
    in2 = $random;
    #11;
  end // always begin
  assert_model
inst1(clk,in1[3],in1[2],in1[1],in1[0],in2[3],in2[2],in2[1],in2[0]);
  always @(posedge clk) begin
    $strobe("%t strobe: abcdefgh = %b%b  ", $time, in1, in2);
  end // always @ (posedge clk)
endmodule

module assert_model (clk,a,b,c,d,e,f,g,h);
  input a,b,c,d,e,f,g,h;
  input clk;
  property P1;
    a ##1 b ##1 c;
  endproperty
```


© 2021 All Rights Reserved.

```
  A1:assert property (@(negedge clk) P1)
  $display("Assertion A1 has passed: ",$time);
endmodule
```

To run this design, perform the following steps:

1. Save it in a file as `sva_example.v`.
2. Run the following `ixcom` command:

   ```
   ixcom +sv -ua +dut+assert_model sva_example.v
   ```

3. Run the following `xeDebug` command:

   ```
   xeDebug <input_tclfile>
   ```

The default action when the assertion fails is to call `$error`. In this example, the assertion fails at time 0, time 20ns, and time 80ns.

Example 2: Using Concurrent Cover Statements

To see how to use a concurrent cover statement, change the module `assert_model` from Example 1 to the following:

```
module assert_model (clk,a,b,c,d,e,f,g,h);
  input a,b,c,d,e,f,g,h;
  input clk;
  property P1;
    a ##1 b ##1 c;
  endproperty

  A1:cover property (@(negedge clk) P1)
  $display("Assertion A1 has passed: ",$time);
endmodule
```

To run this design, perform the following steps:

1. Save it in a file as `sva_example2.v`
2. Run the following `ixcom` command. By default, only 1 bit of counters is kept. To increase the size of the counters, specify `assert_counter 10` in a feature file named `width.ff`, for example, and specify this file as an option to the `ixcom` command:

   ```
   ixcom -ua +dut+assert_model +sv -featureFile width.ff sva_example2.v
   ```

3. Run the following `xeDebug` command

   ```
   xeDebug <input_tclfile>
   ```

Example 3: Using SV Bind

In examples 1 and 2, you instantiated module *`assert_model`* inside module *`top`*. This requires a change in module *`top`*.

© 2021 All Rights Reserved.

To keep from having to change module `top`, you can create a *`bind_module`* with the following information in the file `bind.v`:

```
module bind_model;
bind top assert_model inst1(clk,
in1[3],in1[2],in1[1],in1[0],in2[3],in2[2],in2[1],in2[0]);
endmodule
```

The `bind` construct lets you effectively instantiate module *`assert_model`* inside module *`top`*.

Your module *`top`* will now be:

Note: There is now no instantiation of the module *`assert_model`* in the module *`top`*.

```
`timescale 1ns/100ps


module top;
  reg [3:0] in1;
  reg [3:0] in2;
  reg clk;
  wire [4:0] out1;
  initial
  begin
  clk = 0;
    #100 $finish;
  end

  always
  begin
    #10 clk  = ~clk;
  end

  always
  begin
    #9;
    in1 = $random;
    in2 = $random;
    #11;
  end // always begin

  always @(posedge clk) begin
    $strobe("%t strobe: abcdefgh = %b%b  ", $time, in1, in2);
  end // always @ (posedge clk)

endmodule

module assert_model (clk,a,b,c,d,e,f,g,h);
  input a,b,c,d,e,f,g,h;
  input clk;
  property P1;
    a ##1 b ##1 c;
  endproperty

  A1:cover property (@(negedge clk) P1)
  $display("Assertion A1 has passed: ",$time);
endmodule
```

To run this design, perform the following steps:

© 2021 All Rights Reserved.

1. Save it in a file as `sva_example3.v`.
2. Run the following `ixcom` command:

   ```
   ixcom +sv sva_example3.v bind.v
   ```

3. Run the following `xeDebug` command:

   ```
   xeDebug <input_tclfile>
   ```

You will get the same output from the simulator as in examples 1 and 2.

Example 4: Functional Coverage Information

In this example, you will see Xcelium dumping assertions in `sst2` format and related functional coverage information for the Integrated Comprehensive Coverage tool.

```
`timescale 1ns/100ps
module tb(clk,a,b,c,d,e,f,g,h);
   output reg a,b,c,d,e,f,g,h;
   output reg     clk;
   initial
     begin
    clk = 0;
    #100 $finish;
end

   always
     begin
    #10 clk = ~clk;
     end

   always
     begin
     #9;
     {a,b,c,d} = $random;
     {e,f,g,h} = $random;
     #11;
     end //always begin

   always @(posedge clk) begin
     $strobe("%t strobe: abcdefg = %b%b%b%b%b%b%b ", $time, a,b,c,d,e,f,g);
   end

endmodule // tb

module top;
   wire a,b,c,d,e,f,g,h;
   wire clk;
   tb TB(clk,a,b,c,d,e,f,g,h);
   assert_model inst1(clk,a,b,c,d,e,f,g,h);
endmodule

module assert_model (clk,a,b,c,d,e,f,g,h);
   input wire a,b,c,d,e,f,g,h;
   input wire clk;
   property P1;
      a ##1 b ##1 c;
   endproperty
   property P2;
```


© 2021 All Rights Reserved.

```
      b ##1 d;
   endproperty

   A1:cover property (@(negedge clk) P1)
     $display("Assertion A1 has passed: ",$time);
   A2:cover property (@(negedge clk) P2)
     $display("Assertion A2 has passed: ",$time);
endmodule
```

To run this design, perform the following steps:

1. Save it in a file as `sva_example4.v`.
2. Create a feature file, `example4.ff` with the following content:

   ```
   assert_status enable
   assert_counter 10
   ```

3. Specify the `-featureFile example4.ff` option with the `ixcom` command:

   ```
   ixcom +dut+assert_model +sv -featureFile example4.ff -coverage functional \
         -ua +dut+assert_model sva_example4.v
   ```

4. Run the following `xeDebug` command to connect to the emulator:

   ```
   xeDebug -gui
   ```

Figure 27-2 Assertions Tab

© 2021 All Rights Reserved.

5. Run the following commands:

```
xc on -xt0 -zt0 -run
run
```

To view the simulation waveforms in SimVision, open the simulation database inside SimVision. Highlight the assertion signals, and send them to target waveform windows. In the waveform in Figure 27-3, two assertions are shown. Their counters (that is, `checked count`, `finish count`, and `failure count`) can also be shown by expanding their corresponding assertion signals.

Figure 27-3 SimVision Waveform Window

To view functional coverage information using the Integrated Comprehensive Coverage tool (ICC), launch the application in GUI mode and then open the functional coverage file. In ICC, only the finished count is captured. See Figure 27-4 and Figure 27-5. Only one of the assertion signals out of two is covered. So, the control-oriented coverage is 50%.

© 2021 All Rights Reserved.

Figure 27-4 Integrated Comprehensive Coverage Tool

© 2021 All Rights Reserved.

Examples of PSL Assertion Usage

This section contains two examples of using PSL assertions in a Verilog or a VHDL design:

- Example 1: Using Embedded PSL Assertions (VHDL Only)
- Example 2: Using PSL Assertions in vunits (VHDL Only)
- Example 3: Using Embedded PSL Assertions (Verilog Only)
- Example 4: Using PSL Assertions in vunits (Verilog Only)

Example 1: Using Embedded PSL Assertions (VHDL Only)

This example consists of VHDL files, `test.vhd` and `dut.vhd`, and demonstrates the use of PSL assertions embedded with the design. The PSL assertions are embedded in the `dut.vhd` file.

For designs with PSL assertions, perform the following steps:

- Compiling VHDL Designs with PSL Assertions
- Debugging Designs with PSL Assertions

The `test.vhd` and `dut.vhd` files used in this example are given below.

test.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
entity tb is
end;

architecture a of tb is
  signal clk : std_logic := '0';
  signal reset : std_logic := '0';
  signal stim : bit_vector(5 downto 0);  -- stimulus
begin
  inst1: entity work.dut(a) port map ( clk => clk,
                                       rst => reset,
                                         a => stim(5),
                                         b => stim(4),
                                         c => stim(3),
                                         d => stim(2),
                                         e => stim(1),
                                         f => stim(0),
                                      out1 => open
                                     );
  clock_generator: process (clk)
  begin
     if clk = '0' then
       clk <= '1' after 5 ns, '0' after 10 ns;
     end if;
```


© 2021 All Rights Reserved.

```
  end process clock_generator;

  reset_generator: process
  begin
    reset <= '1' after 301 ns, '0' after 321 ns;
    wait;  -- forever
  end process reset_generator;

  stimulus_reader: process
    use STD.TEXTIO.all;
    file DATA : text open read_mode is "data";      -- stimulus data
    variable l : LINE;
    variable l2 : LINE;
    variable ok : boolean := true;
    variable tmp : bit_vector(5 downto 0);
  begin
    while (not endfile(DATA)) loop
      readline(DATA, l);
      read(l, tmp, ok);
      if (ok) then
        stim <= tmp;
        write(l2, now, right, 8);
        write(l2, tmp, right, 8);
        writeline(OUTPUT, l2);
      else
        report "Error reading from data file" severity NOTE;
      end if;

      wait until falling_edge(clk);
    end loop;

    wait;                              -- forever
  end process stimulus_reader;

monitor: process
begin
wait for 500 ns;
assert false report "Test completed" severity failure;
end process monitor;

end;
```

dut.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use STD.TEXTIO.ALL;

entity dut is
  generic (repcnt: integer := 2 );
  port ( clk: in std_logic := '0';
         rst: in std_logic := '1';
           a: in bit := '0';
           b: in bit := '0';
           c: in bit := '0';
           d: in bit := '0';
           e: in bit := '0';
           f: in bit := '0';
      out1: out bit := '0'
     );
end;

architecture a of dut is
  constant foo : integer := 3;
begin
```


© 2021 All Rights Reserved.

```
-- psl default clock is rising_edge(clk);
-- Sequences are covered
-- psl IMPL: assert [*2 to 3] |-> {a;b};
-- psl gen_cover2_S1: cover {a;b;c};
  process (clk)
  begin
    if rising_edge(clk) then
      out1 <= a or b or c or d or e or f;
  end if;

  end process;

  process (rst)
 begin
    if (rst = '1') then
      report "Reset occurred." severity NOTE;
    end if;

  end process;

end;
```

Following is the data file used for this example:

data

```
001001
110110
000010
100000
000000
100010
011000
001011
011100
```

Compiling VHDL Designs with PSL Assertions

To compile the VHDL design and testbench with PSL assertions, perform the following steps:

1. Run the `vhan` command with the `-assert` option to analyze the VHDL design and testbench.

   ```
   vhan -assert dut.vhd test.vhd
   ```

2. Create `vhpsl.ff` with following the following lines:

   ```
   assert_status enable
   assert_cover enable
   assert_counter 32
   ```

3. Run the `ixcom` command to elaborate and compile the VHDL design.

   ```
   ixcom -ua -coverage functional -featureFile vhpsl.ff -top tb
   ```

   The above command compiles the design with the assertions and creates the Xcelium snapshot. The PSL assertions in the design are elaborated by the `ixcom` command.

© 2021 All Rights Reserved.

4. Run the `xeDebug` command to view the Assertion Coverage Summary and record the functional coverage data.

```
xeDebug -gui
```

Log Files Generated at Compilation Time

The following log files are generated at compilation time:

- `vhan.log`:

  This file contains the output of the `vhan` command.

- `ixcom.log`:

  This file contains the output of the `ixcom` command.

- `xeDebug.log`:

  This file contains the output of the `xeDebug` command.

At run time, the error messages pertaining to assertion failure are displayed in the following format:

```
xmsim: *E,ASRTST (dut.vhd,27): (time 25 NS) Assertion :inst1:IMPL has failed
```

In the above format, `:inst1:IMPL` represents the path of the PSL assertion. `IMPL` is the label of the assertion directive and `inst1` is the path of the module from which the PSL assertion is fired. The filename in which the assertion directive is stored along with the line number is `dut.vhd` and `27`, respectively.

Note: The waveforms for checked and finished counts of the PSL assertions are dumped only if you have specified the `+assertCover` option of the `ixcom` command at the time of design elaboration. To dump and view the assertion status, use the `+assertStatus` option of the `ixcom` command.

Example 2: Using PSL Assertions in vunits (VHDL Only)

This example consists of VHDL files, `test.vhd` and `dut.vhd`, and demonstrates the use of PSL assertions in vunit. The PSL assertions are specified in vunit, which is the `prop.psl` file.

For designs with PSL assertions included as vunit, compile the design in the same manner as given in Example 1: Using Embedded PSL Assertions (VHDL Only) on page 1244. The steps to compile the design with vunit are given in Compiling VHDL Designs with vunit on page 1250. In contrast to compilation of designs with embedded assertions, when compiling designs with vunits, you must include the additional `-propfile` option with the `vhan`

© 2021 All Rights Reserved.

command. The rest of the compilation and debugging processes remain the same for both flows.

The `test.vhd`, `dut.vhd`, and `prop.psl` files used in this example are given below.

test.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

entity tb is
end;

architecture a of tb is
  signal clk : std_logic := '0';
  signal reset : std_logic := '0';
  signal stim : bit_vector(5 downto 0);  -- stimulus
begin
  inst1: entity work.dut(a) port map ( clk => clk,
                                       rst => reset,
                                         a => stim(5),
                                         b => stim(4),
                                         c => stim(3),
                                         d => stim(2),
                                         e => stim(1),
                                         f => stim(0),
                                       out1 => open
                                       );
  clock_generator: process (clk)
  begin
     if clk = '0' then
       clk <= '1' after 5 ns, '0' after 10 ns;
     end if;

  end process clock_generator;

  reset_generator: process
  begin
    reset <= '1' after 301 ns, '0' after 321 ns;
    wait;  -- forever
  end process reset_generator;

  stimulus_reader: process
    use STD.TEXTIO.all;
    file DATA : text open read_mode is "data";      -- stimulus data
    variable l : LINE;
    variable l2 : LINE;
    variable ok : boolean := true;
    variable tmp : bit_vector(5 downto 0);
  begin
    while (not endfile(DATA)) loop
      readline(DATA, l);
        read(l, tmp, ok);
      if (ok) then
        stim <= tmp;
        write(l2, now, right, 8);
        write(l2, tmp, right, 8);
        writeline(OUTPUT, l2);
    else
        report "Error reading from data file" severity NOTE;
    end if;
```


© 2021 All Rights Reserved.

```
     wait until falling_edge(clk);
    end loop;

    wait;                              -- forever
  end process stimulus_reader;

  monitor: process
  begin
    wait for 2000 ns;
    assert false report "Test completed" severity failure;
  end process monitor;

end;
```

dut.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

use STD.TEXTIO.ALL;

entity dut is
  port ( clk: in std_logic := '0';
         rst: in std_logic := '1';
           a: in bit := '0';
           b: in bit := '0';
           c: in bit := '0';
           d: in bit := '0';
           e: in bit := '0';
           f: in bit := '0';
       out1: out bit := '0'
      );
end;

architecture a of dut is
begin
  process (clk)
  begin
    if rising_edge(clk) then
      out1 <= a or b or c or d or e or f;
    end if;

  end process;

  process (rst)
  begin
    if (rst = '1') then
      report "Reset occurred." severity NOTE;
    end if;

  end process;

end;
```

prop.psl

```
vunit A1 (dut(a)) {
-- Sequential repetition:
-- psl default clock is rising_edge(clk);
property P1 is never {a[*2]; b;      c[*3]};
property P2 is always {a;a;b;a};
assert P1;
A2: assert P2;
}
```


© 2021 All Rights Reserved.

Following is the data file used for this example:

data

```
001001
110110
000010
100000
000000
100010
011000
001011
011100
```

Compiling VHDL Designs with vunit

To compile the VHDL design and testbench with assertions in vunit, perform the following steps:

1. Run the `vhan` command with the `-propfile` option to analyze the VHDL design and testbench, and the file containing the vunit.

   ```
   vhan -propfile prop.psl -assert dut.vhd test.vhd
   ```

   The `-profile prop.psl` option enables the reading of the assertions within the verification unit called *`A1`*, linked to design unit with the entity named *`dut`* and architecture named *`a`*. The `-assert` option is required if the DUT contains PSL assertion constructs.

2. Create `vhpsl.ff` with following the following lines:

   ```
   assert_status enable
   assert_cover enable
   assert_counter 32
   ```

3. Run the `ixcom` command to elaborate and compile the design.

   ```
   ixcom -ua -coverage functional -featureFile vhpsl.ff -top tb
   ```

   The above command compiles the design with the assertions.

4. Run the design in simulation acceleration mode using the `xeDebug` command:

   ```
   xeDebug -gui
   ```

Example 3: Using Embedded PSL Assertions (Verilog Only)

This example consists of Verilog files, `test.v` and `dut.v`, and demonstrates the use of PSL assertions embedded with the design. The PSL assertions are embedded in the `dut.v` file.

For designs with PSL assertions, compile the Verilog design with PSL Assertions.

© 2021 All Rights Reserved.

The `dut.v` and `test.v` files used in this example are given below.

dut.v

```
module dut (clk,rst,in1,in2,out1);
  input [3:0] in1;
  input [3:0] in2;
  input rst;
  input clk;
  output [4:0] out1;
  reg [2:0] state;

  reg [2:0] next_state;
  reg [4:0] result[7:0];
  reg [4:0] out1;
  integer idx;

  always @(negedge clk) begin
    if (rst) begin                // synch reset
      next_state = 3'b000;
      for (idx = 0; idx < 8; idx = idx + 1)
        result[idx] = 5'b00000;
      end

    else
      case (state)
      3'b000: begin
              next_state = 3'b001;
              result[next_state] = in1 + in2;
              end

      3'b001: begin
              next_state = 3'b010;
              result[next_state] = in1 - in2;
              end

      3'b010: begin
              next_state = 3'b100;
              result[next_state] = in1 - in2;
              end

      3'b011: begin
              next_state = 3'b101;
              result[next_state] = in1 + in2;
              end

      3'b100: begin
              next_state = 3'b011;
              result[next_state] = in1 + in2;
 end

      3'b101: begin
              next_state = 3'b110;
              result[next_state] = in1 - in2;
              end

      3'b110: begin
              next_state = 3'b111;
              result[next_state] = in1 + in2;
              end

      3'b111: begin
              next_state = 3'b000;
              result[next_state] = in1 + in2;
              end
```


© 2021 All Rights Reserved.

```
      endcase // case(state)
    out1 = result[state];
  end // always @ (negedge clk)

  always @(posedge clk) begin
  $display("State %b, next_state %b", state, next_state);
    state = next_state;
  end

// psl property A1 = always {next_state == state + 1 ||
//                              next_state == 3'b000} @(posedge clk);
//  psl P1: assert (A1);
endmodule
```

test.v

```
module test ();
  reg [3:0] in1;
  reg [3:0] in2;
  reg clk;
  reg reset;
  wire [4:0] out1;
  initial begin
    clk = 0;
    #20;
    reset = 1;
    #30;
    reset = 0;
    #500 $finish;
  end

  always begin
    #10 clk  = ~clk;
  end

  always begin
    #20;
    in1 = $random;
    in2 = $random;
  end

  dut inst1(clk, reset, in1, in2, out1);
  always @(posedge clk) begin
    $strobe("strobe: Reset = %b, In1 = %b, In2 = %b, Output = %b",
              reset, in1, in2, out1);

  end

endmodule
```

Compiling Verilog Designs with PSL Assertions

To compile the Verilog design and testbench with PSL assertions, perform the following steps:

1. Run the `vlan` command with the `-assert` option to analyze the Verilog design and testbench with PSL assertions.

   ```
   vlan -assert dut.v test.v
   ```

2. Create a feature file `psl_a.ff` with the following content:

© 2021 All Rights Reserved.

```
assert_cover enable
assert_counter 32
```

3. Specify the `-featureFile psl_a.ff` option with the `ixcom` command with the required options to elaborate and compile the Verilog design and the testbench.

```
ixcom -ua -top test -coverage functional -featureFile psl_a.ff
```

4. Run the `xeDebug` command to view the Assertion Coverage Summary and record the functional coverage data.

```
xeDebug -gui
```

Log Files Generated at Compilation Time

The following log files are generated at compilation time:

- `vlan.log`

  This file contains the output of the `vlan` command.

- `ixcom.log`:

  This file contains the output of the `ixcom` command.

- `xeDebug.log`:

  This file contains the output of the `xeDebug` command.

At run time, the error message pertaining to assertion failure are displayed in the following format:

```
xmsim: *E,ASRTST (dut.v,70): (time 10 NS) Assertion test.inst1.P1 has failed
```

In the above format, `test.inst1.P1` represents the path of the PSL assertion. `P1` is the label of the assertion directive and `test.inst1` is the path of the module from which the PSL assertion is fired. The filename in which the assertion directive is stored along with the line number is `dut.v` and `70`, respectively.

Note: The waveforms for checked and finished counts of the PSL assertion are dumped only if `assert_cover` is specified in the feature file that is passed to the `ixcom` command at the time of design elaboration. To dump and view the assertion status, specify `assert_status` is in the feature file.

Example 4: Using PSL Assertions in vunits (Verilog Only)

This example consists of Verilog files, `test.v` and `dut.v`, and demonstrates the use of PSL assertions in vunit. The PSL assertions are specified in vunit, which is the `prop.psl` file.

© 2021 All Rights Reserved.

For designs with PSL assertions included as vunit, compile the design in the same manner as given in Example 3: Using Embedded PSL Assertions (Verilog Only) on page 1250. The steps to compile the design with vunit are given in Compiling VHDL Designs with vunit on page 1250. The only difference between compilation of designs with embedded assertions and compilation of designs with vunits is that in the latter flow, the `vlan` command includes the additional `-propfile` option for compiling designs with vunits. The rest of the compilation and debugging processes remain the same for both flows.

The `test.v`, `dut.v`, and `prop.psl` files used in this example are given below.

dut.v

```
module dut (clk,rst,in1,in2,out1);
  input [3:0] in1;
  input [3:0] in2;
  input rst;
  input clk;
  output [4:0] out1;
  reg [2:0] state;
  reg [2:0] next_state;
  reg [4:0] result[7:0];
  reg [4:0] out1;
  integer idx;

  always @(negedge clk) begin
    if (rst) begin                // synch reset
      next_state = 3'b000;
      for (idx = 0; idx < 8; idx = idx + 1)
        result[idx] = 5'b00000;
      end

    else
      case (state)
      3'b000: begin
              next_state = 3'b001;
              result[next_state] = in1 + in2;
              end

      3'b001: begin
              next_state = 3'b010;
              result[next_state] = in1 - in2;
              end

      3'b010: begin
              next_state = 3'b100;
              result[next_state] = in1 - in2;
              end

      3'b011: begin
              next_state = 3'b101;
              result[next_state] = in1 + in2;
              end

      3'b100: begin
              next_state = 3'b011;
              result[next_state] = in1 + in2;
              end

      3'b101: begin
              next_state = 3'b110;
```


© 2021 All Rights Reserved.

```
              result[next_state] = in1 - in2;
              end
      3'b110: begin
              next_state = 3'b111;
              result[next_state] = in1 + in2;
              end
      3'b111: begin
              next_state = 3'b000;
              result[next_state] = in1 + in2;
              end
      endcase // case(state)
    out1 = result[state];
  end // always @ (negedge clk)
  always @(posedge clk) begin
  $display("State %b, next_state %b", state, next_state);
    state = next_state;
  end
// psl property A1 = always {next_state == state + 1 ||
//                              next_state == 3'b000} @(posedge clk);
//  psl P1: assert (A1);
endmodule
```

test.v

```
module test ();
  reg [3:0] in1;
  reg [3:0] in2;
  reg clk;
  reg reset;
  wire [4:0] out1;
  initial begin
    clk = 0;
    #20;
    reset = 1;
    #30;
    reset = 0;
    #500 $finish;
  end
  always begin
    #10 clk  = ~clk;
  end
  always begin
    #20;
   in1 = $random;
    in2 = $random;
  end
  dut inst1(clk, reset, in1, in2, out1);
  always @(posedge clk) begin
    $strobe("strobe: Reset = %b, In1 = %b, In2 = %b, Output = %b",
             reset, in1, in2, out1);
  end
endmodule
```

prop.psl

© 2021 All Rights Reserved.

```
vunit demo (dut) {
// psl default clock = negedge clk;
// psl property A1 = always {next_state == state + 1 ||  next_state == 3'b000};
// psl L1 : assert A1;
}
```

Compiling Verilog Designs with vunit

To compile the Verilog design and testbench with PSL assertions in vunit, perform the following steps:

1. Run the `vlan` command with the `-propfile` option to analyze the Verilog design file and testbench with PSL assertions in vunit:

   ```
   vlan dut.v test.v -propfile prop.psl -assert
   ```

2. Create `vhpsl.ff` with following the following lines:

   ```
   assert_status enable
   assert_cover enable
   assert_counter 32
   ```

3. Run the `ixcom` command with the required options to elaborate and compile the Verilog design and the testbench.

   ```
   ixcom -ua -top test -coverage functional -featureFile vhpsl.ff
   ```

4. Run the `xeDebug` command to view the Assertion Coverage Summary and record the functional coverage data.

   ```
   xeDebug -gui
   ```

Debugging Designs with PSL Assertions

To debug designs with PSL assertions, use SimVision. In this section, the design and testbench of Example 1: Using Embedded PSL Assertions (VHDL Only) on page 1244 and Example 3: Using Embedded PSL Assertions (Verilog Only) on page 1250 have been used.

Perform the following steps to use xeDebug for viewing PSL assertion information:

1. Specify the following command to invoke xeDebug:

   ```
   xeDebug -gui
   ```

2. Open the design, and start the session after specifying the required configuration settings. After you click the Start Session button, the icons in the toolbar at the top of the xeDebug GUI window get activated.

3. Click the icon on the toolbar to display the SimVision *Design Browser* window.

   The *Design Browser* window displays the hierarchy view of the testbench and the DUT. The left pane displays the design browser and the right pane displays various signals.

© 2021 All Rights Reserved.

Select the required signals including the assertion signals from the right pane of the *Design Browser* window, as shown in the following figure.

Figure 27-5 SimVision Design Browser

4. Select the required signals from the right pane of the *Design Browser* window.

5. Click the *Probe and Display* icon highlighted in Figure 27-5 from the *Design Browser* window to display the *Waveform Viewer* window illustrated in Figure 27-3.

6. Specify the following command to invoke ICC to view the coverage information in the ICC database generated for this design.

```
iccr -g
```

The *ICC - Coverage Totals* window is displayed, as shown in the following figure.

© 2021 All Rights Reserved.

Figure 27-6 ICC - Coverage Totals

7. Select the *Functional* tab, as shown in the following figure, to load the functional coverage database.

© 2021 All Rights Reserved.

Figure 27-7 ICC - Functional Tab

8. Select the *File – Open Test* menu to display the *Open Test* dialog, as shown in the following figure.

© 2021
All Rights Reserved.

Figure 27-8 Open cov_work Directory

9. Select the *cov_work* directory and click the *OK* button.

10. Select the subdirectory of the *cov_work* directory that has the coverage data. The contents of the select coverage data directory will be displayed within the *ICC - Coverage Totals*, as shown in the following figure.

© 2021 All Rights Reserved.

Figure 27-9 ICC - Coverage Totals with Data

11. Click the plus sign besides the *Control-oriented Coverage* option to display the functional coverage data, as shown in the following figure.

© 2021 All Rights Reserved.

Figure 27-10 ICC - Coverage Totals Window with Displayed Data

© 2021 All Rights Reserved.

Using Assertions in Dynamic Target Mode

In SA mode, the assertion failure counters are implemented on the software simulator to reduce hardware resource utilization. The assertion failure action blocks are also executed by the simulator. But, in the Dynamic Target mode, because design clocks never stop, the emulator cannot stop, and hence simulator is not available. So, assertion failure counters need to be implemented on the emulator side in the Dynamic Target mode, which is achieved by using the `uncontrolled_clock_domain` feature in the feature file and specify that file to the `ixcom` command with the `-featureFile` option. This enables the failure counters to be tracked on the emulator instead of the software simulator.

The failure action blocks, however, reside on the software simulator side only. The failure action blocks are not executed when the design is run in the Dynamic Target mode. Switch to the static target mode to run the failure action blocks. Then, appropriate messages are displayed for the failing assertions.

Important

If the `assert_cover` feature is not enabled, the fail counters are not generated. Thus, if you specify the `assert_counter_fail_hw` feature, but not the `assert_cover` feature, an error is generated and compilation is aborted.

Running Assertions in Dynamic Target Mode

1. Compile the design with a Feature File, `dyn_assert.ff` that has the following content:

   ```
   uncontrolled_clock_domain enable
   ixcom [ixcom_options] -featureFile dyn_assert.ff
   ```

2. Use the simulator to initialize the design.

   Note: Assertions can also be disabled at this stage, if required.

3. Swap to emulator and start the Dynamic Target mode.

   For example, for steps 2 and 3 described above, you can execute the following set of commands:

   ```
   debug .
   host .
   xc piok
   xc on -autorun
   configPM -dynamicTarget
   ```

   In the Dynamic Target mode, the predefined SDL signal `ASSERTION.FAILURE` enables triggering an action on any assertion failure. The trace clocks can be freezed to provide waveforms of assertion failure intervals.

© 2021 All Rights Reserved.

For example:

```
if (ASSERTION.FAILURE)
{
    Some code;
    trigger;
}
```

- The SDL can be written to trigger either on specific assertions or any assertion failure.
- At trigger time, you can list assertions that either failed or did not get covered.

To disable triggering of SDL at assertion failures, load a new SDL command file which does not contain the assertions-sensitive text.

4. Use the `assertion -summary -final` command to request the assertions summary report generation at any time when running in the Dynamic Target mode or at the time of session termination.

   Note: To generate an assertion summary report at session termination, you must first switch back to the static target mode.

   ```
   configPM -staticTarget
   ```

Support for Assertions Commands when Using Dynamic Targets

In the Dynamic Target mode, the simulator is available before the design is swapped to the emulator and after switching to the static target mode. Support for assertions in the Dynamic Target mode is limited to the features that do not require simulator processing.

For more information, see Assertions Commands Supported with Dynamic Targets and Assertions Commands not Supported with Dynamic Targets.

Assertions Commands Supported with Dynamic Targets

The following assertions commands are supported in the Dynamic Target mode although the output from the `assertion -counter` command is from the time that the counter value on the emulator was last accessed.

- `assertion -check`
- `assertion -counter`
- `assertion -list`
- `assertion -summary -final`

© 2021 All Rights Reserved.

The use of `assertion -summary -final` causes generation of a summary report at session termination.

The `assertion -list` options are supported to enable examining a subset of design assertions. Omitting an assertion locator defaults to selecting all the assertions.

The `assertion -list` command providesthe assertion information while the design clocks are running. This information is gathered by examining the assertion counter signals on the emulator. Following are the relevant command options:

- `assertion -list -failed`

  Returns a cumulative list of all the assertions that have failed since time `0`.

- `assertion -list -uncovered`

  Returns a list of assertions that are still uncovered at the time of the emulator state access.

- `assert -list -failed -uncovered`

  Returns a set of assertions that have either failed since time `0` or are still uncovered at the time of the emulator state access.

The output of the above commands is returned as a Tcl result list. These command options are supported in all run modes.

Important

The design clocks continue to run while processing the `assertion -list` command; therefore, the assertion counters can change before the results are reported. The output always presents the counter values at the time they were examined and not any particular simulation time.

Assertions Commands not Supported with Dynamic Targets

The following assertions commands are not supported in the Dynamic Target mode, and are ignored with a warning. These can be used before the design is swapped to the emulator and at session termination, although some of these commands might not be relevant. For example, the `-off` and `-on` options are not relevant at session termination, when no simulation time is available; the disable or enable actions for assertions, therefore, have no effect.

- `assertion -get`
- `assertion -off`

© 2021 All Rights Reserved.

- `assertion -on`
- `assertion -summary` without the `-final` switch

© 2021 All Rights Reserved.

Support for Specific SVA Constructs

The following tables show the current implementation status of specific SVA Constructs.

- Table 27-3, “Support for SVA Constructs - LRM 1800 - 2005,” on page 1267
- Table 27-4, “Support for SVA Constructs - LRM 1800 - 2009,” on page 1277
- Table 27-5, “Support for SVA Constructs - LRM 1800 - 2017,” on page 1277

Table 27-3 Support for SVA Constructs - LRM 1800 - 2005

SVA Construct	LRM 1800-2005 Reference	Supported	Notes
Immediate Assertion	17.2	yes	Not supported in `always_comb`,`always_latch`, `always@(*)` processes.
`cycle_delay_range sequence_expr`	17.5	yes	Only elaboration time constants are allowed in the cycle delay range expression. The cycle delay range that exceeds 32 and appears in one of the following expressions is not supported: ■ Assertion instantiation ■ Sequences on which methods or `first_match` is called ■ Multiclocked sequence and properties

© 2021 All Rights Reserved.

Table 27-3 Support for SVA Constructs - LRM 1800 - 2005

SVA Construct	LRM 1800-2005 Reference	Supported	Notes
`$` (end of simulation expression)	17.5	yes	
sequence declaration	17.6	yes	
Typed formal argument in sequences	17.6.1	yes	
Sequence repetition: Consecutive	17.7.2	yes	
Sequence repetition: Goto	17.7.2	yes	
Sequence repetition: non-consecutive	17.7.2	yes	
Sampled value functions	17.7.3	yes	Supported in the context of assertions as well as procedural code. Additionally, these functions can be used as part of the actual argument expressions in a module instantiation statement.
`$sampled`	17.7.3	yes	Optional clock must be the same as the assertion clock.
`$sampled` in `action` block	17.7.3	yes	Optional clock must be the same as the assertion clock.
`$rose`	17.7.3	yes	
`$past`	17.7.3	yes	

© 2021 All Rights Reserved.

Table 27-3 Support for SVA Constructs - LRM 1800 - 2005

SVA Construct	LRM 1800-2005 Reference	Supported	Notes
`$fell`	17.7.3	yes	
`$stable`	17.7.3	yes	
Sequence operator: `AND`	17.7.4	yes	
Sequence operator: `INTERSECT`	17.7.5	yes	
Sequence operator: `OR`	17.7.6	yes	
Sequence operator: `first_match`	17.7.7	yes	
Sequence operator: `throughout`	17.7.8	yes	
Sequence operator: `within`	17.7.9	yes	
Clocking event in sequence expression	17.5	yes	
Detecting sequence endpoint (`.ended`)	17.7.10	yes	
Detecting sequence endpoint in multiclock context (`.triggered`, `.matched`)	17.12.5	yes	`.triggered` must have the same clock as the assertion clock. The triggered method is supported in sequence expressions as per the 1800-2009 LRM.
Sequence match item	17.8	yes	

© 2021 All Rights Reserved.

Table 27-3 Support for SVA Constructs - LRM 1800 - 2005

SVA Construct	LRM 1800-2005 Reference	Supported	Notes
Sequence: local variables	17.8	yes	`first_match` is not supported with local variables. Sequence methods cannot modify local variables. No support for local variables: ■ Formal arguments ■ Multiclock properties ■ Multiclock support
Local variables of complex data types	17.8	yes	
Calling subroutine on match of a sequence	17.9	yes	
System function: `$onehot`	17.10	yes	
System function: `$onehot0`	17.10	yes	
System function: `$isunknown`	17.10	yes	In the hardware mode, it evaluates to 0. In the software mode, it follows the LRM.
System function: `$countones`	17.10	yes	
Property declaration	17.11	yes	
Property operator: `not`	17.11	yes	
Property operator: `and`	17.11	no	

© 2021 All Rights Reserved.

Table 27-3 Support for SVA Constructs - LRM 1800 - 2005

SVA Construct	LRM 1800-2005 Reference	Supported	Notes
Property operator: `or`	17.11	no	The Property `OR` operator is replaced with the Sequence `OR` operator if the following conditions are met: ■ The Property `OR` is used with cover directives ■ The Property operands of the `OR` operator are actually sequence expressions.
Property operator: `if-else`	17.11	yes	
Property operator: `|->`	17.11	yes	
Property operator: `|=>`	17.11	yes	
Property: `disable iff`	17.11	yes	
Typed formal argument in property	17.11.1	yes	
Recursive properties	17.11.4	no	

© 2021 All Rights Reserved.

Table 27-3 Support for SVA Constructs - LRM 1800 - 2005

SVA Construct	LRM 1800-2005 Reference	Supported	Notes
Multiclock support	17.12	yes	■ Cannot change clock within `first_match` ■ No support for local variables ■ multiclock sequence methods are supported
`assert` statement (concurrent assertion)	17.13.1	yes	
`assume` statement	17.13.2	yes	
`cover` statement	17.13.3	yes	
Concurrent assertion in procedural code	17.13.5	yes	Assertion controls and counters are supported for procedural assertions. Concurrent assertions are not supported in `always_comb`,`always_latch`, `always@(*)` processes.
Default clocking	17.14	yes	

© 2021 All Rights Reserved.

Table 27-3 Support for SVA Constructs - LRM 1800 - 2005

SVA Construct	LRM 1800-2005 Reference	Supported	Notes
Multiclock properties	17.14.1	yes	■ Cannot change clock within `first_match` ■ No support for local variables ■ Multi-clocked sequence methods are supported
Clock resolution	17.14.1	yes	
Binding properties to scope or instances	17.15	yes	
`expect` statement	17.16	yes	

© 2021 All Rights Reserved.

Table 27-3 Support for SVA Constructs - LRM 1800 - 2005

SVA Construct	LRM 1800-2005 Reference	Supported	Notes
`$asserton`	22.8	yes	Supported if used in one of the following manners: ■ These can exist in any part of a testbench module. ■ These can exist within the `initial` block of a DUT module. ■ These can exist within the tasks called using `ixc tbcall_region` in a DUT module. ■ These can exist within the `fail action` block of the concurrent assertions.

© 2021 All Rights Reserved.

Table 27-3 Support for SVA Constructs - LRM 1800 - 2005

SVA Construct	LRM 1800-2005 Reference	Supported	Notes
`$assertkill`	22.8	yes	Supported if used in one of the following manners: ■ These can exist in any part of a testbench module. ■ These can exist within the `initial` block of a DUT module. ■ These can exist within the tasks called using `ixc tbcall_region` in a DUT module. ■ These can exist within the `fail action` block of the concurrent assertions.

© 2021 All Rights Reserved.

Table 27-3 Support for SVA Constructs - LRM 1800 - 2005

SVA Construct	LRM 1800-2005 Reference	Supported	Notes
`$assertoff`	22.8	yes	Supported if used in one of the following manners: ■ These can exist in any part of a testbench module. ■ These can exist within the `initial` block of a DUT module. ■ These can exist within the tasks called using `ixc tbcall_region` in a DUT module. ■ These can exist within the `fail action` block of the concurrent assertions.

© 2021 All Rights Reserved.

The following table lists support for some of the SVA constructs listed in the LRM 1800-2009.

Table 27-4 Support for SVA Constructs - LRM 1800 - 2009

SVA Construct	LRM 1800-2009 Reference	Supported	Notes
Repetition operators [*], [+]	16.7	yes	Consecutive repetition operators: ■ [*] is an equivalent representation of [*0:$] ■ [+] is an equivalent representation of [*1:$]
delay operators ##[+], ##[*]	16.9.2	yes	Cycle delay ranges: ■ ##[*] is used as an equivalent representation of ##[0:$] ■ ##[+] is used as an equivalent representation of ##[1:$]
default disable iff	16.13	yes	
$changed	16.15.8	yes	

The following table lists support for some of the SVA constructs listed in the LRM 1800-2017.

Table 27-5 Support for SVA Constructs - LRM 1800 - 2017

SVA Construct	LRM 1800-2017 Reference	Supported	Notes
`$inferred_clock`	16.14.7	yes	Returns the expression of the inferred clocking event.

© 2021 All Rights Reserved.

Table 27-5 Support for SVA Constructs - LRM 1800 - 2017

SVA Construct	LRM 1800-2017 Reference	Supported	Notes
`$inferred_disable`	16.14.7	yes	Returns the inferred disable expression

© 2021 All Rights Reserved.

Support for PSL Constructs

The following sections outline:

- Default Clock Implementation
- Extended Support for Built-in Methods (VHDL Only)
- Support for Specific PSL Constructs in a Mixed-HDL Design
- Support for Specific PSL Constructs in a Verilog Design

For identifying PSL assertions support in this release, refer to the PSL 1850-2005 LRM.

Default Clock Implementation

Default clock declarations can be done in both design and verification units. However, there are certain aspects that need to be considered while implementing default clock declarations in design and verification units.

- Default Clock Declaration in a Design Unit
- Default Clock Declarations in a Verification Unit

Default Clock Declaration in a Design Unit

A design unit can have multiple default clock declarations. However, the clock declaration placed nearest to the assertion instance and appears before it within the same scope is considered the default clock declaration for that assertion. Following are the various scenarios for declaring and handling a default clock in a design unit:

- If an assertion instance contains an explicit clock declaration, then the default clock declaration is not applicable to this assertion instance.
- If there are multiple clock declarations before an assertion instance, then the nearest default clock declaration in the design is applied to this assertion instance.
- If the default clock declaration applicable to an assertion instance is unsupported, then the assertion instance is ignored and a warning message is displayed.
- If the current scope does not contain any default clock declarations before the assertion is instantiated, then the assertion is considered to be unclocked.
- If an assertion is instantiated in a sub-scope (generates or named block [for Verilog-only]) and does not have a default clock declaration before it in the sub-scope, then the parent scope is used to find the applicable default clock declaration.

© 2021 All Rights Reserved.

- If the parent scope contains a default clock declaration nearest to the sub-scope and appears before it, then this default clock declaration becomes the default clock for the assertion in the sub-scope.
- If the parent scope does not contain any default clock declaration before the sub-scope, then the default clock declaration is located in the design hierarchy. The default clock declaration in the design hierarchy nearest to the sub-scope becomes the default clock for the assertion in that sub-scope.

  The search for a default clock declaration is terminated in the following two cases:

  - If the default clock declaration is located while moving up in the design hierarchy.
  - The parent architecture (in a mixed-HDL or VHDL design) or module (in a Verilog-only) is hit while moving up in the design hierarchy. In this case, if the parent architecture or module is hit and no default clock declaration is found, then the assertion in the sub-scope is considered to be unclocked.

- For a mixed-HDL or a VHDL Design: If an entity contains a default clock declaration, then the last default clock declaration in an entity becomes the first default clock declaration for the architecture.
- For a mixed-HDL or a VHDL Design: If an entity contains assertions but no default clock declarations, then the assertions in the entity are considered to be unclocked. Default clock declarations are not derived from any architecture to the entity.

Note: All these above considerations are applicable to assertions in an entity. This behavior of declaring and handling default clocks in IXCOM is consistent with the behavior of Xcelium.

Default Clock Declarations in a Verification Unit

In IXCOM, the behavior of the default clock in a verification unit is compliant with PSL LRM 1850-2005. The following are various scenarios for declaring and handling a default clock in a verification unit:

- If there are multiple default clock declarations in the verification unit, it is considered as an error.
- If there is only one default clock declaration in the verification unit, it is applicable to all assertion instances within that verification unit.
- If there is no default clock declaration in a verification unit, the default clock declaration from the inherited verification unit is applicable to all the assertion instances in the current verification unit. However, if multiple clock declarations are derived from the

© 2021 All Rights Reserved.

inherited verification units, it results in an error and the assertions in the current verification unit are ignored with a warning.

- If a default clock is not declared in a verification unit and is also not derived from inherited verification units, the default clock declaration in the default verification unit (default `vmode`) is applied to the assertions within the current verification unit. However, if there are multiple default clock declarations in a default verification unit, then this results in an error.
- If there is no default clock declaration in the current verification unit and if the default clock has not been derived from any inherited verification unit, the assertions in the verification unit are considered to be unclocked.

Note: Xcelium displays a different behavior for default clock declarations in a verification unit.

Extended Support for Built-in Methods (VHDL Only)

The built-in methods, `prev`, `rose`, `fell`, and `stable` are supported in unclocked expressions if these methods are used as boolean terms and are not used in HDL expressions.

But, if these built-in methods appear in HDL expressions of an unclocked expression, then the unclocked assertion is ignored with a warning message. This is illustrated in the following examples.

```
-- psl property always (a -> b or prev (c))
```

This assertion is ignored because `-prev` is used in the HDL expression.

```
-- psl property always (a -> prev (c))
```

This assertion is supported.

The clock expression argument to built-in methods and endpoint is supported if the clock is same as that of the assertion in context. If the assertion is clocked and the built-in methods and endpoints used in the assertion have their own clock expression argument, then both the clocks need to be same. However, if the clock of built-in methods is different from the clock of the assertion in context, then the assertion is ignored with a warning message. In addition, if the built-in methods have clock expression argument but is used in an unclocked assertion, then the assertion is ignored with a warning message.

Differences between Xcelium and IXCOM for PSL Assertions

Following are the main differences between the handling of PSL constructs in Xcelium and IXCOM:

© 2021 All Rights Reserved.

- PSL within entity and architecture declarative regions is not supported in IXCOM. PSL within a process statement is not allowed in IXCOM. Xcelium supports PSL in declarative regions.
- Xcelium does not increment the failure counts for the PSL failing at end of the simulation, for example, strong properties, while IXCOM does that. Therefore, the failure counts for strong properties are different in Xcelium and IXCOM.
- When an assertion fails and finishes simultaneously, Xcelium only increments the failed count and not the finished count, while IXCOM increments both.
- Xcelium ignores the `assume_guarantee` verification directive, whereas IXCOM treats it same as the `assert` directive.

Refer to the *PSL Quick Reference Guide* in the Xcelium documentation set for details of the constructs discussed in this section.

Support for Specific PSL Constructs in a Mixed-HDL Design

The following table shows the current status of support for specific PSL Constructs.

Table 27-6 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
Boolean Layer			
Bit expressions	5.1.1	yes	
Boolean expressions including `true`	5.1.2	yes	
BitVector expressions	5.1.3	yes	
Numeric expressions	5.1.4	yes	
String expressions	5.1.5	yes	

© 2021 All Rights Reserved.

Table 27-6 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
HDL expressions	5.2.1	yes	***Limitations (Verilog-only):*** ■ HDL signals with names that are PSL keywords, such as Clock, A, or E are not supported within PSL statements. ■ Concatenation expressions with single term is not supported.
PSL expressions	5.2.2	limited	Support as per simple subset. Use of `->` and `<->` in the context of boolean expression is not supported. For example, `always {a->b; c->d}`.
Built-in Functions			

© 2021 All Rights Reserved.

Table 27-6 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
`prev(Any_type[,number[, clock_expr]])`	5.2.3.1	yes	**Limitations (VHDL only):** Optional clock expression is supported if it is the same as the clock of the assertion in context. Limitations (Verilog only): ■ Only integral types are supported. ■ Optional clock should be same as the clock of the sequence/ property expression in context.
`next(Any_type)`	5.2.3.2	no	

© 2021 All Rights Reserved.

Table 27-6 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
`stable(Any_type[,clock_expr])`	5.2.3.3	yes	Limitations (VHDL only): ■ Optional clock is supported if it is same as the clock of the assertion in context. ■ Only variable reference, record select, bit select and part-select expressions are supported as first arguments. **Limitations (Verilog-only):** ■ Only integral types are supported. ■ Optional clock should be same as the clock of the sequence/ property expression in context.

© 2021 All Rights Reserved.

Table 27-6 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
`rose(bit[,clock_expr])`	5.2.3.4	yes	Limitations (VHDL only): ■ Optional clock is supported if it is same as the clock of the assertion in context. ■ Only variable reference, record select, bit select and part-select expressions are supported as first arguments. **Limitations (Verilog-only):** ■ Only integral types are supported. ■ Optional clock should be same as the clock of the sequence/ property expression in context.

© 2021 All Rights Reserved.

Table 27-6 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
`fell(bit[,clock_expr])`	5.2.3.5	yes	Limitations (VHDL only): ■ Optional clock is supported if it is same as the clock of the assertion in context. ■ Only variable reference, record select, bit select and part-select expressions are supported as first arguments. **Limitations (Verilog-only):** ■ Only integral types are supported. ■ Optional clock should be same as the clock of the sequence/ property expression in context.

© 2021 All Rights Reserved.

Table 27-6 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
`ended(sequence[,clock_expr])`	5.2.3.6	yes	Limitation (VHDL-only): Optional clock is supported if it is same as the clock of the assertion in context. It is not supported if used in HDL expressions and unclocked properties. Limitation (Verilog-only): Optional clock argument is supported only if it is the same as the clock of enclosing sequence/ property.
`isunknown(BitVector)`	5.2.3.7	yes	Returns `false` for assertions running on the emulator.
`countones(BitVector)`	5.2.3.8	yes	**Limitation (Verilog-only):** Only integral types are supported.
`nondet(Value_List)`	5.2.3.9	no	
`nondet_vector(Number,Value_List)`	5.2.3.10	no	
`onehot(BitVector),onehot0(BitVector)`	5.2.3.11	yes	**Limitation (Verilog-only):** Only integral types are supported.

© 2021 All Rights Reserved.

Table 27-6 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
named endpoint (single cycle)	N/S	yes	Limitation (VHDL-only): Optional clock is supported if it is same as the clock of the assertion in context. It is not supported if used in HDL expressions and unclocked properties. Limitation (Verilog-only): Optional clock is supported only if it is the same as the clock of enclosing sequence/ property.
named endpoint (full temporal multi-cycle)		yes	Limitation (VHDL-only): Optional clock is supported if it is same as the clock of the assertion in context. It is not supported if used in HDL expressions and unclocked properties. Limitation (Verilog-only): Optional clock is supported only if it is the same as the clock of enclosing sequence/ property.
union	5.2.4	no	
clock expression	5.3	yes	

© 2021 All Rights Reserved.

Table 27-6 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
Bit and std_logic types (VHDL)		yes	
Multiple sensitivity		no	Not supported for VHDL
Edge sensitive clock		yes	
`rising_edge()`, `falling_edge()` (VHDL)		yes	
default clock	5.4	yes	If there are multiple clock declarations within a scope, IXCOM accepts the textually closest declaration. **Limitation (VHDL-only):** Only limited expression types are supported as clock expressions. For example, level-sensitive clock expressions are not supported. For more information on default clock behavior, refer to Default Clock Implementation on page 1279.
multiple clock support	4.3.2.3	no	Not supported for VHDL

© 2021 All Rights Reserved.

Table 27-6 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
Unclocked assertions	4.4.1	limited	**Limitation (VHDL-only):** Properties that contain built-in PSL functions, such as `ended`, `prev`, `rose`, `fell`, and `stable`, with clock arguments are not supported. The built-in PSL functions, `prev`, `rose`, `fell`, and `stable` are supported in unclocked assertions only if they are used as Boolean terms and do not appear in HDL expressions.
Temporal Layer			
SERE concatenation (SERE; SERE)	6.1.1.1.1	yes	
SERE sequence fusion (SERE: SERE)	6.1.1.1.2	yes	
SERE or (SERE | SERE)	6.1.1.2.1	yes	
SERE and (non-lgth match) (SERE `&` SERE)	6.1.1.2.2	yes	
SERE and (lgth match) (SERE `&&` SERE)	6.1.1.2.3	yes	
SERE `within`	6.1.1.2.4	yes	
Parameterized sequence (`for..`)	6.1.1.2.5	yes	

© 2021 All Rights Reserved.

Table 27-6 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
SERE consecutive repeat [*],[+]	6.1.2.1	yes	Xcelium does not support function operator in repeat count expression, such as `*1` to `MAX+1`.
SERE non-consecutive repeat [=]	6.1.2.2	yes	
SERE goto repeat [->]	6.1.2.3	yes	
Braced SERE	6.1.2.4	yes	
Clocked SERE	6.1.2.5	yes	**Limitation (VHDL-only):**This is not supported if it is multi-clocked.
FL Property Bool | Property	6.2.1	yes	
Sequential FL Property Sequence!	6.2.1.1	yes	
Clocked FL Property FL_prop@clk_expr	6.2.1.2	yes	
FL `always`	6.2.1.3.1	yes	
FL `never`	6.2.1.3.2	yes	Supported as per simple subset with argument being boolean or sequence.
FL `eventually!`	6.2.1.3.3	yes	
FL `next [n]`	6.2.1.3.4	yes	

© 2021 All Rights Reserved.

Table 27-6 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
FL `next! [n]`	6.2.1.3.4	no	
FL `next_a [finite_range]`	6.2.1.4.1	yes	
FL `next_a! [finite_range]`	6.2.1.4.1	no	
FL `next_e [finite_range]`	6.2.1.4.2	yes	The argument property has to be a boolean.
FL `next_e! [finite_range]`	6.2.1.4.2	no	
FL `next_event[!] [n]`	6.2.1.4.3	no	
FL `next_event_a[!] [n]`	6.2.1.4.4	no	
FL `next_event_e[!] [n]`	6.2.1.4.5	no	
FL `abort` (implemented as `async`)	6.2.1.5.1	yes	
FL `sync_abort`, FL `async_abort`	6.2.1.5.1	yes	
FL `before`, `FL_before_`	6.2.1.5.2	yes	Both arguments should be boolean.
FL `before! FL_before!_`	6.2.1.5.2	yes	Both arguments should be boolean.
FL `until FL_until_`	6.2.1.5.3	yes	RHS has to be a boolean.
FL `until! FL_until!_`	6.2.1.5.3	no	
Sequence implication (`|->`, `|=>`)	6.2.1.6.1	yes	

© 2021 All Rights Reserved.

Table 27-6 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
Parameterized property (`for..`)	6.2.1.7.1	no	
Logical implication (`->`)	6.2.1.7.2	yes	LHS should be a boolean. For example, always `(a->b)->c` is not supported since LHS is a property.
Logical `iff` (`<->`)	6.2.1.7.3	no	
`Boolean AND_OP FL_Property`	6.2.1.7.4	no	Supported by Xcelium.
`Boolean OR_OP FL_Property`	6.2.1.7.5	no	Supported by Xcelium.
`NOT_OP Boolean`	6.2.1.7.6	yes	
LTL operators	6.2.1.8	no	Not supported by Xcelium.
OBE operators	6.2.2	no	
Replicated properties (`forall`)	6.2.3	no	Supported by Xcelium.
PSL Formal parameter types: `const`, `boolean`, `sequence`, `property`	6.3.1.1	yes	
PSL Formal parameter types: `bit`, `bitvector`, `numeric`, `string`	6.3.1.1	yes	
HDL Formal types	6.3.1.2	yes	
Sequence Declaration	6.3.2.1	yes	
Property Declaration	6.3.2.2	yes	

© 2021 All Rights Reserved.

Table 27-6 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
Verification Layer			
support of `labels`	7.1	yes	
support of `report`	7.1	yes	
support of `severity`	N/S	yes	
`assert property [report] [severity]`	7.1.1	yes	
`assume property [report] [severity]`	7.1.2	yes	`assume` is mapped to `assert`, `report` and `severity` is supported in `assume` directive
`assume_guarantee property [report][severity]`	7.1.3	no	Ignored
`restrict sequence [report]`	7.1.4	yes	
`restrict_guarantee sequence [report]`	7.1.5	no	Ignored.
`cover sequence [report]`	7.1.6	yes	
`fairness`	7.1.7	no	Ignored.
`strong fairness`	7.1.7	no	Ignored.
`'vmode'` &`'vprop'`	7.2	yes	
vunit binding (module based)	7.2.1	yes	
vunit binding (to instances)		yes	Not supported for VHDL

© 2021 All Rights Reserved.

Table 27-6 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
vunit inheritance	7.2.2	yes	
vunit scoping rules (cross language binding)	7.2.3	no	
properties in `for-generate` statements	N/S	yes	
properties in `if-case-generate` statements	N/S	yes	
support opt clk expr in built-in functions	5.2.3	yes	Supported if the clock expression is same as that of the assertion.
use of `ended`/`endpoints` in boolean expressions	5.2.3.6	yes	Limitation (VHDL-only): `ended/endpoints` are not supported if these occur within an HDL expression. For example: `a and ended (b)`.
instantiation of props in other props	6.2.1	yes	
parameterized declaration of seq & props	6.3.2.1, 6.3.2.2	yes	
HDL in a vunit	7.2	yes	
pragma support in vunits (protection pragma,..)	7.2	no	
`` `define `` & `` `ifdef `` in external files	7.2	yes	

© 2021 All Rights Reserved.

Table 27-6 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
`%if` and `%for` PSL macro	4.2.5	yes	Support restricted to vunits only
vunits should not be able to see others	7.2.1	yes	
binding to vunit instances	7.2.1	no	
inheritance, vmode, vprop	7.2.1	yes	
Modeling layer	8	limited	Certain features, such as re-declaration of object with the same name are not allowed.
support of complex report args	N/S	yes	
properties in VHDL packages	N/S	yes	
`endpoints/ended` in unclocked properties		no	
clocked `endpoints/ended`		yes	Supported only if the clock expression of the endpoint is same as that of the entire property.
`endpoints/ended` on RHS of `sync_abort/async_abort`		no	
complex first argument of prev	N/S	yes	
Assertions with large repeat count	N/S	no	Not supported, by default, but by providing high value for `+assertMaxIter`

For the purpose of this table:

© 2021 All Rights Reserved.

N/S or Non-Std means Not Specified or included as part of the IEEE 1850 Standard.

Support for Specific PSL Constructs in a Verilog Design

The following table shows the current status of support for specific PSL Constructs in a Verilog design.

Table 27-7 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
Boolean Layer			
Bit expressions	5.1.1	yes	
Boolean expressions including "`true`" (1'b1)	5.1.2	yes	
BitVector expressions	5.1.3	yes	
Numeric expressions	5.1.4	yes	
String expressions	5.1.5	yes	
HDL expressions	5.2.1	yes	■ HDL signals with names that are PSL keywords, such as Clock, A, or E are not supported within PSL statements. ■ Concatenation expressions with single term is not supported.

© 2021 All Rights Reserved.

Table 27-7 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
PSL expressions	5.2.2	limited	Support as per simple subset. Use of `->` and `<->` in the context of boolean expression is not supported. For example, `always {a->b; c->d}`.
Built-in Functions			
`prev(Any_type[,number[,clock_expr]])`	5.2.3.1	yes	■ Only integral types are supported. ■ Optional clock should be same as the clock of the sequence/property expression in context.
`next(Any_type)`	5.2.3.2	no	
`stable(Any_type[,clock_expr])`	5.2.3.3	yes	■ Optional clock is supported if it is same as the clock of the sequence/property expression in context. ■ Only integral types are supported.
`rose(bit[,clock_expr])`	5.2.3.4	yes	■ Optional clock is supported if it is same as the clock of the sequence/property expression in context. ■ Only integral types are supported.

© 2021 All Rights Reserved.

Table 27-7 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
`fell(bit[,clock_expr])`	5.2.3.5	yes	■ Optional clock is supported if it is same as the clock of the sequence/property expression in context. ■ Only integral types are supported.
`isunknown(BitVector)`	5.2.3.7	yes	Returns false if running on the emulator.
`ended(sequence[,clock_expr])`	5.2.3.6	yes	■ Optional clock is supported if the clock is same as the clock of the enclosing sequence/property in context. In HDL expressions, `ended` is treated as SVA ended. ■ `endpoints`/`ended` on RHS of `sync_abort`/ `async_abort` are not supported.
`countones(BitVector)`	5.2.3.8	yes	Only integral types are supported.
`nondet(Value_List)`	5.2.3.9	no	
`nondet_vector(Number,Value_List)`	5.2.3.10	no	
`onehot(BitVector)`, `onehot0(BitVector)`	5.2.3.11	yes	Only integral types are supported.

© 2021 All Rights Reserved.

Table 27-7 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
named endpoint (single cycle)	N/S	yes	Optional clock is supported if the clock is same as the clock of the enclosing sequence/ property.
named endpoint (full temporal multi-cycle)		yes	Optional clock is supported if the clock is same as the clock of the enclosing sequence/ property.
union	5.2.3.12	no	
clock expression	5.3	yes	
Built-in functions rose, fell		yes	
Multiple sensitivity		yes	
Edge sensitive clock		yes	
HDL_CLOCK_EXPR	4.3.2.3	yes	
default clock	5.4	yes	If there are multiple clock declarations within a scope, IXCOM/Xcelium accepts the textually closest declaration.
multiple clock support	4.2.3.2.2	yes	
Unclocked assertions	4.4.1	yes	
Temporal Layer			

© 2021 All Rights Reserved.

Table 27-7 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
SERE concatenation (SERE; SERE)	6.1.1.1.1	yes	
SERE sequence fusion (SERE: SERE)	6.1.1.1.2	yes	
SERE or (SERE | SERE)	6.1.1.2.1	yes	
SERE and (non-lgth match) (SERE `&` SERE)	6.1.1.2.2	yes	
SERE and (lgth match) (SERE `&&` SERE)	6.1.1.2.3	yes	
SERE `within`	6.1.1.2.4	yes	
Parameterized sequence (`for..`)	6.1.1.2.5	yes	
SERE consecutive repeat [`*`],[`+`]	6.1.2.1	yes	Xcelium does not support function operator in repeat count expression, such as `*1` to `MAX+1`.
SERE non-consecutive repeat [`=`]	6.1.2.2	yes	
SERE goto repeat [`->`]	6.1.2.3	yes	
Braced SERE	6.1.2.4	yes	
Clocked SERE	6.1.2.5	yes	
FL Property Bool | Property	6.2.1	yes	
Sequential FL Property Sequence!	6.2.1.1	yes	

© 2021 All Rights Reserved.

Table 27-7 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
Clocked FL Property FL_prop@clk_expr	6.2.1.2	yes	
FL `always`	6.2.1.3.1	yes	
FL `never`	6.2.1.3.2	yes	Supported as per simple subset with argument being boolean or sequence.
FL `eventually!`	6.2.1.3.3	yes	
FL `next [n]`	6.2.1.3.4	yes	
FL `next! [n]`	6.2.1.3.4	no	
FL `next_a [finite_range]`	6.2.1.4.1	yes	
FL `next_a! [finite_range]`	6.2.1.4.1	no	
FL `next_e [finite_range]`	6.2.1.4.2	yes	The argument property has to be a boolean.
FL `next_e! [finite_range]`	6.2.1.4.2	no	
FL `next_event[!] [n]`	6.2.1.4.3	no	
FL `next_event_a[!] [n]`	6.2.1.4.4	no	
FL `next_event_e[!] [n]`	6.2.1.4.5	no	
FL `abort` (implemented as `async`)	6.2.1.5.1	yes	

© 2021 All Rights Reserved.

Table 27-7 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
FL `sync_abort`, FL `async_abort`	6.2.1.5.1	yes	
FL `before`, `FL_before_`	6.2.1.5.2	yes	Both arguments should be boolean.
FL `before! FL_before!_`	6.2.1.5.2	yes	Both arguments should be boolean.
FL `until FL_until_`	6.2.1.5.3	yes	RHS has to be a boolean.
FL `until! FL_until!_`	6.2.1.5.3	no	
Sequence implication (`|->`, `|=>`)	6.2.1.6.1	yes	
Parameterized property (`for..`)	6.2.1.7.1	no	
Logical implication (`->`)	6.2.1.7.2	yes	LHS should be a boolean. For example, always `(a->b)->c` is not supported since LHS is a property.
Logical `iff` (`<->`)	6.2.1.7.3	no	
`Boolean AND_OP FL_Property`	6.2.1.7.4	no	Supported by Xcelium.
`Boolean OR_OP FL_Property`	6.2.1.7.5	no	Supported by Xcelium.
`NOT_OP Boolean`	6.2.1.7.6	yes	
LTL operators	6.2.1.8	yes	Not supported by Xcelium.
OBE operators	6.2.2	no	

© 2021 All Rights Reserved.

Table 27-7 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
Replicated properties (`forall`)	6.2.3	no	Supported by Xcelium.
PSL Formal parameter types: `const`, `boolean`, `sequence`, `property`	6.3.1.1	yes	
PSL Formal parameter types: `bit`, `bitvector`, `numeric`, `string`	6.3.1.1	yes	
HDL Formal types	6.3.1.2	yes	
Sequence Declaration	6.3.2.1	yes	
Property Declaration	6.3.2.2	yes	
Verification Layer			
support of `labels`	7.1	yes	
support of `report`	7.1	yes	
support of `severity`	N/S	yes	
`assert property [report] [severity]`	7.1.1	yes	
`assume property [report][severity]`	7.1.2	yes	`assume` is mapped to `assert`; `report` and `severity` is supported in `assume` directive.
`assume_guarantee property [report][severity]`	7.1.3	no	Ignored

© 2021 All Rights Reserved.

Table 27-7 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
`restrict sequence [report]`	7.1.4	yes	
`restrict_guarantee sequence`	7.1.5	no	Ignored.
`cover sequence [report]`	7.1.6	yes	
`fairness`	7.1.7	no	Ignored.
`strong fairness`	7.1.7	no	Ignored.
`'vmode'` &`'vprop'`	7.2	yes	
vunit binding (module-based)	7.2.1	yes	
vunit binding (to instances)	7.2.1	yes	
vunit inheritance	7.2.2	yes	
vunit scoping rules (cross language binding)	7.2.3	no	
properties in `for-generate` statements	N/S	yes	
properties in `if-case-generate` statements	N/S	yes	
pragma support in vunits (protection pragma,..)	7.2	no	
`%if` and `% for` PSL macro	4.2.5	yes	Support restricted to vunits only.

© 2021 All Rights Reserved.

Table 27-7 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
support opt clk expr in built-in functions	5.2.3	yes	Supported if the clock expression is same as that of the property in context.
use of `ended` in boolean expressions	5.2.3.6	yes	
instantiation of props in other props	6.2.1	yes	
parameterized declaration of seq & props	6.3.2.1, 6.3.2.2	yes	
HDL in a vunit	7.2	yes	
vunits should not be able to see others	7.2.1	yes	
Modeling layer	8	limited	
support of complex report args	N/S	yes	
`endpoints`/`ended` in unclocked properties		yes	
clocked `endpoints/ended`		yes	Supported only if the clock expression of the endpoint is same as that of the entire property.
`endpoints/ended` on RHS of `sync_abort/async_abort`		no	
complex first argument of prev		yes	

© 2021 All Rights Reserved.

Table 27-7 Support for Specific PSL Constructs

PSL Construct	PSL1850-2005 LRM Section	Supported	Notes
Assertions with large repeat counts		no	It is not supported by default. Supported by providing a high value for `+assertMaxIter`.
Multiclocked properties/ sequence		yes	Optional clock argument in PSL-builtin functions should be same as that of the sequence/property expression. The sequence fusion, sequence and (&), sequence or (I), sequence length matching and (&&), and sequence within operators are not supported. The repetition of multiply clocked sequence is not supported. The property operator until, until_, before, before_, abort, sync_abort, async_abort, property and, property or, logical if, logical iff, and overlapping implication are not supported to have multiply clocked operands.

© 2021 All Rights Reserved.

Support for Open Verification Library Components

The Open Verification Library (OVL) components can be compiled to use SVA or PSL for the underlying assertion language.

Supported OVL Components

IXCOM supports the following OVL 2.6 version components.

```
ovl_code_distance
ovl_hold_value
ovl_no_contention
ovl_reg_loaded
ovl_stack
ovl_always
ovl_always_on_edge
ovl_bits
ovl_change
ovl_cycle_sequence
ovl_decrement
ovl_delta
ovl_even_parity
ovl_fifo_index
ovl_frame
ovl_handshake
ovl_implication
ovl_increment
ovl_mutex
ovl_never
ovl_next
ovl_no_overflow
ovl_no_transition
ovl_no_underflow
ovl_odd_parity
ovl_one_cold
ovl_one_hot
ovl_proposition
ovl_quiescent_state
ovl_range
ovl_req_requires
ovl_time
ovl_transition
ovl_unchange
ovl_valid_id
ovl_value
ovl_width
ovl_win_change
ovl_win_unchange
ovl_window
ovl_zero_one_hot
ovl_never_unknown
ovl_never_unknown_async
ovl_next_state
ovl_fifo
ovl_arbiter
ovl_multiport_fifo
```


© 2021 All Rights Reserved.

```
ovl_memory_async
ovl_memory_sync
ovl_req_ack_unique
```


© 2021 All Rights Reserved.

28

Methodology Guidelines for Compiling Assertions with the IXCOM Compiler

This appendix includes the following topics:

- Inspecting IXCOM Log File for Assertion-Related Information
- Number of Assertions Synthesized by the IXCOM Compiler
- Advanced Support for SVA and PSL Assertions with Behavioral Compilation
- Differences in Handling of Assertions
- Best Practices for Assertions

Inspecting IXCOM Log File for Assertion-Related Information

After compiling the design with the IXCOM compiler, refer to the compilation log file for assertion-related information, and get answers to questions, such as the following:

- How many assertions were present in the design?
- Did the IXCOM compiler find a syntax or semantic violation (error) in the definition of the assertion (property or methods)? See Inspecting IXCOM Log File for Assertion-Related Information.
- Did the IXCOM compiler ignore an assertion because:
  - It is not supported in the current release?
  - Another compiler switch was required?

    For example, you need to use the `+assertMaxIter+<N>` option to increase the limit of repetitions.

© 2021 All Rights Reserved.

- Did the IXCOM compiler optimize away an assertion in the `always or never succeed` category?

Number of Assertions Synthesized by the IXCOM Compiler

During compilation, the IXCOM compiler generates an error if the design is too large to compile for the specified number of domains. In other cases, the IXCOM compiler might be able to compile the design, but report in the log file that the design requires a large number of steps to run. Occasionally, even if the design compiles successfully, it is possible that the design requires a large number of behavioral cycles while running on the emulator. If you observe such behavior, check the IXCOM log file to see if the number of assertions is unusually large.

Note: If there are a large number of assertions in one module, HDL-ICE Compiler might take very long to synthesize the assertions. In addition, there are internal registers created by the compiler, for which the widths are determined by the number of assertions per module. If the number exceeds 2097151, compilation fails due to the width of the registers.

After elaborating the HDL files, the IXCOM compiler lists the following in the log file, ***before*** generating the instrumentation for the assertions:

- Number of unique assertions found in the design and testbench
- Number of total instances of the assertions

  An assertion might have more than one instance when it is in a `generate` block or if the module containing the assertion is instantiated multiple times.
- Number of PSL assertions, number of concurrent SystemVerilog assertions, and the number of concurrent SystemVerilog assertions present in a procedural context

After generating the instrumentation for the assertions, the IXCOM log file includes the following list of:

- Supported and unsupported assertions

  Note: The instrumentation is generated only for the supported assertions.
- Number of assertions in the DUT
- Number of assertions in the testbench

© 2021 All Rights Reserved.

Example with Count of Assertions Displayed in the ixcom.log File

```
Top level design units:
        tb
Packages:
Design hierarchy summary:
                                        Instances     Unique   Splits
        Modules:                            6           7        6

Warning! in file logic.v at line 15             [ASSERTION_UNSUPPORTED]
       Unsupported concurrent assertion LOGICINVALIDCMD: sequence match items and
multiclocked assertions

Warning! in file logic.v at line 15             [ASSERTION_UNSUPPORTED]
       Unsupported concurrent assertion LOGICINVALIDCMD: sequence match items and
multiclocked assertions

        Assertions Statistics:
            Assertion summary by types:
                                                                Instances   Unique
              Concurrent Assertions                           :    8          8
              PSL                                             :    4          4
              SVA                                             :    3          3
              Procedural SVA                                  :    1          1

Post processing summary of concurrent assertion instances:
     Supported (Evaluated at runtime)                         :    7
         Assertions in DUT (Mapped to HW partition)           :    5
         Assertions in TB (Mapped to SW partition)            :    2
         Un-supported                                         :    1
```

Errors Generated during Compilation of a Design with Assertions

The IXCOM compiler might generate errors or warnings during compilation of the design during the analyzer or elaborator stages, before it has synthesized the assertions. These messages can be found in the log files.

The following sections provide a brief description of the different types of errors and warnings.

Finding Syntax and Semantic Errors in a Log File

If an assertion is written in illegal HDL syntax, the analyzer gives an error and aborts. If an assertion is written in violation of the LRM, the elaborator gives an error message, prefixed with `ASSERTION_ILLEGAL`.

Example of Compilation Errors in ixcom.log File

```
Compiling source file "controller.v"
Highest level modules:
        top
Design hierarchy summary:
                                Instances     Unique      Splits
        Modules:                    3           3           3
        Assertions:                 2           2
```


© 2021 All Rights Reserved.

```
Error! in file controller.v at line 31              [ASSERTION_ILLEGAL_RANGE]
        Range must be ascending in a repeat sequence or a range sequence
        Errors : 1 Warnings : 0 Info : 0
  // Log end : Wed Jul  6 19:31:27 2011
```

Finding Unsupported Constructs in a Log File

The IXCOM compiler supports a large variety of SVA and PSL constructs, but some constructs are only partially supported because these cannot be mapped to the emulator. Such cases are reported in the IXCOM log file as warnings, prefixed with `ASSERTION_UNSUPPORTED`.

The IXCOM compiler ignores such assertions and allows the compilation to proceed. One such example is shown below.

Note that the IXCOM compiler reports fewer synthesized assertions (0 assertions in this case).

Example Warnings for Unsupported Assertions in ixcom.log File

```
Compiling source file "controller.v"
Highest level modules:
        top
Design hierarchy summary:
                        Instances   Unique   Splits
        Modules:                3      3        3
        Assertions:             0      0
Warning! in file controller.v at line 31               [ASSERTION_UNSUPPORTED]
        Unsupported concurrent assertion DUT1: real operand type in a boolean
expression
        Errors : 0 Warnings : 1 Info : 0
  // Log end : Wed Jul  6 19:19:34 2011
```

Optimized Assertions

If the sequence expression of an assertion is a constant expression, the IXCOM compiler computes it during compilation. The IXCOM compiler generates a warning about such an assertion, noting in the log that it either `always succeeds` or `never succeeds`, and ignores it while mapping the DUT to the emulator. This way, the IXCOM compiler can save hardware capacity while compiling a design. To understand how to rewrite such a concurrent assertion so that it is not optimized away, refer to Using Immediate Assertions when the Sequence Expression is a Constant Expression.

© 2021 All Rights Reserved.

Assertions Using X or Z Values

The `X` or `Z` values are not supported on the emulator. These are converted to `0` or `1` when the design is moved to hardware.

Those assertions that check the value of a variable to be `X` or `Z`, and meet the criteria listed below are optimized from the hardware partition and run only on the software simulator:

- The property must not be temporal and are only consequent. This means that the properties using immediate implication are not optimized.
- The sequence expressions must result in a constant on the hardware because of checks for `X` and `Z` values. For example, the expression `(var === 1‘bX)` will always be false on the hardware and, will, therefore, reduce all expressions AND’ing with it to a constant.

Example of IXCOM-Compiler Optimization of an Assertion that Checks for X or Z Values

```
property p1;
  ((sel[1] === 1'bZ || sel[2] === 1'bZ) &&
   (A[width-1:0] !== 1'bZ && B[width-1:0] !== 1’bZ) ||
   !assert_driven_sel)
endproperty
```

In the above example, the assertion runs only on the software simulator, and a warning is issued during compilation:

```
Warning in file test.v at line 45      [ASSERTION_OPTIMIZED_ON_HW]
Assertion checking ‘X’ or ‘Z’ values A1:  it will execute only when design is run
on simulator
```

If an assertion that checks for `x` or `z` values does not meet the above criteria, the IXCOM compiler cannot automatically direct such assertions to be run only on the software simulator. As a result, the value conversion (from `x` or `z` to `0` or `1`) when a design swaps to hardware renders the `x` or `z` checking assertion as useless for verification, and unnecessarily consumes emulator capacity. As an alternative, use `‘ifdef` constructs to conditionally enable such assertions in a *simulation-only* environment.

Example of Writing Assertions Within ‘ifdef to Conditionally Enable or Disable Assertions

```
‘ifdef SIMONLY
ISXORZ : assert property (@(posedge clk) (a == 1‘bx));
‘endif
ixcom +sv +define+SIMONLY test.v //to enable the assertion in a simulation-only
                                 environment
ixcom +sv test.v //to disable or ignore the assertion in an emulation environment
```


© 2021 All Rights Reserved.

Reporting Weights (Estimates of Gate Counts) for Assertions

During compilation, the instrumentation for assertions adds extra logic, requiring more gates, depending on the complexity of the assertions. The estimated gate count for each assertion in the design is termed as weight of the assertion. It is based on certain heuristics, and it can differ significantly from the actual number of gates needed to emulate the assertion.

The assertion weights are reported in two CSV files, namely `assertion_inst.csv` and `assertion_mod_inst.csv`, in the current directory.

Note: The weights of each concurrent assertions in the design are reported. The weights for the immediate assertions are not included in the report.

Reporting Weights Of Assertion Instances

The weights of the assertions are reported in the `assertion_inst.csv` file. Following is an example of the content of this file:

Assertion Instance	Weightage	Module Name	Source File	Line Number	Behavioral
tb_top.inst1.dt1.ba1.B1	100	mod_assertion	bind_mod.v	14	No
tb_top.inst1.dt1.ba1.B2	50	mod_assertion	bind_mod.v	15	No
tb_top.inst1.dt1.vu1.VA1	100	dut1	prop.psl	12	No
tb_top.inst1.dt2.dt3.A4	5100	dut3	design.v	40	Yes
tb_top.inst1.dt2.dt3.vu2.VA2	50	dut3	prop.psl	16	No
tb_top.inst1.dt2.dt3.vu2.VA3	50	dut3	prop.psl	17	No
tb_top.inst1.dt2.dt4.A4	5100	dut3	design.v	40	Yes
tb_top.inst1.dt2.ba2.B1	100	mod_assertion	bind_mod.v	14	No

© 2021 All Rights Reserved.

Assertion Instance	Weightage	Module Name	Source File	Line Number	Behavioral
tb_top.inst1.dt2.ba2.B2	50	mod_assertion	bind_mod.v	15	No
tb_top.inst1.A1	100	dut_top	design.v	14	No
tb_top.inst1.A2	50	dut_top	design.v	15	No

Reporting Weights Of Module Instances

The hierarchical weights of assertions are reported in the `assertion_mod_inst.csv` file. The Local Weightage of module instance is the sum of Weightage of all the assertions in that instance of the module. The Weightage of module instance is the sum of Local Weightage of that module instance and Weightage of all the sub-instances of that module instance.

Consider the following example:

Module Instance	Weightage	Local Weightage	Module Name	Source File	Line #
tb_top.inst1.dt1.ba1	150	150	mod_assertion	bind_mod.v	2
tb_top.inst1.dt1	250	100	dut1	design.v	21
tb_top.inst1	250	0	dut	design.v	45

Here, `tb_top.inst1.dt1` is a module instance of the module named `dut1`. The sub-instance of module instance `tb_top.inst1.dt1` is `tb_top.inst1.dt1.ba1`. The module instance `tb_top.inst1.dt1.ba1` has no sub-instance. So, its weightage is the same as local weightage, which is 150. The module instance `tb_top.inst1.dt1` has one sub-instance named `tb_top.inst1.dt1.ba1` with weightage 150. It has local weightage 100. So, its weightage is 250 (100 + 150). The module instance `tb_top.inst1` has only sub-instance `tb_top.inst1.dt1` with weightage of 250. It has no local weightage. So, its weightage is the same as sum of weightage of all its sub-instances, equalling to 250 because it has only one sub-instance.

© 2021 All Rights Reserved.

Following is an example of the `assertion_mod_inst.csv` file.

Module Instance	Weightage	Local Weightage	Module Name	Source File	Line Number
tb_top.inst1.dt1.ba1	150	150	mod_assertion	bind_mod.v	2
tb_top.inst1.dt1	250	100	dut1	design.v	21
tb_top.inst1.dt2.dt3	5200	5200	dut3	design.v	33
tb_top.inst1.dt2.dt4	5100	5100	dut3	design.v	33
tb_top.inst1.dt2.ba2	150	150	mod_assertion	bind_mod.v	2
dut2	10450	0	dut2	design.v	26
tb_top.inst1	10850	150	dut_top	design.v	2

Advanced Support for SVA and PSL Assertions with Behavioral Compilation

The IXCOM compiler can map most of the assertion constructs in the DUT to the emulator so that the module or entity that contains these assertions is not treated as an embedded module. However, some constructs cannot be easily mapped to hardware and would normally be part of the software partition. These include assertions that contain multiple clocks or local variables. The IXCOM compiler can map these assertion constructs to the emulator by compiling the modules and entities containing these assertions as behavioral modules. Thus, more assertions can be supported on the emulator, and need not be restricted to the software partition.

Mapping the assertions in this manner increases step count and behavioral evaluations, thereby, slowing down the emulator and/or using extra hardware capacity, depending on the complexity of the assertions, and the number of times these are invoked. This would also reduce the resources available for other parts of the design.

© 2021 All Rights Reserved.

When a module is compiled as a behavioral module, the following information message, implying that one or more assertions in a module requires behavioral compilation, is generated in the `xc_work/bc.log` file. Note that you need not give any additional compiler options. The IXCOM compiler switches to behavioral compilation automatically to handle these types of assertions.

```
Module memory at line 45 in mclktest.v
  id:2, svaBehMod
-------

Latch mode not used: sva module

Automatic Setting of rtl_clk:
  vars marked as rtl_clk:
    clk

Beh Procs found at lines 55, 55

Multiple Drivers:
rtl_clk used as XpClk:
  clk

Detected dynamic loop related to line 55 in module memory
```

Check the log file for messages similar to those shown above to determine if the step count is unexpectedly high or the operating speed is low.

Tip

To determine the reason for an unusual step count and operating speed, temporarily ignore the assertions (by using the `+no_sva+<mod>` and `+no_psl+<mod>` options of the `ixcom` command) in the design or specifically in the module, which was compiled as a behavioral module, and recompile the design. Then, examine the new statistics.

```
INFO (db2util-1067): This design is scheduled in 110 steps.
INFO (qt2dadb-1086): Maximum emulator operating speed is 4310 kHz.
```

To see the run-time impact of behavioral cycles, check for messages in the run-time log files like `xeDebug.log`.

```
XErun> exit
--- HW execs 9 evals 4 bevals 8 tbsyncs in 3.05 sec (338445.33 us/eval;
```

Assertions with Multi-Clocked Consequent

Multi-clocked assertions of the form shown below are supported on the emulator by using behavioral compilation.

Cadence recommends that you write single-clocked assertions when possible. Single-clocked assertions use fewer hardware resources on the emulator, and facilitate easy debug if the assertion fails.

© 2021 All Rights Reserved.

Example of Single-Clocked and Multi-Clocked Assertions in the Design

```
module memory(input wire CLK1,CLK2,a,b,c,d);
  //multi-clocked assertion MA1
  MA1: assert property (@(posedge CLK1) a ##1 b |=> c ##1 @(posedge CLK2) d);
  //singly-clocked assertion SA1
  SA1: assert property (@(posedge CLK1) a ##1 b |=> c ##1 d);
endmodule
```

Local Variables and Sequence Match Items

Assertions containing local variables (or sequence match items) of the form shown below are supported on the emulator with behavioral compilation.

Cadence recommends that you write assertions without local variables whenever possible. Performance is generally better if the local variable stays within a sequence, and does not cross over a property operator.

Example of Assertions Containing Local Variables

```
module memory(input wire CLK,a,b,c,d);

  property LP1;
   reg lv1;
   (a,lv1=1'b0) ##1 (b,lv1=1'b1) |=> (lv1) ##1 c;
  endproperty
  //assertion with local variables
  LA1: assert property (@(posedge CLK)LP1());

  reg seqMatch2;

  property LP2;
   (a,seqMatch2=1'b0) ##1 (b,seqMatch2=1'b1) |=> (seqMatch2) ##1 c;
  endproperty
  //assertion with sequence match items
  SM2: assert property (@(posedge CLK)LP2());
endmodule
```

Handling of Large Delays

The IXCOM compiler uses behavioral compilation to handle large delays in some cases.

- Single large delay in property declaration:

  If a single large delay is found in the form seen in property `P2` below, and the assertion `C2`, which asserts this property, the IXCOM compiler internally uses behavioral compilation to compile this assertion for the same reason as used for `C1` above.

  ```
  property P2;
      (@(posedge clk) a ##[1:250] b |=> c[*10:$] ##1 d);
  endproperty
  C2 : assert property (P3());
  ```


© 2021 All Rights Reserved.

- Single large delay in assertion statement:

  If a single large delay is used in an assertion that directly asserts an expression, the IXCOM compiler generates a warning about an unsupported assertion, and suggests that the `+assertMaxIter+<N>` option should be used for compilation. When you include this option, behavioral compilation is not required for this assertion.

  ```
  C3 : assert property (@(posedge clk) a ##[1:250] b |=> c[*10:$] ##1 d);
  ```

Handling expect Statements

The IXCOM compiler supports `expect` statements in the DUT and testbench. However, if the `expect` statement exists in a DUT module, you need to explicitly specify behavioral compilation for that module by using one of the following methods:

- Use the `+bc+<module_name>` option of the `ixcom` command
- Add an `ixc_ctrl` directive in the module

Example of Using the +bc Option

```
ixcom +sv +dut+dut +bc+dut dut.v tb.v
```

Example of Using the $ixc_ctrl Directive

```
module dut(...);
   initial $ixc_ctrl("map_bc");
   ...
   ...
endmodule
```

Differences in Handling of Assertions

The following sections describe the differences between assertion handling by

- Assertion-Handling Differences Between Xcelium Simulator and Emulator (Compiled Using the IXCOM Compiler)
- Assertion-Handling Differences Between the IXCOM Compiler and LRM

© 2021 All Rights Reserved.

Assertion-Handling Differences Between Xcelium Simulator and Emulator (Compiled Using the IXCOM Compiler)

Clearly, an assertion can fail when the design and assertions are run on the emulator. If the same assertion does not fail, or fails at a different simulation time, or fails a different number of times on a simulator such as *xrun*, the discrepancy needs to be investigated.

Trace-Based Counting

One source for such discrepancies is that IXCOM always does trace-based counting whereas another simulator (like xrun) might perform attempt-based counting by default. In attempt-based counting, an assertion fails or finishes only once. Multiple finishes or failures from a single attempt are not counted or reported. In trace-based counting, each trace of the assertion attempt might fail or finish individually. Refer to the *Assertion Checking in Simulation* guide for more explanation about the assertion counting modes. To run *xrun* with trace-based counting, use the `-assert_count_traces` option of the `xrun` command.

It is also possible that a failing assertion in the Xcelium simulator might not fail when the design is run on the emulator. This is possible when the IXCOM compiler does not support a particular assertion and ignores it during compilation. If Xcelium supports this assertion, the results between IXCOM and Xcelium are different when the assertion passes or fails.

In-Flight Assertions at End of Simulation

Another difference between IXCOM and Xcelium is the treatment of the assertion-in-flight at the end of simulation. The IXCOM compiler reports these as failures whereas Xcelium, by default, does not report these as failures.

Assertion-Handling Differences Between the IXCOM Compiler and LRM

The following sections outline the differences between the handling of assertions as done by the IXCOM compiler and that prescribed by the LRM.

Vacuous Passes

The IXCOM compiler does not support vacuous passes of assertions if the antecedent itself failed while evaluating the property. Such cases do not trigger the execution of the pass action block. Simulators, such as Xcelium, typically, trigger the execution of pass action block even when the assertion passes vacuously.

© 2021 All Rights Reserved.

Naming generate Blocks

The IXCOM compiler and Xcelium have a different naming strategy when inlining generate blocks. In the following example, the IXCOM compiler will generate inlined blocks with names like `gblk_0_`, `gblk_1_` and so on. Xcelium, on the other hand, will generate inlined blocks with names like `gblk[0]`, `gblk[1]` and so on.

This will cause differences anywhere assertions are referred to using hierarchies like:

- Assertion summary results will show assertion as "`top.dt.gblk_0_.C1`" in the IXCOM compiler and "`top.dt.gblk[0].C1`" in Xcelium.
- The TCL commands used for turning off assertions in the IXCOM compiler and Xcelium will be different because of the full hierarchical name.
- The full path leading to "an assertion in a generate block" will need to be different in ACTs like `$assertoff`, `$asserton`, or `$assertkill`.

Example

```
genvar i;
generate for(i=0;i<5;i++) begin : gblk
  C1 : assert #0 (b & c | d);
  assert #0 (b & c | d);
end
endgenerate
```

Naming vunit Blocks

When a vunit is bound to a module by name, the IXCOM compiler maps it to a named inlined block in the module. Xcelium instead will inline it in the module. This causes assertions to have a different hierarchical name in the IXCOM compiler as compared to that in Xcelium. For example, consider the assertion failure messages below for the same assertion. The first one is when xrun compiles the design. The second one is when design is compiled by the IXCOM compiler. The name of the vunit in this example is "`myvunit1`".

```
xmsim: *E,ASRTST (./a.prop,6): (time 5 NS) Assertion top.dt1.AV1 has failed
xmsim: *E,ASRTST (a.prop,6): (time 5 NS) Assertion top.dt1.myvunit1.AV1 has failed
```

Best Practices for Assertions

The following sections suggest best practices for compiling the designs with assertions with the IXCOM compiler and running these designs on the Palladium Z1 emulator.

- Using ifdef, Bind, or Vunits to Write Assertions

© 2021 All Rights Reserved.

- Using Immediate Assertions when the Sequence Expression is a Constant Expression
- Using Variables Instead of Constants in Sequence Expressions
- Labeling Assertions

Using ifdef, Bind, or Vunits to Write Assertions

It is possible to add an assertion into the design after the design has been coded. This is useful in the case of system level integration or when using a third-party IP.

You can use the *bind* approach to writing SystemVerilog Assertions and *attach* or *bind* separate files including assertions with a module in the DUT. Similarly, you can use PSL vunits to attach PSL assertions to modules within the DUT. Both approaches facilitate writing and adding of assertions in the verification environment much later in the developmental cycle, without having to modify the DUT. Such assertions can also be intentionally and temporarily ignored easily at compile time by not including the *bind* or *vunit* file, containing the assertions content, in the compile script.

For more information and examples on *bind* and *vunit*, refer to the SystemVerilog and PSL LRM documents, respectively.

Another way of conditionally including SystemVerilog Assertions or PSL in Verilog assertions is by writing them in `` `ifdef <*define_name*> `` blocks in the HDL. This allows you to enable and disable assertions during compilation by defining the appropriate `<define_name>` during compilation.

Using Immediate Assertions when the Sequence Expression is a Constant Expression

If the sequence expression of an assertion is a constant expression, the IXCOM compiler computes it, and reports it in the IXCOM log file that an assertion was found to be of the `always succeeding` or `never succeeding` type. Such an assertion is ignored at run time, and does not `finish` or `fail` because the IXCOM compiler optimizes it away after issuing a warning message about this condition.

You might want to use assertions to safeguard against an accidental change in the design from setting an inappropriate value to one or more parameters, which would otherwise lead to an incorrect value of a constant sequence expression. With such usage of assertions, a compile-time warning about `always succeed` or `never succeed` might not be enough. If you want such an assertion to fail at run time, Cadence recommends that you rewrite this concurrent assertion as an immediate assertion in an `initial` block. Immediate assertions

© 2021 All Rights Reserved.

have the least cost in terms of capacity and run-time performance. The IXCOM compiler does not ignore or optimize away immediate assertions during compilation.

Example of Rewriting a Concurrent Assertion as an Immediate Assertion

This example illustrates a concurrent assertion that checks the value of a parameter in the procedural code. The IXCOM compiler regards an assertion that includes a constant (parameter) and no other variables, as either `always succeed` or `never succeed` depending on the value of the parameter; it can be over-ridden.

```
module memory(input wire CLK,a,b);
  parameter DEPTH=32'b0100;
  always @(posedge CLK) begin
    A1: assert property (DEPTH < 32'b0100);
    <other_code...>
  end
endmodule
```

The IXCOM compiler ignores such concurrent assertions, and generates the following message in the IXCOM log file:

```
Info in file proc.v at line 4          [ASSERTION_IGNORED]
        Ignore concurrent assertion A1: always succeed
```

Rewrite such assertions as `immediate` assertions in the following manner if you want a run-time failure when the value of constant expression is invalid. Immediate assertions are easier to synthesize and ideal for such cases.

```
module memory(input wire CLK,a,b);
parameter DEPTH=32'b0100;
  initial begin
   A1: assert (DEPTH < 32'b0100);
  end
  always @(posedge CLK) begin
   <other_code>
  end
endmodule
```

Using Variables Instead of Constants in Sequence Expressions

In the previous section, a concurrent assertion was converted to an immediate assertion to check the value of a parameter at run time. However, at times, you might want an assertion to be evaluated continuously instead of only once in an `initial` block. In the following example, the assertion has been written to check for a sequence that never happens. As a result, a constant consequent `0` has been written so that it always fails once the antecedent has been found successfully.

© 2021 All Rights Reserved.

Example of Rewriting an Assertion by Replacing Constants with Variables to Prevent it from Getting Optimized

This example illustrates a concurrent assertion equivalent to `never` because it has a constant expression `0` as a consequence. The IXCOM compiler optimizes away such assertions in the `never succeed` category. This helps in making the hardware capacity available on the emulator.

```
module memory(input wire CLK,a,b);
  always @(posedge CLK) begin
   A1: assert property (a ##1 b |=> 1'b0);
   <other_code>
  end
endmodule
```

If this assertion checks that the `a ##1 b` sequence `never` happens, and you do not want this assertion to be optimized away, rewrite the assertion as follows:

```
module memory(input wire CLK,a,b);
   wire zero = 1'b0;
  always @(posedge CLK) begin
    A1: assert property (a ##1 b |=> zero);
    <other_code>
  end
endmodule
```

Labeling Assertions

For the most efficient use of assertion-related features when using the IXCOM compiler, it is important to not leave any assertions in the design unlabeled. The IXCOM compiler generates the instrumentation for unlabeled assertions, but assertions with auto-generated labels (in PSL) or no labels (in SVA) are not very informative, and you would need to rely on the line or file information to match the real assertion in the design with the IXCOM-generated label of the assertion. Debugging and gathering coverage information of such assertions, therefore, is not a very straightforward process. In the absence of a naming ideology in the SystemVerilog LRM, all verification tools have their own labeling mechanism for such assertions, thereby, further increasing the inconsistency. The difference in label generation for unlabeled assertions across different tools makes it difficult to track reports generated from different tools. So, label the assertions to ease debugging and collecting of coverage data from different tools.

© 2021 All Rights Reserved.

29

Implementing SCE-MI Pipes

The VXE TBA solution provides an implementation of the SCE-MI pipes interface. A SCE-MI pipe is defined as a construct that enables streaming of transactions between the testbench partition (C-side in SCE-MI definition) running on the simulator and hardware partition (HDL-side in SCE-MI definition) running on the Palladium Z1 emulator.

Is it correct to say that the SCE-MI pipes feature is available only in the IXCOM releases and that it is not available in HDLICE and VXE releases?

If the above statement is correct, which all IXCOM releases does this change apply to? - IXCOM 20.05 or 20.12 or both?

VXE supports SCE-MI 2.0, 2.1, and 2.2 standards for SCE-MI pipes. The SCE-MI 2.1 implementation is not completely backward-compatible with the SCE-MI 2.0 implementation because of variations in the two standards. VXE provides two sets of libraries for the two SCE-MI implementations, as discussed in "Software Components of SCE-MI Pipes Interface" on page 1339.

If only IXCOM supports SCE-MI, then we should say IXCOM supports, not VXE supports. Please confirm so that this change can be implemented in the doc.

This chapter contains the following sections:

- Overview of Transaction Pipes
- SCE-MI 2.2 Features
- SCE-MI 2.1 Pipes Interface Features
- Software Components of SCE-MI Pipes Interface
- Testbench-Side Thread-Neutral C Interfaces
- DUT-Side HDL Interface
- Executing the Simulator Thread
- Reset of SCE-MI Pipes

© 2021 All Rights Reserved.

- Using Language-Based Adaptors
- Generating DUT Clocks with SCE-MI Pipes
- Compiling and Running Designs with SCE-MI Pipes
- Examples: Implementing SCE-MI Pipes

Overview of Transaction Pipes

VXE provides support for the SCE-MI 2.2, SCE-MI 2.1, and SCE-MI 2.0 implementations of transaction pipes as defined by:

- Section 5.8, *Pipes-Based Interface: Transaction Pipes of the Accellera Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual Version 2.2*.
- Section 5.8, *Pipes-Based Interface: Transaction Pipes* of the *Accellera Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual Version 2.1*.
- Section 5.7, *Pipes-based Use Model: Transaction Pipes* of the Accellera SCE-MI Reference Manual Version 2.0.

Refer to these manuals for a complete description of Pipes functionality and the SCE-MI API.

All the features described in the relevant sections of these manuals are implemented in VXE, exactly as specified, unless otherwise stated in this chapter.

SCE-MI Transaction Pipes are unidirectional, which means that the transactions flow only in one direction. This ensures that the transactions are received in the same order as they were sent when the receiver accesses these transactions through function calls. SCE-MI pipes are of the following two types:

- **Input pipes**: SCE-MI pipes that pass one-way transactions from the testbench (running on the simulator) to the DUT (running on the emulator).
- **Output pipes**: SCE-MI pipes that pass one-way transactions from the DUT (running on the emulator) to the testbench (running on the simulator).

SCE-MI 2.2 Features

This section discusses the VXE implementation for the following features of SCE-MI 2.2. The reference section number in the *Accellera Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual Version 2.2* has been specified in square brackets.

- C++ HVL-side Transaction Pipes API [5.8.2.4]

© 2021 All Rights Reserved.

- SystemVerilog HVL-side Transaction Pipes API [5.8.2.3]
- Clocked SCE-MI Pipes [4.8.9]
- Byte Array Access Functions [5.8.2.1 C-Side transaction pipes API]
- Direct Memory Interface [5.9]

C++ HVL-side Transaction Pipes API

Following is a list of classes and functions for C++ HVL-side transaction pipes per SCE-MI 2.2. In the *Accellera Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual Version 2.2*, these functions are listed under section 5.8.2.4.

Please share the correct and complete path for scemi22_cpp.h .

To use these functions, include the `scemi22_cpp.h` file located in <~~*XE_INSTALL_DIR*~~*IXCOM_INSTALL_HOME*>/etc/tba.

- Base class: `scemi_pipe`
- Derived classes: `scemi_input_pipe` and `scemi_ouput_pipe`

```
class scemi_pipe {
     scemi_pipe (const char *hdl_path); // input: Path to HDL endpoint
     instance~scemi_pipe();
// Common API functions
    svBit get_direction();
    int get_depth();
    int get_bytes_per_element();
    scemi_pipe_notify_callback_handle set_notify_callback
         (scemi_pipe_notify_callback notify_callback,
         void *notify_context, int callback_threshold = 0);
    void clear_notify_callback (scemi_pipe_notify_callback_handle
    notify_callback_handle);
    void *get_notify_context(
    scemi_pipe_notify_callback_handle notify_callback_handle);
    int put_user_data(void *user_key,void *user_data);
    void *get_user_data(void *user_key);
    svBit set_eom_auto_flush(svBit enable);
};
class scemi_input_pipe: public scemi_pipe {
     scemi_input_pipe(const char *hdl_path);
     ~scemi_input_pipe();
// API functions for input pipe
    void send(
    int num_elements,
    const svBitVecVal *data,
    svBit eom);
```


© 2021 All Rights Reserved.

```
    void flush();
    int try_send(
    int byte_offset,
    int num_elements,
    const svBitVecVal *data,
    svBit eom);
    int try_flush();
    int can_send();
};
class scemi_output_pipe: public scemi_pipe {
     scemi_output_pipe(const char *hdl_path);
     ~scemi_output_pipe();
// API functions for output pipe
    void receive(
    int num_elements,
    int *num_elements_valid,
    svBitVecVal *data,
    svBit eom);
    int try_receive(int byte_offset,
    int num_elements,
    svBitVecVal *data,
    svBit *eom);
    svBit pipe_in_flush_state();
    int can_receive();
    };
```

SystemVerilog HVL-side Transaction Pipes API

The SystemVerilog HVL-side transaction pipes API follows the C-side API very closely except that all the API functions are encapsulated as methods of a SystemVerilog class which is defined in the `scemi_pipes_pkg.sv` package in <~~*XE_INSTALL_DIR*~~*IXCOM_INSTALL_HOME*>/etc/tba.

Please review and specify the complete path for scemi_pipes_pkg.sv.

- The classes for the SystemVerilog HVL-side API package are as follows: `scemi_pipe_notify_callback`: This class is the base class for pipe notifications.
  - `scemi_pipe`: This class is the base class for data input and output of all System Verilog pipes. To transport dynamic byte arrays, the `scemi_pipe` class is directly extended to two classes: `scemi_dynamic_input_pipe` and `scemi_dynamic_output_pipe`.
    - `scemi_dynamic_input_pipe`: To transport packed bit vectors, this class is further extended to `scemi_static_input_pipe`.
    - `scemi_dynamic_output_pipe`: To transport packed bit vectors, this class is further extended to `scemi_static_output_pipe`.

© 2021 All Rights Reserved.

The following section lists the functions in these classes. For more information about the functions, see section 5.8.2.3 in the *Accellera Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual Version 2.2*.

```
class scemi_pipe_notify_callback
    virtual function void notify();
    new ();

class scemi_pipe;
    chandle pipe_handle;
    int depth;
    int byte_per_element
//functions

    new(input string hdl_path );
    bit get_direction();
    int get_depth();
    bit set_eom_auto_flush(input bit enable);
    bit in_flush_state();
    void set_notify_callback(scemi_pipe_notify_callback notify_callback,int
    callback_threshold = 0);
    void clear_notify_callback(scemi_pipe_notify_callback notify_callback );
    class scemi_dynamic_input_pipe;

//Functions

    send_bytes(input int num_elements,
    ref byte unsigned data[],
    input bit eom);
    flush();
    int try_send_bytes(
         input int byte_offset,
         input int num_elements,
         ref byte unsigned data[],
         input bit eom);
    bit try_flush();
    int can_send();

class scemi_dynamic_output_pipe;

//Functions

    receive_bytes(input int num_elements,
         output int num_elements_valid,
         ref byte unsigned data[], output bit eom);

    int try_receive_bytes(
         input int byte_offset,
         input int num_elements,
         ref byte unsigned data[], output bit eom);
    int can_receive();

class scemi_static_input_pipe;

tasks/functions:

    send_bits(

        input int num_elements,
        input bit [STATIC_PAYLOAD_MAX_BYTES*8-1:0] data,
        input bit eom);

    int try_send_bits(

        input int byte_offset,
        input int num_elements,
        input bit [STATIC_PAYLOAD_MAX_BYTES*8-1:0] data,
        input bit eom);
```


© 2021 All Rights Reserved.

```
class scemi_static_output_pipe;
//Functions:
    receive_bits(
        input int num_elements,
        output int num_elements_valid,
        output bit [STATIC_PAYLOAD_MAX_BYTES*8-1:0] data,
        output bit eom);
    int try_receive_bits(
        input int byte_offset,
        input int num_elements,
        output bit [STATIC_PAYLOAD_MAX_BYTES*8-1:0] data,
        output bit eom);
```

Clocked SCE-MI Pipes

The Clocked pipes are used for complex use models that make calls with multiple elements simultaneously and require managing the `byte_offset` argument of the pipe.

In the clocked pipe usage, the pipe has an optional port to which a clock can be connected. The following example shows usage of a clocked input pipe:

```
scemi_input_pipe #(..., .IS_CLOCKED_INTF=1) input_pipe(clock);
always @(posedge clock)begin
     input_pipe.receive(0,1,data,eom);
     <process received data>
end
```

The `IS_CLOCKED_INTF` parameter of a pipe controls whether or not clocking is used. This parameter can have the following values:

- `0`: Clocking disabled: The clock input to the pipe is completely ignored. So, the pipe is an unclocked pipe.
- `1`: Clocking enabled: The clock input to the pipe is required.

The source file for clocked pipes is `scemi_pipe22.svp`, which is located in <~~*XE_INSTALL_DIR*~~*IXCOM_INSTALL_HOME*>/etc/tba.

Please specify the correct and complete path.

Examples

```
interface scemi_input_pipe( input bit pipe_clock);
// Existing parameters
    parameter IS_CLOCKED_INTF = 0;
    .....
    task receive (
        input int num_elements,
        output int num_elements_valid,
```


© 2021 All Rights Reserved.

```
        output bit [PAYLOAD_MAX_BITS-1:0] data,
        output bit eom,
        input int sync_control = IS_CLOCKED_INTF );
…endinterface
interface scemi_output_pipe( input bit pipe_clock);
// Existing parameters
    parameter IS_CLOCKED_INTF = 0;
    …
    task send (
        input int num_elements,
        input bit [PAYLOAD_MAX_BITS-1:0] data,
        input bit eom,
        input int sync_control = IS_CLOCKED_INTF);
        task flush (
        input int sync_control = IS_CLOCKED_INTF );
        …
endinterface
```

In any of the blocking calls such as `receive()` on an input pipe, and `send()` and `flush()` on an output pipe, the last argument of the call is a flag indicating if the call blocks asynchronously or is to sync on the posedge of the clock or the negedge of the clock. Accordingly, the valid values of `sync_control` are as follows:

- `0`: block asynchronously
- `1`: sync on clock posedge
- `2`: sync on clock negedge

The default value of `sync_control` is the value of the `IS_CLOCKED_INTF` parameter as shown in the example above.

For more information on the clocked pipes, see section 5.8.5.4.1(Clocked vs unclocked pipe semantics) in the *Accellera Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual Version 2.2*.

Byte Array Access Functions

VXE supports the following byte array access functions per SCE-MI 2.2. In the *Accellera Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual Version 2.2*, these functions are listed under section 5.8.2.1.

To use these functions, include the `uatba_scemi.h` file located in <~~*XE_INSTALL_DIR*~~*IXCOM_INSTALL_HOME*>/etc/tba.

Please share the complete path.

```
void scemi_pipe_c_send_bytes(
```


© 2021 All Rights Reserved.

```
        void *pipe_handle,                  // input: pipe handle
        int num_elements,                   // input: #elements to be written
        const char *data,                   // input: data
        svBit eom );                        // input: end-of-message marker
                                               flag

int scemi_pipe_c_try_send_bytes(             // return: #requested elements
                                             // that are actually sent

        void *pipe_handle,                   // input: pipe handle
        int byte_offset,                     // input: byte offset into data
        int num_elements,                    // input: #elements to be sent
        const char *data,                    // input: data
        svBit eom );                         // input: end-of-message marker flag

void scemi_pipe_c_receive_bytes(

        void *pipe_handle,                    // input: pipe handle
        int num_elements,                     // input: #elements to be read

        int *num_elements_valid,              // output: #elements that are valid
        char *data,                           // output: data
        svbit *eom );                         // output: end-of-message marker flag

int scemi_pipe_c_try_receive_bytes(            // return: #requested elements
                                               // that are actually received

        void *pipe_handle,                     // input: pipe handle
        int byte_offset,                      // input: byte offset into data, below
        int num_elements,                     // input: #elements to be read
        char *data,                           // output: data
        svBit *eom );                         // output: end-of-message marker flag
```

Direct Memory Interface

The SCE-MI Direct Memory Interface (SCE-MI DMI) provides a C-side interface to directly access an HDL-side memory. This interface increases the co-modeling efficiency by providing more controllability and observability of HDL-side memories.

To directly access an HDL-side memory, you need to retrieve a handle to the memory. This handle can be retrieved by the string path of the memory instance. The string path is the full path containing the memory array variable. Once the memory handle is retrieved, you can access the memory either by blocks of memory words or by a single memory word.

- Block Interface
- Word Interface

The C-side interface API consists of the following set of function declarations.

To use these functions, include the `uatba.h` and `marg_user.h` file located in <~~*XE_INSTALL_DIR*~~*IXCOM_INSTALL_HOME*>/etc/tba.

Please share the complete path to access uatba.h and marg_user.h

© 2021 All Rights Reserved.

Configuration and Query Functions

```
void *scemi_mem_c_handle(                // return : HDL-side memory handle
    const char *hdl_path );              // input : path to the memory
void scemi_mem_get_size(
        void *mem_handle,                // input : HDL-side memory handle
        unsigned int *width,             // output : bit-width of the memory
        unsigned long long *depth );     // output : depth of the memory
```

Block Interface

```
void scemi_mem_get_block(
        void *mem_handle,                      // input : HDL-side memory handle
        unsigned long long mem_word_offset ,   // input : memory address (offset)
        unsigned long long num_words,          // input : length of data
        void *data );                          // input : user data block
        void scemi_mem_put_block(
        void *mem_handle,                     // input : HDL-side memory handle
        unsigned long long mem_word_offset,   // input : memory address (offset)
        unsigned long long num_words,         // input : length of data
        void *data );                         // input : user data block
        Word Interface
void scemi_mem_write_word (
        void *mem_handle,                       // input : HDL-side memory handle
        unsigned long long mem_word_offset,     // input : memory address (offset)
        const svBitVecVal *data,                // input : memory word
        unsigned char *byte_enable_buf,         // input : byte enable buffer
        unsigned int flush = 0);                 // input : flush write buffer
void scemi_mem_read_word (
        void *mem_handle,                        // input : HDL-side memory handle
        unsigned long long mem_word_offset,      // input : memory address (offset)
        svBitVecVal *data,                       // output : memory word
        unsigned long long pre_fetch);           // input : number of words to pre-
        fetch
```

The sample Makefile files for the Direct Memory Interface are available in ${~~XE_INSTALL_DIR~~IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_dmi.

Please share the correct path.

Here, `num_words` means number of addresses of the memory. The word size depends on the memory-width definition of the `mem_handle` argument.

SCE-MI 2.1 Pipes Interface Features

This section discusses the VXE implementation for the following features of SCE-MI 2.1 Pipes Interface:

© 2021 All Rights Reserved.

- Deferred and Immediate Visibility Modes
- Notification Threshold
- Callback Threshold
- Functions of SCE-MI 2.1

Deferred and Immediate Visibility Modes

VXE supports both the `immediate` and `deferred` visibility modes defined by SCE-MI 2.1.

Should we say that "IXCOM supports both instead of VXE supports.."

Set the visibility mode for each pipe by setting an appropriate value for the HDL-side `VISIBILITY_MODE` parameter of the pipe interface. For details about setting the visibility mode in the pipes-based interface, refer to *Section 5.8.5.1.1* of the *Accellera Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual Version 2.1*.

SCE-MI defines the default value for the `VISIBILITY_MODE` parameter as `0`, which denotes an illegal pipe interface with unspecified mode of operation. If you do not set the `VISIBILITY_MODE` parameter, VXE sets this parameter to `immediate` and a warning message is generated at run time.

Notification Threshold

A pipe supports a static HDL-side notification threshold. The threshold can be:

- The minimum number of elements that must be added to an empty pipe for consumer notification, or
- The minimum number of elements that must be removed from a full pipe for a producer notification.

Specify the notification threshold by using the `NOTIFICATION_THRESHOLD` parameter.

When the notification threshold is set to 1, notifications due to send and receive operations occur immediately, that is, after adding at least one element to an empty pipe or removing at least one element from a full pipe. Notifications due to flush operations are not affected by the notification threshold setting.

The valid values for the `NOTIFICATION_THRESHOLD` parameter depend on the visibility mode. In the `deferred` visibility mode, the `NOTIFICATION_THRESHOLD` will be set to `BUFFER_MAX_ELEMENTS` regardless of the value you set. In the `immediate` visibility mode,

© 2021 All Rights Reserved.

`NOTIFICATION_THRESHOLD` will be set to `BUFFER_MAX_ELEMENTS` unless it is explicitly set to 1. Any value for `NOTIFICATION_THRESHOLD` other than `1` or `BUFFER_MAX_ELEMENTS` results in an error.

For details about notification threshold in the pipes-based interface, refer to *Section 5.8.5.1.3* of the *Accellera Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual Version 2.1*.

Callback Threshold

The `callback_threshold` can be set to a value between `1` and `NOTIFICATION_THRESHOLD`. For an input pipe, dynamic callback can occur whenever `scemi_pipe_c_can_send() >= callback_threshold`. For an output pipe, dynamic callback can occur whenever `scemi_pipe_c_can_receive() >= callback_threshold`. A warning message is generated if an invalid value is specified.

To ensure identical implementation-independent callback timing, `callback_threshold` should be set to the `notification_threshold` of the pipe.

For details about callback threshold in the pipes-based interface, refer to *Section 5.8.5.3.3* of the *Accellera Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual Version 2.1*.

Functions of SCE-MI 2.1

Following is a list of functions for SCE-MI 2.1. The reference section number in the *Accellera Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual Version 2.1* has been specified in square brackets.

- `void *scemi_pipe_get_user_data(void *pipe_handle, void *user_key);` [5.8.5.3.4]
- `void scemi_pipe_put_user_data(void *pipe_handle, void *user_key, void *user_data);` [5.8.5.3.4]
- `svBit scemi_pipe_get_direction(void *pipe_handle);` [5.8.5.3.3]
- `svBit scemi_pipe_c_in_flush_state(void *pipe_handle);` [5.8.5.3.2]
- `typedef void *scemi_pipe_notify_callback_handle;` [5.8.5.3.3]
- `scemi_pipe_notify_callback_handle scemi_pipe_set_notify_callback(`

  `void *pipe_handle,`

© 2021 All Rights Reserved.

```
scemi_pipe_notify_callback notify_callback,
void *notify_context,
int callback_threshold);
```
[5.8.5.3.3]

- `void scemi_pipe_clear_notify_callback(`
  `scemi_pipe_notify_callback_handle`
  `notify_callback_handle);` [5.8.5.3.3]

© 2021 All Rights Reserved.

Software Components of SCE-MI Pipes Interface

The core components for implementing the SCE-MI pipes interface are a set of encrypted HDL and pre-compiled shared libraries that implement the non-blocking `send` and `receive` operations of SCE-MI pipes interface.

Encrypted HDL and Pre-Compiled Shared Libraries

- **Encrypted HDLs:**

  The following Verilog files are used to implement the core functions of SCE-MI pipe and interfaces on the HDL-side:

  - For SCEMI-MI 2.0: `scemi_input_pipe.vp` and `scemi_output_pipe.vp`
  - For SCEMI-MI 2.1: `scemi_pipe21.svp`
  - For SCEMI-MI 2.2: `scemi_pipe22.svp`

  Note: The same set of files is used for implementing SCE-MI 2.0, SCE-MI 2.1, and SCE-MI 2.2 pipes. At compilation time, however, use the `+define+SCEMI21` option for a design that utilizes SCE-MI 2.1 and `+define+SCEMI22` option for a design that utilizes SCE-MI 2.2.

- **Pre-compiled shared libraries:**

  - For SCE-MI 2.0 pipes, use the `libuatba.so` and `libuatba_dc.so` shared libraries and the `uatba.h` C header file to implement the core functions of the pipe and thread-neutral C-interface for the testbench partition.

Shall we change VXE to IXCOM in the following para?

The VXE SCE-MI 2.0 libraries (`libuatba.so` and `libuatba_dc.so`) contain only non-blocking SCE-MI interface functions. To access the blocking interface functions, you need to compile the blocking interface source code provided with the VXE TBA solution, such as the `uatba_scemi.c`, `uatba_event.cpp` and `uatba_sc_event.cpp` files.

  - For SCE-MI 2.1 pipes, use the `libscemi21.so` and `libscemi21_dc.so` shared libraries and the `uatba.h` header file to implement the core functions of the pipe and thread-neutral C-interface for the testbench partition.

    The blocking interface functions have been built into the SCE-MI 2.1 libraries (`libscemi21` and `libscemi21_dc`). You do not need to add the blocking interface source code in the compilation.

© 2021 All Rights Reserved.

- For SCE-MI 2.2 pipes, use the `libscemi22.so`,`libscemi22_dc.so`, and `libscemi22_hmpi.so` shared libraries and the `uatba.h` header file to implement the core functions of the pipe and thread-neutral C-interface for the testbench partition.

  The blocking interface functions have been built into the SCE-MI 2.2 libraries (`libscemi22.so` and l`ibscemi22_dc.so)`. You do not need to add the blocking interface source code in the compilation.

The HMPI libraries for SCE-MI 2.0, 2.1, and 2.2 also have built-in blocking interface functions. The names of the HMPI libraries are as follows:

- SCE-MI 2.0: `libuatba_hmpi.so`
- SCE-MI 2.1: `libscemi21_hmpi.so`
- SCE-MI 2.2: `libscemi22_hmpi.so`

The "`_dc`" in the names of the libraries mentioned above refers to Direct C or direct access to C code.

The "`_dc.so`" libraries are used only for pure software simulation.

Caution

The HMI/HMPI feature is being deprecated in VXE 20.05 release. It will not be supported in future releases.

These core components are used with language-based adaptors provided in source form to implement blocking `send` and `receive` operations of the SCE-MI pipe. The following table lists the language adaptors.

Table 29-1 Language Adaptors

Language Adaptor	Description
`uatba_event.cpp` and `uatba_event.sv`	These source files are used to implement simulation thread suspension and continuation control from C-side. Include these files with all language adaptors based on the native simulator threads. See Executing the Simulator Thread on page 1344 for more information.

© 2021 All Rights Reserved.

Language Adaptor	Description
`uatba_scemi.c` and `uatba_scemi.h`	These source files are used to implement the SCE-MI blocking C-interfaces. See SCE-MI Pipes Blocking C Interfaces on page 1348 for more information.
`uatba_cpp.h`	This source file is used to implement the C++ interface adaptor. See SCE-MI Pipes C++ Interface on Simulator Thread on page 1348 for more information.
`uatba_sc.h` and `uatba_sc.cpp`	These source files are used to implement the SystemC interface adaptor. See SCE-MI Pipes SystemC Interface on SystemC Thread on page 1349 for more information.
`uatba_sv.sv` and `uatba_class.svh`	These source files are used to implement the SystemVerilog interface adaptor. See SCE-MI Pipes SVTB Interface on Simulator Thread on page 1350 for more information.
`uatba_sn.e` and `svtb.sv`	These source files are used to implement the Specman interface adaptor. See SCE-MI Pipes Specman Interface on Specman Thread on page 1351 for more information.

Testbench-Side Thread-Neutral C Interfaces

Based on the current SCE-MI pipes definition, following are the thread-neutral C-interface functions supported on the testbench side:

Common Configuration and Query Functions:

- `void *scemi_pipe_c_handle(const char *endpoint_path );`
- `typedef void (*scemi_pipe_notify_callback)(void *context );`
- `void *scemi_pipe_get_notify_context(void *pipe_handle );`
- `int scemi_pipe_get_bytes_per_element(void *pipe_handle );`
- `int scemi_pipe_get_depth(void *pipe_handle );`

© 2021 All Rights Reserved.

SCE-MI 2.0 Functions:

- `void scemi_pipe_set_notify_callback(void *pipe_handle, scemi_pipe_notify_callback notify_callback, void *notify_context);`
- `int scemi_pipe_get_direction(void *pipe_handle);`
- `void scemi_pipe_set_eom_auto_flush(void *pipe_handle, svBit enabled);`

SCE-MI 2.1 and 2.2 Functions:

- `typedef void *scemi_pipe_notify_callback_handle;`
- `scemi_pipe_notify_callback_handle scemi_pipe_set_notify_callback(void *pipe_handle, scemi_pipe_notify_callback notify_callback, void *notify_context, int callback_threshold);`
- `void scemi_pipe_clear_notify_callback(scemi_pipe_notify_callback_handle callback_handle);`
- `svBit scemi_pipe_set_eom_auto_flush(void *pipe_handle, svBit enabled );`
- `svBit scemi_pipe_get_direction(void *pipe_handle);`
- `void scemi_pipe_put_user_data(void *pipe_handle, void *user_key, void *user_data);`
- `void *scemi_pipe_get_user_data(void *pipe_handle, void *user_key);`
- `svBit scemi_pipe_c_in_flush_state(void *pipe_handle);`

SCE-MI Reference Manual 2.0 defines the `set_depth` function shown below, but this function has no effect in the VXE implementation of SCE-MI pipes because the depth of the buffer is defined at the HDL-side.

```
void scemi_pipe_set_depth(void *pipe_handle, int num_elements );
```

A warning message is printed out when this function is used.

SCE-MI input pipe non-blocking interface:

```
int scemi_pipe_c_try_send(void *pipe_handle, int byte_offset, int
                          num_elements, const svBitVecVal *data, svBit eom );
```


© 2021 All Rights Reserved.

```
int scemi_pipe_c_can_send(void *pipe_handle );
void scemi_pipe_c_try_flush(void *pipe_handle);
```

SCE-MI output pipe non-blocking interface:

```
int scemi_pipe_c_try_receive(void *pipe_handle, int byte_offset, int
                             num_elements, svBitVecVal *data, svBit *eom );
int scemi_pipe_c_can_receive(void *pipe_handle);
```

In addition to the `svBitVector`-based interfaces defined in the SCE-MI standard, IXCOM supports the unpacked open byte array interfaces. To directly transfer `unpacked open byte array` into the pipe, use the `uatba_try_send_ba` function. This function has the same semantics as the `scemi_pipe_c_try_send` function except that the C-char array is used as the data type.

```
int uatba_try_send_ba(void *pipe_handle, int byte_offset,
                      int num_bytes, const char *data, int eom);
```

This function returns the number of requested bytes that are actually sent. Similarly, the `uatba_try_receive_ba` function returns the number of bytes actually received. The syntax of the `uatba_try_receive_ba` function is given below:

```
int uatba_try_receive_ba(void *pipe_handle, int byte_offset,
                         int num_bytes, char *data, int *eom);
```

DUT-Side HDL Interface

Based on the current SCE-MI pipe definitions, the DUT-side interfaces are implemented as Verilog modules as given below:

```
interface scemi_input_pipe;
        parameter BYTES_PER_ELEMENT = 1;
        parameter PAYLOAD_MAX_ELEMENTS = 1;
        parameter BUFFER_MAX_ELEMENTS = 16;
        `ifdef DEFER_MODE
             parameter VISIBILITY_MODE = 2;  // 1:immediate 2:deferred
        `else
           `ifdef SCEMI21
              parameter VISIBILITY_MODE = 0;
           `else
              parameter VISIBILITY_MODE = 1;
           `endif
        `endif
        `ifdef SCEMI21
             parameter NOTIFICATION_THRESHOLD = BUFFER_MAX_ELEMENTS;
        `endif
        `ifdef HMPI
             parameter hmpi = 1;
        `else
             parameter hmpi = 0;
        `endif
        event sn_notify;
        task receive(input int num_elements, output int num_elements_valid,
             output bit [PAYLOAD_MAX_BITS-1:0] data, output bit eom );
             function int try_receive( int byte_offset, int num_elements,
```


© 2021 All Rights Reserved.

```
            output bit [PAYLOAD_MAX_BITS-1:0] data, output bit eom );
            function int can_receive(); // return: #elements that can be received
endinterface
interface scemi_output_pipe;
        parameter BYTES_PER_ELEMENT = 1;
        parameter PAYLOAD_MAX_ELEMENTS = 1;
        parameter BUFFER_MAX_ELEMENTS = 16;
        `ifdef DEFER_MODE
           parameter VISIBILITY_MODE = 2;  //2:deferred
        `else
        `ifdef SCEMI21
           parameter VISIBILITY_MODE = 0;
        `else
           parameter VISIBILITY_MODE = 1;
        `endif
        `endif
        `ifdef SCEMI21
           parameter NOTIFICATION_THRESHOLD = BUFFER_MAX_ELEMENTS;
        `endif
        `ifdef HMPI
           parameter hmpi = 1;
        `else
           parameter hmpi = 0;
        `endif
        event sn_notify;
        task send(input int num_elements, input bit [PAYLOAD_MAX_BITS-1:0] data,
                  input bit eom );
        task flush;
            function int try_send( int byte_offset, int num_elements,
                                   bit [PAYLOAD_MAX_BITS-1:0] data,
                                   input bit eom );
            function int can_send();
endinterface
```

VXE exports a special event signal, `sn_notify` in the above modules for Specman.

Caution

The HMI/HMPI feature is being deprecated in VXE 20.05 release. It will not be supported in future releases.

Executing the Simulator Thread

Implementation of the SCE-MI pipes blocking interface needs to be based on the thread system running on the testbench side. For a simulation environment, the simulator thread is the thread system common to all the hardware verification languages supported by the simulator. Using the SystemVerilog DPI export task mechanism, the common simulation thread functions can be accessed from all the hardware verification languages through their C-interfaces.

© 2021 All Rights Reserved.

The `uatba_event.sv` defines the SystemVerilog DPI module function used by the SCE-MI pipes blocking interface to implement thread execution control.

```
module uatba_event;
    class uatba_ev;
    static uatba_ev eArr[];
    export "DPI-C" function int uatba_new_event(); // return event_id
    export "DPI-C" task uatba_trigger(int event_id);
    export "DPI-C" task uatba_wait(int event_id);
endmodule
```

Specify this module as the top-level module during the elaboration phase. This module contains an open array of SystemVerilog events. To wait for a SCE-MI pipes event, a blocking interface uses the following sequence:

1. The `uatba_new_event()` function is called to allocate an event and set the returned *`event_id`* as the *`notifiy_context`*.
2. The `uatba_trigger()` function is set as the `scemi_pipe_notify_callback` function.
3. The `uatba_wait_event(event_id)` function is called to suspend its execution and wait for the callback-notification event.

© 2021 All Rights Reserved.

Reset of SCE-MI Pipes

In the software-simulation and simulation-acceleration environments, at times you might need to reset the simulator. For example you might have encountered many errors and need to rerun a test. You might even need to reset only the SCE-MI pipes, but not reset the simulator when issues like protocol errors are detected. Correspondingly, there are two kinds of SCE-MI Reset in SCE-MI pipes-based software-simulation and simulation-acceleration environments:

- Simulator reset

  You issue the simulator-reset manually by using the `reset` command. With the execution of the `reset` command, the hardware modules are re-instantiated and registers are set to known values, the SCE-MI simulator-reset software function is called to clean up and release the SCE-MI resources. When you issue the `run` command on the CLI to rerun the test, the software modules are re-generated or re-constructed on rerun. Therefore, the SCE-MI pipes are re-bound and re-registered by user code.

- Protocol reset

  The protocol-reset is a C function to be called to reset SCE-MI pipes when the protocol errors have been detected. You can call the protocol-reset function, `scemi_reset()` in your code either from the testbench side when protocol errors are detected, or from the DUT-side when a hardware-reset signal is asserted. To initiate the protocol-reset from the hardware side, you need to call the imported DPI function `scemi_reset()` when the hardware-reset signal is asserted.

To support both kinds of reset, the VXE software has provided an extension to the SCE-MI 2.0, 2.1, and 2.2 versions. This extension provides the required reset capability, while ensuring that the SCE-MI implementation in VXE is fully compatible with the SCE-MI standard.

Note: It is your responsibility to force BFMs attached to input or output pipes to an idle state after the use of `scemi_reset`. A state machine attached to an input pipe could use the `num_elements_valid == 0` to test for a `scemi_reset`; however, an output pipe does not have a similar indicator. Therefore, the proxy model of the output pipe will need to force the BFM to an idle state, or set a flag prior to the `scemi_reset`, which can be tested when the `send()` function returns.

The SCE-MI simulator-reset function is automatically called by the CLI `reset` command. This section focuses on the requirements for SCE-MI protocol-reset function to help you understand the functionality and how to use it.

© 2021 All Rights Reserved.

SCE-MI Protocol-Reset Functionality

The SCE-MI protocol-reset feature is available with both SCE-MI 2.0 (`libuatba.so`), SCE-MI 2.1 (`libscemi21.so`), and SCE-MI 2.2 (`libscemi22.so` and `libscemi22_dc.so`) software libraries provided with the VXE software installation.

The SCE-MI protocol-reset function is implemented as a C function that you can call from the testbench or from a DUT module by using an imported DPI call.

```
int scemi_reset()
```

This function always returns `0`.

When the `scemi_reset()` function is called, all the SCE-MI pipe registers in the hardware and the variables in the software are initialized to known values. All buffers are emptied, data is discarded, and all blocked functions are terminated.

Note: If the simulator is reset, Xcelium needs to regenerate or reconstruct the software modules at the time of the rerun.

Input Pipes

Following blocked functions are terminated:

- `scemi_pipe_c_send()` returns. Sent data discarded.
- `scemi_pipe_c_flush()` returns. Flush action aborted.
- `receive()` returns with `num_elements_valid = 0`.

Output Pipes

Following blocked functions are terminated:

- `scemi_pipe_c_receive()` returns with `num_elements_valid = 0`.
- `send()` returns. Sent data discarded.
- `flush()` returns. Flush action aborted.

Input-Output Pipes

- All callback registrations are maintained.
- All autoflush states are maintained.

© 2021 All Rights Reserved.

- All User Data is maintained.
- All pending callbacks are discarded.
- All data buffered in the pipes is discarded.

Using Language-Based Adaptors

The VXE TBA solution supports the following language-based adaptors:

- SCE-MI Pipes Blocking C Interfaces
- SCE-MI Pipes C++ Interface on Simulator Thread
- SCE-MI Pipes SystemC Interface on SystemC Thread
- SCE-MI Pipes SVTB Interface on Simulator Thread
- SCE-MI Pipes Specman Interface on Specman Thread

SCE-MI Pipes Blocking C Interfaces

This interface uses the `uatba_event.sv` function for thread execution control. See Executing the Simulator Thread on page 1344 for more information. The caller C-thread must be a native thread under simulator management. For example, the C thread can be created by calling a DPI import function from HDL-side or calling a DPI export function to `fork-join_none` thread from C.

Blocking input pipe interface:

```
void scemi_pipe_c_send(void *pipe_handle, int num_elements,
                       const svBitVecVal*data, svbit eom );
void scemi_pipe_c_flush(void *pipe_handle );
void uatba_send_ba(void *pipe_handle, int num_bytes,
                   const char *data, int eom);
```

Blocking output pipe interface:

```
void scemi_pipe_c_receive(void *pipe_handle, int num_elements, int
                          *num_elements_valid, svBitVecVal *data, svbit *eom );
void uatba_receive_ba(void *pipe_handle, int num_bytes, int *num_bytes_valid,
                      char *data, int *eom );
```

SCE-MI Pipes C++ Interface on Simulator Thread

The C++ interface provides an object-oriented interface shell on top of the native SCE-MI C interface. This interface also assumes that the caller thread is a simulator-native thread.

© 2021 All Rights Reserved.

```
class tba_c_pipe{
    public:
        tba_c_pipe(const char *path_name);
        void set_eom_auto_flush(int enabled ); // for SCE-MI 2.0
        svBit set_eom_auto_flush(svBit enabled );// for SCE-MI 2.1 and 2.2
        int get_bytes_per_element();
        int get_depth();
                        };
class tba_c_in_pipe : public tba_c_pipe {
    public:
        tba_c_in_pipe(const char *path_name);
        int can_send();
        int try_send(int byte_offset, int num_elements,
                     const svBitVecVal *data, svBit eom );
        int try_send_ba(int byte_offset, int num_bytes,
                        const char *data, int eom );
        void try_flush();
        void send(int num_elements, const svBitVecVal *data, svbit eom );
        void send_ba(int num_bytes, char *data, int eom);
        void flush();
                                                }
class tba_c_out_pipe : public tba_c_pipe {
public:
        tba_c_out_pipe(const char *path);
        int can_receive();
        int try_receive(int byte_offset, int num_elements,
                        svBitVecVal *data, svBit *eom );
        int try_receive_ba(int byte_offset, int num_bytes,
                           char *data, int *eom );
        void receive(int num_elements, int *num_elements_valid,
                     svBitVecVal *data, svbit *eom );
        void receive_ba(int num_bytes, int *num_bytes_valid,
                        char *data, int *eom );
        }
```

SCE-MI Pipes SystemC Interface on SystemC Thread

This interface is identical to the C++ interface except for the class name. However, for implementation, this interface uses only the SystemC event classes and `wait` functions and does not use the event objects from `uatba_event.sv`. This interface library can be used with OSCI-based SystemC verification environment without an HDL simulator. For simulator-based SystemC verification environment, use either the C++ interface on simulator thread or this SystemC interface on SystemC thread.

```
class tba_sc_pipe {
    tba_sc_pipe(const char *path_name);
    <Same_as_tba_c_pipe>;
    }
class tba_sc_in_pipe : public tba_sc_pipe {
    public:tba_sc_in_pipe(const char *path_name);
    <Same_of_tba_c_in_pipe>;}
class tba_sc_out_pipe : public tba_sc_pipe {
```


© 2021 All Rights Reserved.

```
    public:tba_sc_out_pipe(const char *path_name);
    <Same_as_tba_c_outpipe>;
}
```

SCE-MI Pipes SVTB Interface on Simulator Thread

The SystemVerilog Testbench (SVTB) interface to the SCE-MI pipes channel is implemented as the SystemVerilog interface with similar interface functions and tasks as defined in C++.

Xcelium does not support `string` and `open` array in interface task calls. Therefore, two wrapper classes for `path_name` and `byte_array` should be specified in a separate SystemVerilog header file, `uatba_class.svh`.

```
class path_name;
      string s;
      function
         new (string path);
         s = path;
      endfunction
endclass
class byte_array;
      byte a[];
      function new(int size);
        a = new[size];
      endfunction

endclass
```

The SystemVerilog interfaces are defined in the `uatba_sv.sv` file:

```
`include uatba_class.sv
interface tba_sv_in_pipe;
     parameter PAY_LOAD_MAX_BITS = 256;
     function void connect(path_name pname);
     function void set_eom_auto_flush(int enabled);
     function int get_bytes_per_element();
     function int get_depth();
     function int can_send();
     function int try_send(input int byte_offset, input int num_elements,
                           input bit [PAY_LOAD_MAX_BITS-1:0] data, input bit eom);
     function int try_send_ba(input int byte_offset,input int num_bytes,
                              input byte_array data,input int eom);
     function int try_flush();
     task send(input int num_elements, input bit [PAY_LOAD_MAX_BITS-1:0] data,
               input bit eom);
     task send_ba(input int num_bytes, input byte_array data, input int eom);
     task flush();
endinterface
interface tba_sv_out_pipe;
     parameter PAY_LOAD_MAX_BITS = 256;
     function void connect(path_name pname);
     function void set_eom_auto_flush(int enabled);
     function int get_bytes_per_element();
     function int get_depth();
     function int can_send();
     function int can_receive();
     function int try_receive(input int byte_offset,
```


© 2021 All Rights Reserved.

```
                              input int num_elements,
                              output bit [PAY_LOAD_MAX_BITS-1:0] data,
                              output bit eom);
     function int try_receive_ba(input int byte_offset, input int num_bytes,
                                 input byte_array data, output int eom);
     task receive(input int num_elements, output int num_elements_valid,
                  output bit [PAY_LOAD_MAX_BITS-1:0] data, output bit eom );
     task receive_ba(input int num_bytes, output int num_bytes_valid,
                     input byte_array data, output int eom );
endinterface
```

SCE-MI Pipes Specman Interface on Specman Thread

The Specman interface is implemented as an ***e*** unit with similar interface functions and tasks as defined in C++. The implementation of this interface is completely based on Specman SN-C interface to call the SCE-MI thread-neutral interface. The Specman interface also uses an external event port to get notification callbacks indirectly. It does not require any special foreign language adaptors.

```
unit tba_in_pipe {
     run() is also { create binding get_unit().full_hdl_path(); }
     sn_notify: in event_port is instance;
     keep sn_notify.hdl_path() == "sn_notify";
     set_eom_auto_flush(enabled: int) is { <implememtation> };
     get_bytes_per_element(): int is {<implememtation> };
     get_depth(): int is{<implememtation> };
     can_send(): int is {<implememtation> };
     try_send(byte_offset: int, num_elements: int, data: list of bit, eom: bit ):
              int is {<implememtation> };
     send(byte_offset: int, num_elements: int,
          data: list of bit, eom: *bit)
          @sys.any is {<implememtation> };
     flush() @ sys.any is {};
};
unit tba_out_pipe {
     run() is also { create binding get_unit().full_hdl_path();}
     sn_notify: in event_port is instance;
     keep sn_notify.hdl_path() == "sn_notify";
     set_eom_auto_flush(enabled: int) is {<implememtation> };
     get_bytes_per_element(): int is {<implememtation> };
     get_depth(): int is{<implememtation> };
     can_receive(): int is {<implememtation> };
     try_receive(byte_offset: int, num_elements: int,
                 data: list of bit, eom: *bit ): int is {<implememtation> };
     receive(num_elements: int,  num_elements_valid: *int,
             data: list of bit, eom: *bit)@sys.any is {<implememtation> };
};
```

Note: Due to the Specman `event_port` limitation, the `uatba_sn.e` blocking functions add one extra clock delay to the end of the blocking `wait` operation.

© 2021 All Rights Reserved.

Using Specific Libraries for Different SCE-MI Versions in Specman Designs

The Specman adaptor requires certain text macros to be defined in order for the adaptor to function properly with the SCE-MI 2.0, SCE-MI 2.1, and SCEMI-2.2 versions.

For SCE-MI 2.2 version with regular pipes, use the following compilation options:

- The `+define+SCEMI22` option of the `ixcom` and `xrun` commands
- The `-sndefine "SCEMI22=1"` option of the `ixcom` and `xrun` commands
- The `libscemi22.so` and `libscemi22_dc.so` libraries

For SCE-MI 2.1 version with regular pipes, use the following compilation options:

- The `+define+SCEMI21` option of the `ixcom` and `xrun` commands
- The `-sndefine "SCEMI21=1"` option of the `ixcom` and `xrun` commands
- The `libscemi21.so` and `libscemi21_dc.so` libraries

For SCE-MI 2.0 version, use `libuatba.so` and `libuatba_dc.so` libraries for regular pipes.

To instantiate regular pipes, use `scemi_input_pipe` and `scemi_output_pipe` interface names. The regular pipe interfaces are defined in `scemi_input_pipe.vp`, `scemi_output_pipe.vp`, and `scemi_pipe.vp` files.

Generating DUT Clocks with SCE-MI Pipes

To avoid race conditions between the testbench and DUT, non-blocking assignments should be used to generate clocks for DUT and blocking assignments should be used to generate clocks for the testbench.

For example, use the following code to generate clocks for the DUT:

```
reg clk = 0;
always #1 clk  <=  ~clk;
```

Use the following code to generate clocks for the testbench:

```
reg clk = 0;
always #1 clk = ~clk;
```


© 2021 All Rights Reserved.

Compiling and Running Designs with SCE-MI Pipes

The VXE implementation of SCE-MI pipes supports two utilities for compiling and running designs. These utilities are IXCOM (invoked using the `ixcom` command) and *xrun* (invoked using the `xrun` command). Correspondingly, at a high-level you can compile and run designs in two flows:

- Compile and run a design using IXCOM for software and hardware.
- Compile the design using *xrun*, that is without using IXCOM, and run the design in pure software-simulation mode.

The major differences between using IXCOM and *xrun* are the compile options and the shared libraries used with these two compilers.

- With *xrun*, the shared libraries used are:
  - For SCE-MI 2.2 - `libscemi22_dc.so`
  - For SCE-MI 2.1 - `libscemi21_dc.so`
  - For SCE-MI 2.0 - `libuatba_dc.so`
- With *ixcom*, the shared libraries used are:
  - For SCE-MI 2.2 - `libscemi22.so`
  - For SCE-MI 2.1 - `libscemi21.so`
  - For SCE-MI 2.0 - `libuatba.so`
- None of the compilation options specific to `ixcom` should be used when using *xrun* to compile a design targeted for the pure software-only mode.
- Additionally, to compile test cases using TBA Specman ***e*** adaptor (as described in SCE-MI Pipes Specman Interface on Specman Thread on page 1351):

  Specify the `-sndefine "NO_IXCOM=1"` option with the `xrun` command.
- For SA mode only, use the 1X mode for faster simulation. See Synchronizing the Fastest Design Clock with Emulator FCLK on page 1115 for more information about the implementation of 1X mode in SA mode.

Compiling and Running Designs in 32-Bit and 64-Bit Modes

This section describes the settings required for compiling and running TBA designs in 32-bit and 64-bit modes.

© 2021 All Rights Reserved.

To compile and run designs for 32-bit mode:

- Add ${~~XE_INSTALL_DIR~~IXCOM_INSTALL_HOME}/bin to the PATH variable
- Add ${~~XE_INSTALL_DIR~~IXCOM_INSTALL_HOME}/lib to the LD_LIBRARY_PATH variable

To compile and run designs for 64-bit mode:

- Add ${~~XE_INSTALL_DIR~~IXCOM_INSTALL_HOME}/bin/64bit to the PATH variable
- Add ${~~XE_INSTALL_DIR~~IXCOM_INSTALL_HOME}/lib/64bit to the LD_LIBRARY_PATH variable

Sample Makefiles

Sample Makefiles are provided in the VXE TBA solution to help you compile and run a design using IXCOM and *xrun*. These Makefiles can be accessed from the examples available in the ${~~XE_INSTALL_DIR~~IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm~~uatba~~ directory.

Makefile Targets

The following Makefile targets are used:

- comp_hw
- comp_sw
- run_hw
- run_sw
- comp_xrun
- run_xrun

Makefile Variables

The following variables are used to define your source code and the VXE TBA source code in the sample Makefile:

- V_SRC - defines all user *.v and *.sv modules
- C_SRC - defines all user *.c and *.cpp files

© 2021 All Rights Reserved.

- `VPLIB` - defines all VXE TBA `*.vp` files
- `SVLIB` - defines all VXE TBA `*.sv` files, VXE TBA `*.e` files, user testbench `*.sv` files, and user testbench `*.e` files
- `TBA_SRC` - defines all VXE TBA `*.cpp` and `*.c` files required for the design
- `CLIB` - defines the VXE TBA shared library, `libuatba.so` to be used with IXCOM
- `CLIB_DC` - defines the VXE TBA shared library, `libuatba_dc.so` to be used with *xrun*

Compiling and Running Designs

- To compile and run in hardware and software:

```
make comp_hw
make run_hw
make run_sw
```

- To compile and run in the software simulation mode only:

```
make comp_sw run_sw
```

- To compile and run in the software mode using *xrun* without using IXCOM:

```
make comp_xrun run_xrun
```

Compiling Designs

- To compile a SystemC based testbench **using ixcom flow**, first use `vlan` to compile Verilog-only files and `vhan` to compile VHDL source code. Then, use the `ixcom` command for compilation. Lastly, use the `xrun` command to elaborate:

```
comp_hw: clean comp_rtl_hw comp_tb
comp_sw: clean comp_rtl_sw comp_tb
comp_rtl_hw:
     vlan -32 +sv $(V_SRC) $(VPLIB)
     ixcom -32 -xmcompile +sv -ua -top dut +dut+dut

comp_rtl_sw
     vlan -32 +sv $(V_SRC) $(VPLIB)
     ixcom -32 -xmcompile +sv -top dut

comp_tb
     xrun -elaborate -TIMESCALE 1ns/1ns \
          -f xc_work/xrun.f $(CINC) $ (C_SRC) \
          -sysc -DPI -top sc_main $(SC_SRC) $(CLIB)
```

- To compile a SystemC based testbench **for pure software-only mode using** *xrun*:

```
comp_xrun:

    xrun -access +rw -sysc +sv -c +define+NO_IXCOM -top sc_main -top \
    dut $(V_SRC) $(VPLIB) $(CINC) $(C_SRC) $(SC_SRC) $(CLIB_DC) -end
```

Note: Some test cases using SystemC testbenches might hang if the

© 2021 All Rights Reserved.

`+scAllowSCFromPLI` option is not passed to the `xrun` command at run time. Cadence recommends that the `+scAllowSCFromPLI` option is passed to the `xrun` command when running a SystemC test case.

Note: To use void functions that do not return a zero as tasks, and call these as DPI tasks, it is necessary to use the `-dpi_void_task` option of the `xrun` command to avoid errors by *xrun* compiler.

- To compile a non-SystemC based Testbench **using ixcom**, use the `ixcom` command to compile the Verilog modules. Then, use the `xrun` command to elaborate the design.

```
comp_hw: clean comp_rtl_hw comp_tb
comp_sw: clean comp_rtl_sw comp_tb
comp_rtl_hw:
     ixcom -32 -xmcompile +sv $(V_SRC) $(VPLIB) -ua +dut+dut

comp_rtl_sw:
     ixcom -32 -xmcompile +sv $(V_SRC) $(VPLIB)

comp_tb:
     xrun -elaborate -TIMESCALE 1ns/1ns \
          -f xc_work/xrun.f $(CINC) $(C_SRC) \
          -top uatba_event $(SVLIB) $(TBA_SRC) $(CLIB)
```

- To compile a non-SystemC based Testbench for pure software-only mode **using** *xrun*:

```
comp_xrun:
     xrun -access +rw +define+NO_IXCOM -c +sv $(V_SRC) $(VPLIB) $(SVLIB) \|
          $(CINC) $(C_SRC) $(TBA_SRC) $(CLIB_DC)
```

Sample Makefile to Compile a Non-SystemC Testbench for 32-bit Mode

```
V_SRC = top.v
C_SRC = test.cpp
SRC_DIR = ${XE_INSTALL_DIR}/etc/tba
VPLIB = ${SRC_DIR}/sccmi_input_pipe.vp ${SRC_DIR}/sccmi_output_pipe.vp
SVLIB = ${SRC_DIR}/uatba_event.sv
TBA_SRC = $(SRC_DIR)/uatba_sccmi.c $(SRC_DIR)/uatba_event.cpp
CLIBDIR = ${XE_INSTALL_DIR}/lib
CLIB = $(CLIBDIR)/libuatba.so
CLIB_DC = $(CLIBDIR)/libuatba_dc.so
CINC = -I$(XE_INSTALL_DIR)/etc/tba

SRC_DIR = ${IXCOM_INSTALL_HOME}/share/vxe/etc/ixcom
VPLIB = ${SRC_DIR}/scm_pipe.sv
SVLIB =  ${SRC_DIR}/ixcom_sim.sv
TBA_SRC =
CLIBDIR = ${IXCOM_INSTALL_HOME}/tools.lnx86/lib/${BITS}bit
CLIB = $(CLIBDIR)/libscm.so
CLIB_DC = ${CLIB} ${SRC_DIR}/marg_sim.c
CINC = -I$(IXCOM_INSTALL_HOME)/share/vxe/etc/ixcom
comp_hw: clean comp_rtl_hw comp_tb
comp_sw: clean comp_rtl_sw comp_tb
comp_xrun: clean comp_xrun_all
comp_rtl_hw:
     ixcom -32 -xmcompile +sv $(V_SRC) $(VPLIB) -ua +dut+dut

comp_rtl_sw:
     ixcom -32 -xmcompile +sv $(V_SRC) $(VPLIB)
```


© 2021 All Rights Reserved.

```
comp_tb:
     xrun -elaborate -TIMESCALE 1ns/1ns \
          -f xc_work/xrun.f $(CINC) $(C_SRC) \
          -top uatba_event $(SVLIB) $(TBA_SRC) $(CLIB)

comp_xrun_all:
     xrun -access +rw +define+NO_IXCOM -c +sv $(V_SRC) $(VPLIB) \ $(SVLIB)
           $(CINC) $(C_SRC) $(TBA_SRC) $(CLIB_DC)

run_sw:
     xrun -R -input sw.run

run_xrun:
     xrun -R -input sw.run

run_hw:
     xrun -R input hw.run

clean:
     rm -rf INCA* WORK AxisWork xc_* *.o *.log *.so cds.lib hdl.var .design .key \
            .log .xe* CLOCK_SOURCES DBEngine.msg PDB QTDB cellList dbFiles tmp \
            xe.msg xeCompile.key
```

Using Makefile to Compile a Non-SystemC Testbench for 64-bit mode

Following are the differences between the 32-bit and 64-bit Makefile:

- The `CLIBDIR` environment variable is set to the 64-bit directory:

  `CLIBDIR = ${~~XE_INSTALL_DIR~~IXCOM_INSTALL_HOME}/lib/64bit`

- The `ixcom` command is replaced with `ixcom -64`.
- The `vlan` command is replaced with `vlan -64`.
- The `xrun` command is replace with `xrun -64`.

Using Makefile to Compile SCE-MI ~~2.1~~ Files

To simplify the VXE-TBA compilation flow, certain compilation option files are provided for the different compilation flows. These compilation option files are located in the `${IXCOM_INSTALL_HOME}/share/vxe/etc/ixcom` directory and have `.args` as the filename extension.

Following is an example of the Makefile using the compilation option files to compile a basic testcase that contains only one DUT file named top.v and one C testbench file named `test.cpp`.

```
CLIBDIR = ${CLIBDIR_PATH}/lib/${BITS}bit
CLIBDCDIR = ${CLIBDCDIR_PATH}/lib/${BITS}bit


VPLIB = ${SRC_DIR}/scm_pipe.sv
SVLIB = ${SRC_DIR}/ixcom_sim.sv
```


© 2021 All Rights Reserved.

```
C_MARG = ${SRC_DIR}/marg_sim.c

CLIB = $(CLIBDIR)/libscm.so
CLIB_DC = $(CLIBDCDIR)/libscm.so


CINC = -I${SRC_DIR}


comp_rtl_hw:
   $(IXCOM) -${BITS} -nccompile +sv $(V_SRC) $(VPLIB) +incdir+${SRC_DIR} -ua\
   +dut+user
comp_rtl_sw:
   $(IXCOM) -${BITS} -nccompile +sv $(V_SRC) $(VPLIB)
comp_tb:
   $(RUN) -${BITS} -elaborate -TIMESCALE 1ns/1ns \
   -f ${IXCOM_HOME}/share/uxe/etc/ixcom/scm_tb.args $(CINC) \
    $(C_SRC) $(SVLIB) $(CLIB)
```

~~To simplify the VXE-TBA compilation flow, certain compilation option files are provided for the different compilation flows. These compilation option files are located in the `/etc/tba` directory and have `.args` as the filename extension.~~

~~Following is an example of the Makefile using the compilation option files to compile a basic testcase that contains only one DUT file named top.v and one C testbench file named test.cpp.~~

```
comp_rtl_hw:
     ixcom -xmcompile -F ${XE_INSTALL_DIR}/etc/tba/ua.args
            -ua top.v +dut+dut
comp_tb:
     xrun -f ${XE_INSTALL_DIR}/etc/tba/tb.args test.cpp
```

~~Using compilation options file eliminates the need for typing out the following options:~~

```
comp_rtl_hw:
     ixcom -xmcompile +sv -ua +define+SCEMI21 \
            ${XE_INSTALL_DIR}/etc/tba/scemi_pipe21.svp top.v +dut+dut
comp_tb:
     xrun -elaborate -DSCEMI21 -TIMESCALE 1ns/1ns -f xc_work/xrun.f \
          -I${XE_INSTALL_DIR}/etc/tba test.cpp -top uatba_event \
            ${XE_INSTALL_DIR}/etc/tba/uatba_event.cpp \
            ${XE_INSTALL_DIR}/etc/tba/uatba_event.sv \
            ${XE_INSTALL_DIR}/lib/${BITS}bit/libscemi21.so
```

~~The sample Makefile files for different language-based testcases using SCE-MI 2.1 are available in the `${XE_INSTALL_DIR}/gift/tutorial/uatba/scemi21` and `${XE_INSTALL_DIR}/gift/tutorial/uatba/scemi21_hmpi` directories. As evident from the Makefiles in these examples, these are identical except that the compilation option file used for compiling the testbench is different. To disable compilation for hardware, remove the `-ua` option from the original `ixcom` or `xrun` command.~~

© 2021 All Rights Reserved.

Caution

~~***The HMI/HMPI feature is being deprecated in VXE 20.05 release. It will not be supported in future releases.***~~

Using Compilation Option Files

The names of the compilation option files and their usages are summarized in the following table~~s~~.

Table 29-2 Compilation Option Files for IXCOM Flow

Filename	Used with
`scm_ixcom.args`	`ixcom -f` command to compile DUT files
`scm_tb.args`	`xrun -f` command to compile testbench files and link with IXCOM DUT
`scm_tb.args`	For pure xrun, use the `xrun -f` command to compile testbench and DUT files
`scm_pvcs.args`	For pure VCS, use the `vcs -file` command to compile testbench and DUT files
~~`ua.args`~~	~~`ixcom` command to compile DUT files~~
~~`ixcom.args`~~	~~`xrun` command to invoke `ixcom` command for compiling the DUT files~~
~~`vlan.args`~~	~~`xrun` command to invoke `vlan` command for compiling the mixed-language DUT files~~
~~`tb.args`~~	~~`xrun` command to compile testbench files using C/C++/SC and elaborate the design~~
~~`tb-sv.args`~~	~~`xrun` command to compile testbench files using SystemVerilog and elaborate the design~~
~~`tb-e.args`~~	~~`xrun` command to compile testbench files using Specman ***e*** and elaborate the design~~
~~`tb-hmpi.args`~~	~~`xrun` command to compile the HMPI testbench files and elaborate the design~~

© 2021 All Rights Reserved.

Table 29-3 ~~Compilation Option Files for Software Simulation Flow Using the xrun Command~~

~~Filename~~	~~Used with~~
~~xmsim.args~~	~~xrun command to compile C/C++/SC based testbench files and all DUT files~~
~~xmsim-sv.args~~	~~xrun command to compile SystemVerilog based testbench files and all DUT files~~
~~xmsim-e.args~~	~~xrun command to compile Specman *e* based testbench files and all DUT files~~

~~Using Makefile to Compile SCE-MI 2.2 Files~~

~~To simplify the VXE-TBA compilation flow, certain compilation option files are provided for the different compilation flows. These compilation option files are located in~~ ~~`<XE_INSTALL_DIR>/etc/tba`~~ ~~directory and have~~ ~~`.args`~~ ~~as the filename extension.~~

~~Following is an example of the Makefile using the compilation option files to compile a basic testcase that contains only one DUT file named top.v and one C testbench file named test.cpp.~~

```
comp_rtl_hw:
      ixcom -xmcompile -F ${XE_INSTALL_DIR}/etc/tba/ua22.args \
            -ua top.v +dut+dut

comp_tb:
      xrun -f ${XE_INSTALL_DIR}/etc/tba/tb22.args test.cpp
```

~~Using compilation options file eliminates the need for typing out the following options:~~

```
comp_rtl_hw:
      ixcom -xmcompile +sv -ua \
            +define+SCEMI22 \
            ${XE_INSTALL_DIR}/etc/tba/scemi_pipe22.svp \
            top.v +dut+dut

comp_tb:
      xrun -elaborate \
            -DSCEMI21 \
            -TIMESCALE 1ns/1ns \
            -f xc_work/xrun.f \
            -I${XE_INSTALL_DIR}/etc/tba test.cpp \
            -top uatba_event \
            ${XE_INSTALL_DIR}/etc/tba/uatba_event.cpp \
            ${XE_INSTALL_DIR}/etc/tba/uatba_event.sv \
            ${XE_INSTALL_DIR}/lib/${BITS}bit/libscemi22.so
```


© 2021 All Rights Reserved.

~~The sample Makefile files for different language-based testcases using SCE-MI 2.2 are available in the `${XE_INSTALL_DIR}/gift/tutorial/uatba/sccmi22` and `${XE_INSTALL_DIR}/gift/tutorial/uatba/sccmi22_hmpi` directories. As evident from the Makefiles in these examples, these are identical except that the compilation option file used for compiling the testbench is different. To disable compilation for hardware, remove the `-ua` option from the original `ixcom` or `xrun` command.~~

~~*Using Compilation Option Files*~~

~~The names of the compilation option files and their usages are summarized in the following tables.~~

Table 29-4 ~~Compilation Option Files for IXCOM Flow~~

~~Filename~~	~~Used with~~
~~`ua22.args`~~	~~`ixcom` command to compile DUT files~~
~~`ixcom22.args`~~	~~`xrun` command to invoke `ixcom` command for compiling the DUT files~~
~~`vlan22.args`~~	~~`xrun` command to invoke `vlan` command for compiling the mixed-language DUT files~~
~~`tb22.args`~~	~~`xrun` command to compile testbench files using C/C++/SC and elaborate the design~~
~~`tb-sv22.args`~~	~~`xrun` command to compile testbench files using SystemVerilog and elaborate the design~~
~~`tb-e22.args`~~	~~`xrun` command to compile testbench files using Specman ***e*** and elaborate the design~~
~~`tb-hmpi22.args`~~	~~`xrun` command to compile the HMPI testbench files and elaborate the design~~

Table 29-5 ~~Compilation Option Files for Software Simulation Flow Using the xrun Command~~

~~Filename~~	~~Used with~~
~~`xmsim22.args`~~	~~`xrun` command to compile C/C++/SC based testbench files and all DUT files~~

© 2021 All Rights Reserved.

~~Filename~~	~~Used with~~
~~`xmsim-sv22.args`~~	~~`xrun` command to compile SystemVerilog based testbench files and all DUT files~~
~~`xmsim-e22.args`~~	~~`xrun` command to compile Specman ***e*** based testbench files and all DUT files~~

Note: ~~Compilation argument files for SCEMI 2.1 are also included. The files are identical in name and function and do not include '22' in the file name. For example, the `tb.args` file is the SCEMI 2.1 version of the `tb22.args` file.~~

© 2021 All Rights Reserved.

Examples: Implementing SCE-MI Pipes

To illustrate the implementation of the SCE-MI pipes, the following examples are provided in the IXCOM~~VXE~~ installation directory.

- ~~SCE-MI 2.0 Examples~~
- ~~SCE-MI 2.1 Examples~~
- ~~SCE-MI 2.2 Examples~~

~~SCE-MI 2.0 Examples~~

- SystemC Testbench Using C Interface Examples
- C++ Testbench Using C Interface Examples
- C Testbench Using C Interface Examples
- SystemC Testbench Using SystemC Interface Examples
- C++ Testbench Using C++ Interface Examples
- SystemVerilog Testbench Using SVTB Interface Examples
- ~~Specman e Testbench Using Specman e Adaptor Interface Examples~~
- HMPI Examples: C, OSCI SystemC, and SystemVerilog Testbenches Using HMPI Thread

SystemC Testbench Using C Interface Examples

- Blocking Interface of Input Pipe with SystemC Testbench

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_c/sysc/input_pipe_blk_dpi_sysc~~

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_c/sysc/input_pipe_blk_dpi_sysc

- Non-blocking Interface of Input Pipe with SystemC Testbench

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_c/sysc/input_pipe_nb_dpi_sysc~~

  $IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_c/sysc/input_pipe_nb_dpi_sysc

- Blocking Interface of Output Pipe with SystemC Testbench

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_c/sysc/output_pipe_blk_dpi_sysc~~

© 2021 All Rights Reserved.

${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_c/sysc/output_pipe_blk_dpi_sysc

- Non-blocking Interface of the Output Pipe with SystemC Testbench

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_c/sysc/output_pipe_nb_dpi_sysc~~

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_c/sysc/output_pipe_nb_dpi_sysc

- Blocking Interface of Input Pipe with SystemC Testbench and VPI Functions

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_c/sysc/input_pipe_blk_export_vpi_sys~~c

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_c/sysc/input_pipe_blk_export_vpi_sysc

- marg API Functions with SystemC Testbench for Input Pipe

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_c/sysc/input_marg_export_sysc~~

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_c/sysc/input_marg_export_sysc

- marg API Functions with SystemC Testbench for Output Pipe

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_c/sysc/output_marg_export_sysc~~

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/uatba_c/sysc/output_marg_export_sysc

The above examples use the following components:

Pipes	
SCE-MI Input pipe, blocking interface	`input_pipe_blk_dpi_sysc`
SCE-MI Input pipe, non-blocking interface	`input_pipe_nb_dpi_sysc`
SCE-MI output pipe, blocking interface	`output_pipe_blk_dpi_sysc`
SCE-MI output pipe, non-blocking interface	`output_pipe_nb_dpi_sysc`
DPI	
Import DPI	`input_pipe_blk_dpi_sysc`
Export DPI	`input_pipe_blk_dpi_sysc`
Cross module variable access API	
export_read, export_deposit	`input_pipe_blk_export_vpi_sysc`
marg C-interface functions	`input_marg_export_sysc`, `output_marg_export_sysc`

© 2021 All Rights Reserved.

C++ Testbench Using C Interface Examples

- Non-blocking Interface of Input Pipe with C++ Testbench

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_c/cpp/input_pipe_nb_dpi_cpp~~

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_c/cpp/input_pipe_nb_dpi_cpp

- Blocking Interface of Input Pipe with C++ Testbench

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_c/cpp/input_pipe_blk_dpi_cpp~~

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_c/cpp/input_pipe_blk_dpi_cpp

- Non-blocking Interface of the Output Pipe with C++ Testbench with DPI

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_c/cpp/output_pipe_nb_dpi_cpp~~

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_c/cpp/input_pipe_blk_dpi_cpp

- Blocking Interface of the Output Pipe with C++ Testbench with DPI

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_c/cpp/output_pipe_blk_dpi_cpp~~

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_c/cpp/output_pipe_blk_dpi_cpp

- marg API Functions with C++ Testbench for Input Pipe

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_c/cpp/input_marg_export_cpp~~

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_c/cpp/input_marg_export_cpp

- marg API Functions with C++ Testbench for Output Pipe

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_c/cpp/output_marg_export_cpp~~

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_c/cpp/output_marg_export_cpp

- Blocking Interface of Input Pipe with C++ Testbench and VPI Functions

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_c/cpp/input_pipe_blk_vpi_cpp~~

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_c/cpp/input_pipe_blk_export_vpi_cpp

The above examples use the following components:

Pipes

© 2021 All Rights Reserved.

SCE-MI Input pipe, blocking interface	`input_pipe_blk_dpi_cpp`
SCE-MI Input pipe, non-blocking interface	`input_pipe_nb_dpi_cpp`
SCE-MI output pipe, blocking interface	`output_pipe_blk_dpi_cpp`
SCE-MI output pipe, non-blocking interface	`output_pipe_nb_dpi_cpp`
DPI	
Import DPI	`input_pipe_blk_dpi_cpp`
Export DPI	`input_pipe_blk_dpi_cpp`
Cross module variable access API	
export_read, export_deposit	`input_pipe_blk_export_vpi_cpp`
marg C-interface functions	`input_marg_export_cpp`, `output_marg_export_cpp`

C Testbench Using C Interface Examples

- Non-blocking Interface of Input and Output Pipe with C Testbench

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_c/c/input_output_pipe_nb_c~~

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_c/c/input_output_pipe_nb_c

- Blocking Interface of Input and Output Pipe with C Testbench

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_c/c/input_output_pipe_blk_c~~

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_c/c/input_output_pipe_blk_c

The above examples use the following components:

Pipes	
SCE-MI Input pipe and Output_pipe blocking interface	`input_output_pipe_blk_c`
SCE-MI Input pipe and Output_pipe non-blocking interface	`input_output_pipe_nb_c`

SystemC Testbench Using SystemC Interface Examples

- Non-blocking Interface of Input and Output Pipe with SystemC Testbench

© 2021 All Rights Reserved.

~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_sc/input_output_pipe_nb_sysc~~

${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_base/io_pipe_nb_sysc

- Blocking Interface of Input and Output Pipe with SystemC Testbench

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_sc/input_output_pipe_blk_sysc~~

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_base/io_pipe_blk_sysc

The above examples use the following components:

Pipes

SCE-MI Input pipe and Output_pipe blocking interface	`input_output_pipe_blk_sysc`
SCE-MI Input pipe and Output_pipe non-blocking interface	`input_output_pipe_nb_sysc`

C++ Testbench Using C++ Interface Examples

- Non-blocking Interface of Input and Output Pipe with C++ Testbench

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_cpp/input_output_pipe_nb_cpp~~

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_base/io_pipe_nb_cpp

- Blocking Interface of Input and Output Pipe with C++ Testbench

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_cpp/input_output_pipe_blk_cpp~~

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_base/io_pipe_blk_cpp

The above examples use the following components:

Pipes

SCE-MI Input pipe and Output_pipe blocking interface	`input_output_pipe_blk_cpp`
SCE-MI Input pipe and Output_pipe non-blocking interface	`input_output_pipe_nb_cpp`

SystemVerilog Testbench Using SVTB Interface Examples

- Non-blocking Interface of Input and Output Pipe with SystemVerilog Testbench

© 2021 All Rights Reserved.

~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_sv/input_output_pipe_nb_sv~~

${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/uatba_sv/io_pipe_bits_nb_sv

- Blocking Interface of Input and Output Pipe with SystemVerilog Testbench

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_sv/input_output_pipe_blk_sv~~

  ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/uatba_sv/io_pipe_bits_blk_sv

The above examples use the following components:

Pipes	
SCE-MI Input pipe and Output_pipe blocking interface	~~input_output_pipe_blk_sv~~io_pipe_bits_nb_sv
SCE-MI Input pipe and Output_pipe non-blocking interface	~~input_output_pipe_nb_sv~~io_pipe_bits_blk_sv

~~**Specman *e* Testbench Using Specman *e* Adaptor Interface Examples**~~

- ~~Blocking Interface of Input and Output Pipe with Specman *e* Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_e/input_output_pipe_blk_e~~

- ~~Non-blocking Interface of Input and Output Pipe with Specman *e* Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/uatba_e/input_output_pipe_nb_e~~

~~The above examples use the following components:~~

~~**Pipes**~~	
~~SCE-MI Input pipe and Output_pipe blocking interface~~	~~input_output_pipe_blk_e~~
~~SCE-MI Input pipe and Output_pipe non-blocking interface~~	~~input_output_pipe_nb_e~~

HMPI Examples: C, OSCI SystemC, and SystemVerilog Testbenches Using HMPI Thread

- ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/hmpi/hm_c_pt~~

© 2021 All Rights Reserved.

- ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/hmpi/hm_c_sv~~
- ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/hmpi/hm_concurrent_pt~~
- ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/hmpi/hm_osci_sc~~
- ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/hmpi/hm_osci_scemi_1_2~~
- ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/hmpi/hm_sv_sv~~
- ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/hmpi/hm_sv_sv_no_clk~~
- ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/hmpi/hm_sv_sv_schd~~
- ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_hmpi/hm_c_pt
- ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_hmpi/hm_c_sv
- ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_hmpi/hm_concurrent_pt
- ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_hmpi/hm_osci_sc
- ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_hmpi/hm_osci_scemi_1_2
- ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_hmpi/hm_sv_sv
- ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_hmpi/hm_sv_sv_no_clk
- ${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_hmpi/hm_sv_sv_schd

Caution

The HMI/HMPI feature is being deprecated in VXE 20.05 release. It will not be supported in future releases.

© 2021 All Rights Reserved.

Other Examples

- Blocking Interface of Input and Output Pipe with C Testbench

  `${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_base/io_pipe_blk_c`

- Non-blocking Interface of two Static and one Dynamic Input Pipe with notification

  `${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_base/i_pipe_nb_ntf_2static_1dyn_sv`

- Non-blocking Interface of a Static Input Pipe with notification

  `${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_base/i_pipe_nb_ntf_sta_sv`

- Non-blocking Interface of two Dynamic Output Pipes with notification

  `${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_base/o_pipe_nb_ntf_2dyn_sv`

- Non-blocking Interface of Output Pipe with notification

  `${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_base/o_pipe_nb_ntf_sv`

- Non-blocking Interface of Input Pipe with notification

  `${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_base/i_pipe_nb_ntf_sv`

- Non-blocking Interface of Input and Output Pipe

  `${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_base/io_pipe_bits_nb_sv`

  `${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_base/io_pipe_bytes_nb_sv`

- Blocking Interface of Input and Output Pipe

  `${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_base/io_pipe_bits_blk_sv`

  `${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_base/io_pipe_bytes_blk_sv`

- Non-blocking Interface of Input and Output Pipe with SystemC Testbench

  `${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_base/io_pipe_nb_sysc`

- Blocking Interface of Input and Output Pipe with SystemC Testbench

  `${IXCOM_INSTALL_HOME}/share/vxe/gift/tutorial/scm/scemi_base/io_pipe_blk_sysc`

~~SCE-MI 2.1 Examples~~

- ~~C Testbench Using C Interface Examples - SCE-MI 2.1~~
- ~~C++ Testbench Using C++ Interface Examples - SCE-MI 2.1~~
- ~~SystemVerilog Testbench Using SVTB Interface Examples - SCE-MI 2.1~~

© 2021 All Rights Reserved.

- ~~SystemC Testbench Using SystemC Interface Examples - SCE-MI 2.1~~
- ~~Specman e Testbench Using Specman e Adaptor Interface Examples - SCE-MI 2.1~~
- ~~HMPI Examples: C, OSCI SystemC, and SystemVerilog Testbenches Using HMPI Thread - SCE-MI 2.1~~

~~C Testbench Using C Interface Examples - SCE-MI 2.1~~

- ~~Non-blocking Interface of Input and Output Pipe with C Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi21/input_output_pipe_nb_c~~

- ~~Blocking Interface of Input and Output Pipe with C Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi21/input_output_pipe_blk_c~~

~~C++ Testbench Using C++ Interface Examples - SCE-MI 2.1~~

- ~~Non-blocking Interface of Input and Output Pipe with C++ Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi21/input_output_pipe_nb_cpp~~

- ~~Blocking Interface of Input and Output Pipe with C++ Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi21/input_output_pipe_blk_cpp~~

~~SystemVerilog Testbench Using SVTB Interface Examples - SCE-MI 2.1~~

- ~~Non-blocking Interface of Input and Output Pipe with SystemVerilog Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi21/input_output_pipe_nb_sv~~

- ~~Blocking Interface of Input and Output Pipe with SystemVerilog Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi21/input_output_pipe_blk_sv~~

~~SystemC Testbench Using SystemC Interface Examples - SCE-MI 2.1~~

- ~~Non-blocking Interface of Input and Output Pipe with SystemC Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi21/input_output_pipe_nb_sysc~~

- ~~Blocking Interface of Input and Output Pipe with SystemC Testbench~~

© 2021 All Rights Reserved.

~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi21/input_output_pipe_blk_sysc~~

~~Specman *e* Testbench Using Specman *e* Adaptor Interface Examples - SCE-MI 2.1~~

- ~~Blocking Interface of Input and Output Pipe with Specman *e* Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi21/input_output_pipe_blk_e~~

- ~~Non-blocking Interface of Input and Output Pipe with Specman *e* Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi21/input_output_pipe_nb_e~~

~~HMPI Examples: C, OSCI SystemC, and SystemVerilog Testbenches Using HMPI Thread - SCE-MI 2.1~~

~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi21_hmpi/hm_c_pt~~
~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi21_hmpi/hm_c_sv~~
~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi21_hmpi/hm_osci_sc~~
~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi21_hmpi/hm_sv_sv_no_clk~~
~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi21_hmpi/hm_sv_vpi_sched~~

Caution

~~***The HMI/HMPI feature is being deprecated in VXE 20.05 release. It will not be supported in future releases.***~~

~~SCE-MI 2.2 Examples~~

- ~~C Testbench Using C Interface Examples - SCE-MI 2.2~~
- ~~C++ Testbench Using C++ Interface Examples - SCE-MI 2.2~~
- ~~SystemVerilog Testbench Using SVTB Interface Examples - SCE-MI 2.2~~
- ~~SystemC Testbench Using SystemC Interface Examples - SCE-MI 2.2~~
- ~~Specman e Testbench Using Specman e Adaptor Interface Examples - SCE-MI 2.2~~
- ~~HMPI Examples: C, OSCI SystemC, and SystemVerilog Testbenches Using HMPI Thread - SCE-MI 2.2~~

~~C Testbench Using C Interface Examples - SCE-MI 2.2~~

- ~~Non-blocking Interface of Input and Output Pipe with C Testbench~~

© 2021 All Rights Reserved.

~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_base/io_pipe_nb_c~~

- ~~Blocking Interface of Input and Output Pipe with C Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_base/io_pipe_blk_c~~

~~C++ Testbench Using C++ Interface Examples - SCE-MI 2.2~~

- ~~Non-blocking Interface of Input and Output Pipe with C++ Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_base/io_pipe22_nb_cpp~~

- ~~Blocking Interface of Input and Output Pipe with C++ Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_base/io_pipe22_blk_cpp~~

~~SystemVerilog Testbench Using SVTB Interface Examples - SCE-MI 2.2~~

- ~~Non-blocking Interface of two Static and one Dynamic Input Pipe with notification~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_base/i_pipe_nb_ntf_2static_1dyn_sv~~

- ~~Non-blocking Interface of a Static Input Pipe with notification~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_base/i_pipe_nb_ntf_sta_sv~~

- ~~Non-blocking Interface of two Dynamic Output Pipes with notification~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_base/o_pipe_nb_ntf_2dyn_sv~~

- ~~Non-blocking Interface of Output Pipe with notification~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_base/o_pipe_nb_ntf_sv~~

- ~~Non-blocking Interface of Input Pipe with notification~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_base/i_pipe_nb_ntf_sv~~

- ~~Non-blocking Interface of Input and Output Pipe~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_base/io_pipe_bits_nb_sv~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_base/io_pipe_bytes_nb_sv~~

- ~~Blocking Interface of Input and Output Pipe~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_base/io_pipe_bits_blk_sv~~

© 2021 All Rights Reserved.

~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_base/io_pipe_bytes_blk_sv~~

~~SystemC Testbench Using SystemC Interface Examples - SCE-MI 2.2~~

- ~~Non-blocking Interface of Input and Output Pipe with SystemC Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_base/io_pipe_nb_sysc~~

- ~~Blocking Interface of Input and Output Pipe with SystemC Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_base/io_pipe_blk_sysc~~

~~Specman e Testbench Using Specman e Adaptor Interface Examples - SCE-MI 2.2~~

- ~~Blocking Interface of Input and Output Pipe with Specman e Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_base/io_pipe_blk_e~~

- ~~Non-blocking Interface of Input and Output Pipe with Specman e Testbench~~

  ~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22/scemi22_base/io_pipe_nb_e~~

~~HMPI Examples: C, OSCI SystemC, and SystemVerilog Testbenches Using HMPI Thread - SCE-MI 2.2~~

~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22_hmpi/hm_c_pt~~
~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22_hmpi/hm_c_sv~~
~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22_hmpi/hm_osci_sc~~
~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22_hmpi/hm_sv_sv_no_clk~~
~~${XE_INSTALL_DIR}/share/vxe/gift/tutorial/uatba/scemi22_hmpi/hm_sv_vpi_sched~~

Caution

~~The HMI/HMPI feature is being deprecated in VXE 20.05 release. It will not be supported in future releases.~~

© 2021 All Rights Reserved.

30

PLI/VPI and VHPI Support

IXCOM supports both PLI/VPI standard for Verilog designs and VHPI standard for the VHDL designs. This chapter discusses the following:

- PLI/VPI Support in IXCOM
- VHPI Support in IXCOM

Important

The content of this chapter pertains only to SA mode.

PLI/VPI Support in IXCOM

IXCOM supports the Programming Language Interface 1.0 (PLI 1.0) and Programming Language Interface 2.0 (PLI 2.0 or VPI) standards for linking C or C++ code with the simulation snapshot. The PLI standards are specifically used for Verilog designs, and VHPI is the corresponding standard for VHDL designs.

Following steps describe the use-flow for linking the C/C++ code with the simulation snapshot:

1. **Step 1**: Compile the C/C++ source files of the PLI application into a shared library.

   For example, you have a `veriuser.c` file, compile this C file and the other PLI source files to create a PLI application, which is a C/C++ shared library, such as `libpli.so`. The PLI application uses the same C/C++ compiler setup as used for IXCOM.

2. **Step 2**: Use IXCOM to compile the design and generate the snapshot. The IXCOM compilation can be a single-step or a multi-step process, as shown:

   a. In single-step compile mode, specify the `+loadpli1` or `+loadvpi` option with the `ixcom` command, specifying the library name and (optionally) any bootstrap routine names, along with other options.

© 2021 All Rights Reserved.

b. In multi-step compile mode, perform the following steps. The multi-step compilation is used when the design contains some VHDL or SystemC portions.

- Specify the `vlan` command. The `vlan` command does not accept a shared library as input for PLI information.
- Generate a PLI table file containing PLI definitions. This file can be created using the `ncpli2tab` utility. Then, provide this file to `vlan` using the `-p` option to parse the design. For more information on creating a PLI table file, refer to Creating a PLI Table File on page 1376.
- In the elaboration step, run the `ixcom` command with the `+loadpli1` or `+loadvpi` option, specifying the library name and any bootstrap routine names, apart from the usual `-top` *`<top module name>`* and other options.

3. **Step 3**: Use `xeDebug` to simulate the design. During simulation, the `xeDebug` command initiates the calls to relevant VPI access and tasks and functions of the PLI application.

Note: For more information on PLI 1.0 standard, refer to the *PLI 1.0 User Guide and Reference* manual in the Xcelium documentation set.
For more information on PLI 2.0 standard, refer to the *VPI User Guide and Reference* manual in the Xcelium documentation set.

Creating a PLI Table File

In the multi-step compilation mode using IXCOM, `vlan` requires a PLI table file. The PLI table file is required to successfully parse the Verilog part of the design that can contain tasks or functions defined in the PLI application.

Following two types of schemes are supported for PLI table files:

- Using Cadence Verilog-Compatible Scheme
- Using the Synopsys VCS-Compatible Scheme

Using Cadence Verilog-Compatible Scheme

Perform the following steps to use the Cadence Verilog-Compatible PLI table file with IXCOM:

1. Compile the `veriuser.c` file and the other PLI source files to create a PLI application (a C/C++ shared library, such as `libpli.so`), using the same C/C++ compiler setup for IXCOM.
2. Use the PLI application in the following manner:

© 2021 All Rights Reserved.

a. In single-step compile mode, specify the `+loadpli1` or `+loadvpi` option with the `ixcom` command, specifying the library name and (optionally) any bootstrap routine names, along with other options.

b. In multi-step compile mode, perform the following steps:

- Specify the `vlan` command.
- Generate the special IXCOM table file that contains the PLI definitions in the Verilog-XL compatible format, using the `ncpli2tab` utility. Specify the `+loadpli1` or `+loadvpi` option, with the `libpli.so` or `libvpi.so` library and any bootstrap routine names. Specifying the bootstrap routine names is optional.
- Specify the name of the PLI table with the `vlan` command using the `-p` option.
- Specify the `ixcom` command with the `+loadpli1`(for PLI1.0 shared library) or `+loadvpi` option (for VPI shared library), `-top` *<top module name>*, and other options.
- Specify the `xeDebug` command to run the design.

Following table lists the Verilog-XL compatible syntax for a user-defined task and function entry in the PLI table file:

PLI Table Entry	Verilog-XL Compatible Syntax
user-defined task	`$myplitask`
user-defined function	`$myplifunc type=unsigned size=32`

Note: The Cadence Verilog-compatible scheme is the default PLI scheme for IXCOM.

Using the Synopsys VCS-Compatible Scheme

Perform the following steps to use the Synopsys VCS-Compatible PLI table file with IXCOM:

Note: This scheme can only be used if you have a VCS style table file containing PLI definitions.

1. Compile the `veriuser.c` file and the other PLI source files to create a PLI application (a C/CPP shared library, such as `libpli.so`), using the same C/C++ compiler setup for IXCOM.
2. Use the PLI application in the following manner:

© 2021 All Rights Reserved.

a. In single-step compile mode, you do not need to create a PLI table file. In this case, IXCOM internally creates and uses a PLI table file when you specify the `+loadpli1`(for PLI 1.0 shared library) or `+loadvpi` option (for VPI shared library) with the `ixcom` command.

b. In multi-step compile mode, perform the following steps:

- Specify the `vlan` command.
- Create a special IXCOM table VCS-compatible file that contains the PLI definitions. You can also manually create this table file.
- Specify the name of the PLI table with the `vlan` command using the `-P` option.
- Specify the `ixcom` command with the `+loadpli1`(for PLI 1.0 shared library) or `+loadvpi` option (for VPI shared library), `-top` *<top module name>*, and other options.
- Specify the `xeDebug` command to run the design.

Following table lists the VCS compatible syntax for a user-defined task and function entry in the PLI table file:

PLI Table Entry	VCS Compatible Syntax
user-defined task	`$myplitask`
user-defined function	`$myplifunc type=unsigned size=32`

Using the ncpli2tab Utility

The `ncpli2tab` utility takes a shared library, and optional bootstrap routine(s) information as input and provides information about the system tasks and functions that you have defined in a tabular format. This output can be saved in a file, which is called as the PLI table file, and is provided to IXCOM. The `ncpli2tab` utility generates the table file in the Verilog-XL format for the Verilog Compatible Scheme. IXCOM uses this generated tab file for information about user defined system tasks and functions. This utility is also internally used with IXCOM when you specify the `+loadpli1` or `+loadvpi` options and provide a PLI application, such as shared library.

ncpli2tab Options for Using PLI/VPI Features

Following options related to PLI files are supported by `ncpli2tab` utility in this release:

© 2021 All Rights Reserved.

Note: The `ncpli2tab` utility only supports Verilog-XL format. The `-p` and `-P` options are not required.

- `+loadpli1=<arg>`: Used to specify a shared library containing the PLI 1.0 application.
- `+loadvpi=<arg>`: Used to specify a shared library containing the VPI application.
- `-o <arg>`: Used to specify the output file. If nothing is specified, `ncpli2tab` writes output to `stdout`.

Following is the syntax of using the `ncpli2tab` for PLI 1.0:

```
ncpli2tab +loadpli1="./libpli"
ncpli2tab +loadpli1="./libpli:bootstrap_routine(s)"
```

Following is the syntax of using the `ncpli2tab` for VPI:

```
ncpli2tab +loadvpi="./libvpi"
ncpli2tab +loadvpi="./libvpi:bootstrap_routine(s)"
```

PLI/VPI Options to IXCOM

IXCOM supports the following PLI/VPI options:

- `-p <arg>`: Provides information about PLI tasks in Verilog-XL format.
- `-P <arg>`: Provides information about PLI tasks in VCS format
- `+loadpli1=<arg>`: Provides a shared library containing the PLI 1.0 application. However, IXCOM internally generates a tab file in the Verilog-XL format for its usage.
- `+loadvpi=<arg>`: Provides a shared library containing the VPI application. However, IXCOM internally generates a tab file in the Verilog-XL format for its usage.

Combinations of Bootstrap Routines Supported by IXCOM

You can specify the bootstrap routines with the IXCOM command as values of the `+loadpli1` and `+loadvpi` options. These options are supported by both `ixcom` and `xrun` commands.

Following combinations of bootstrap routines are supported by IXCOM:

- `ixcom +loadpli1="lib1:boot_routine1.boot_routine2"`

  In the above command, the elaborator loads the library, *lib1*, and executes the bootstrap routine, *boot_routine1*. The simulator loads the library, *lib1*, and executes the bootstrap routine, *boot_routine2*. You use this command format when you want to run a PLI application only during simulation. Such a PLI application can

© 2021 All Rights Reserved.

perform tasks, such as recording the duration of a simulation or establishing communication with another simulation session.

Note: The bootstrap routine, *boot_routine2* cannot register new system tasks in addition to the ones registered by the bootstrap routine, *boot_routine1*.

- `ixcom +loadpli1="lib1:boot_routine1,boot_routine2.boot_routine2"`

  In the above command, the elaborator loads the library, *lib1*, and executes the bootstrap routines, *boot_routine1* and *boot_routine2*. The simulator loads the library, *lib1*, and executes the bootstrap routine, *boot_rooutine2*.

- `ixcom +loadpli1="lib1:boot_routine1"`

  In the above command, the elaborator loads the library, *lib1*, and executes the bootstrap routine, *boot_routine1*. The simulator loads the library, *lib1*, and executes the bootstrap routine, *boot_routine1*.

- `ixcom +loadpli1="lib1:boot_routine1."`

  In the above command, the elaborator loads the library, *lib1*, and executes the bootstrap routine, *boot_routine1*. The simulator just loads the library, *lib1*, and performs no other action.

- `ixcom +loadpli1="lib1:.boot_routine1"`

  In the above command, elaborator loads the library, *lib1*. The simulator loads the library, *lib1*, and executes the bootstrap routine, *boot_routine1*.

  Note: The above command format is typically used when you need to load and run PLI applications that do not register any system tasks.

- `ixcom +loadpli1="lib1:"`

  In the above command, the elaborator loads the library, *lib1*. The simulator loads the library, *lib1*. Both use the `veriusertfs` array or `vlog_startup_routines`, if available in absence of any bootstrap routines or functions.

- `ixcom +loadvpi="vpilib:boot_routine1.boot_routine2"`

  In the above command, the elaborator loads the library, *vpilib*, and executes the routine, *boot_routine1*. The simulator loads the library, *vpilib*, and executes the routine, *boot_routine2*.

- `ixcom +loadvpi="vpilib:boot_routine1,boot_routine2.boot_routine3"`

© 2021 All Rights Reserved.

In the above command, the elaborator loads the library, *vpilib*, and executes the routines, *boot_routine1* and *boot_routine2*. The simulator loads the library, *vpilib*, and executes the routine, *boot_routine3*.

- `ixcom +loadvpi="vpilib:boot_routine1"`

  In the above command, the elaborator loads the library, *vpilib*, and executes the routine, *boot_routine1*. The simulator loads the library, *vpilib*, and executes the routine, *boot_routine1*.

- `ixcom +loadvpi="vpilib:boot_routine1."`

  In the above command, the elaborator loads the library, *vpilib*, and executes the routine, *boot_routine1*. The period [**.**] at the end of the command line indicates that the simulator does not perform any action.

- `ixcom +loadvpi="vpilib:.boot_routine1"`

  In the above command, the elaborator loads the library, *vpilib*. The simulator loads the library, *vpilib*, and executes the routine, *boot_routine1*. You typically use this command format to load and run VPI applications that do not register any system tasks.

- `ixcom +loadvpi="vpilib:"`

  In the above command, both the elaborator and the simulator load the library, *vpilib*. No functions or routines are executed.

Restrictions on Using PLI/VPI Applications

While working with PLI applications, you need to take care of certain restrictions related to:

- Using a PLI/VPI Application in a Mixed-HDL Design
- Using a Verilog PLI Application in a Testbench
- Using a PLI/VPI Application in a DUT Module
- Statically Linking PLI/VPI Applications
- Naming PLI/VPI Applications

Using a PLI/VPI Application in a Mixed-HDL Design

If the design contains Verilog and VHDL portions, PLI/VPI tasks can only be called from the Verilog portion of the design. In addition, the PLI/VPI tasks can only be used to traverse and `set`, `force`, or `read` Verilog signals and variables.

© 2021 All Rights Reserved.

PLI and VPI routines cannot be used to traverse the hierarchy of a VHDL design, and to `set`, `force`, and `read` VHDL signals and variables. You should make use of VHPI routines to perform these actions on the VHDL part of the design.

Using a Verilog PLI Application in a Testbench

When using a Verilog PLI application, consider the following:

- Using Task/Function Routines Instead of Access Routines
- Referencing Nets and Registers Across the Hierarchy from a PLI Routine
- Referencing Nets and Registers in DUT Modules from the Testbench

Using Task/Function Routines Instead of Access Routines

Following guidelines are recommended when writing Verilog PLI applications. This is because, Verilog PLI applications are run on the software side, and should be written efficiently.

- Use task or function routines instead of access routines in the PLI code. The task and function routines simulate more efficiently on Xcelium and other compiled-code simulators.
- Use the `+tfReadOnly` option with the `ixcom` command, when the arguments of the PLI/VPI tasks and functions are read-only and not read-write.

Referencing Nets and Registers Across the Hierarchy from a PLI Routine

IXCOM compilation generates a configuration file, `hdl.var` in the `xc_work` directory. The `hdl.var` file contains the `-access +rw` argument, which gives read and write access to all signals in the entire design. This allows PLI/VPI application to access/update signals throughout the design hierarchy.

Cadence recommends that you do not change the values of `-access` argument as it might make the signals inaccessible to PLI/VPI application. If a PLI/VPI routine attempts to read/write to signals that were not accessible, an appropriate run-time error is generated, as shown below:

```
Error from IUS:
          ERROR:          ACC       VISNOR
          Attempting to read from <full name of signal> which does not have read
access.
          ERROR:          ACC       VISNOW
          Attempting to place a value into <full name of signal> which does not
have write access.
```


© 2021 All Rights Reserved.

Referencing Nets and Registers in DUT Modules from the Testbench

You can cross-reference nets and registers in DUT modules from the testbench for the purpose of using them in PLI calls, if these nets and registers are exported during compilation. You can export the nets and registers in following two ways:

- Signals are automatically exported if, you reference a DUT net or register in the testbench or other module running in software simulation mode.

  For example, the following is a read reference to a DUT signal:

  ```
  always @(posedge testbench.chip.sys_clk)
  ```

- Signals are explicitly exported using either the `$export_read` or `$export_frcrel` Verilog system tasks from a testbench module.

Using a PLI/VPI Application in a DUT Module

A DUT module containing PLI/VPI calls is run on the software side. This is because the emulator cannot directly call a PLI/VPI routine.

In case the DUT module containing the PLI/VPI calls is to be run on the emulator, all the PLI/VPI calls of the DUT module should be ignored. To ignore all the PLI/VPI calls from a DUT module, specify the `+rtlIgnTask+<pli_call_name>` argument with the `ixcom` command. The *`pli_call_name`* argument indicates the names of all the PLI/VPI tasks and functions called from the DUT module. This allows the DUT module to be run on the emulator.

Statically Linking PLI/VPI Applications

IXCOM does not support statically linked PLI/VPI applications. It is supported by Xcelium for backward compatibility.

Naming PLI/VPI Applications

A boot-strap function is not necessary if the `veriusertfs` (for PLI) or `vlog_startup_routines` (for VPI) array of function pointers is initialized within the PLI or VPI application. However, if the shared library is not named as `libpli.so` (for PLI) or `libvpi.so` (for VPI), Xcelium requires a boot-strap function.

© 2021 All Rights Reserved.

Example: Using PLI and VPI Applications

Following example demonstrates how to use a PLI or VPI function. This example uses following three source files:

- *veriuser.c*: Contains the registration of the task that you have defined using a `veriusertfs` array. Therefore, PLI bootstrap routine is not required in this case.
- *myplitest.c*: Contains the implementation of the PLI task's call function.
- *design_file.v*: Contains HDL design that calls the PLI/VPI task.

veriuser.c

```
.........
.........
.........
.........
s_tfcell veriusertfs[] =
{
    /*** Template for an entry:
    { usertask|userfunction, data,
      checktf(), sizetf(), calltf(), misctf(),
      "$tfname", forwref?, Vtool?, ErrMsg? },
    Example:
    { usertask, 0, my_check, 0, my_func, my_misctf, "$my_task" },
    ***/
    /*** add user entries here ***/
    { userfunction, 0, 0, size1, mypli1, mypli1_misc, "$mypli1", 1 },
    { userfunction, 0, 0, size32, mypli2, mypli2_misc, "$mypli2", 1 },
    { userrealfunction, 0, 0, size64, mypli3, mypli3_misc, "$mypli3", 1 },
    { usertask, 0, 0, 0, mypli4, mypli4_misc, "$mypli4", 1 },
    {0} /*** final entry must be 0 ***/
};
.........
.........
.........
.........
```

myplitest.c

```
.........
.........
.........
.........
int mypli1()
{
   tf_putp(0, 1);
}
int mypli2()
{
   tf_putlongp(0, 12, 0);
}
int mypli3()
{
   tf_putrealp(0, 123.456);
}
int mypli4()
```


© 2021 All Rights Reserved.

```
{
   io_printf("In mypli4 at time %d\n", tf_gettime());
}
.........
.........
.........
.........
```

design_file.v

```
module myTopMod;
    integer mypliretval;
    initial begin
      mypliretval = $mypl2;
      $mypli4;
    end
endmodule
```

Perform the following steps to use the PLI/VPI application:

1. **Step 1**: Compile the C/C++ source files of the **PLI application** into a shared library.

   Specify the following commands to compile the C or C++ source files of the PLI application into a shared library.

   ```
   cc -m32 -fpic -c -g -I$CDS_INST_DIR/tools/include -DVXL myplitest.c \
      -o myplitest.o
   cc -m32 -fpic -c -g -I$CDS_INST_DIR/tools/include -DVXL veriuser.c \
      -o veriuser.o
   cc -m32 -shared -o libpli.so myplitest.o veriuser.o
   ```

   Similarly, C/C++ source files for a **VPI application** can be:

   - *`vpi_user.c`*: Contain the registration of the task that you have defined using a `vlog_startup_routines` function. Therefore, VPI bootstrap routine is not required in this case.
   - *`myvpitest.c`*: Contains the implementation of the task's call function.
   - *`design_file.v`*: Contains HDL design that calls the PLI/VPI task.

   Specify the following commands to compile the C or C++ source files of the VPI application into a shared library:

   ```
   cc -m32 -fpic -c -g -I$CDS_INST_DIR/tools/include -DVXL myvpitest.c \
      -o myvpitest.o
   cc -m32 -fpic -c -g -I$CDS_INST_DIR/tools/include -DVXL vpi_user.c \
      -o vpi_user.o
   cc -m32 -shared -o libvpi.so myvpitest.o vpi_user.o
   ```

   Note: The above `CC` compilation commands are to compile a shared library on 32-bit linux platform and are only indicative. These commands vary according to the compiler and the operating system used.

© 2021 All Rights Reserved.

2. **Step 2**: Run the `ixcom` command to compile the design and generates the snapshot.Then, run the `xrun` command to simulate the design and initiate the calling function of the PLI or the VPI task.

   a. Single-step Compilation for a **PLI** application: In this mode, specify the following commands to compile and run the design:

   ```
   ixcom +loadpli1="./libpli" design_file.v [other_options]
   xeDebug <input_tclfile>
   ```

   b. Multi-step Compilation for a **PLI** application: In this mode, specify the following commands to compile and run the design. Multi-step compilation is used when the design contains some VHDL or SystemC portions.

   - Run the `ncpli2tab` command with the PLI application to generate a Verilog-XL style table file, *`mytfs.pli`*, as shown:

     ```
     ncpli2tab +loadpli1="./libpli" -o mytfs.pli
     ```

   - Run the `vlan` command with the Verilog-XL style table file, `mytfs.pli`, generated in the previous step, as shown. This step parses the HDL design.

     ```
     vlan -p mytfs.pli design_file.v [other_options]
     ```

   - Run the `vhan` command, as shown:

     ```
     vhan design_file.vhd [other_options]
     ```

   - Run the `ixcom` command to compile the design and generate the snapshot, as shown:

     ```
     ixcom +loadpli1="./libpli" -top myTopMod [other_options]
     ```

   - Run the `xeDebug` command to simulate the design and initiate the calling function of the PLI task, as shown:

     ```
     xeDebug <input_tclfile>
     ```

   c. Single-step compilation for a **VPI** application: In this mode, specify the following commands to compile and run the design:

   ```
   ixcom +loadvpi="./libvpi" design_file.v [other_options]
   ```

   d. Multi-step compilation for a **VPI** application: In this mode, specify the following commands to compile and run the design. Multi-step compilation is used when the design contains some VHDL or SystemC portions.

   - Run the `ncpli2tab` command with the VPI application to generate a Verilog-XL style table file, as shown:

     ```
     ncpli2tab +loadvpi="./libvpi" -o mytfs.pli
     ```

   - Run the `vlan` command with the Verilog-XL style table file generated in the previous step, as shown. This step parses the HDL design.

     ```
     vlan -p mytfs.pli design_file.v [other_options]
     ```


© 2021 All Rights Reserved.

- Run the `vhan` command, as shown:

  ```
  vhan design_file.vhd [other_options]
  ```

- Run the `ixcom` command to compile the design and generate the snapshot, as shown:

  ```
  ixcom +loadvpi="./libvpi" -top myTopMod [other_options]
  ```

- Run the `xeDebug` command to simulate the design and initiate the calling function of the VPI task, as shown:

  ```
  xeDebug <input_tclfile>
  ```

VHPI Support in IXCOM

VHDL Programming Language Interface or VHPI is a standard interface that enables you to link C/C++ programs with the VHDL design.

A C/C++ application using VHPI interface can be integrated with a VHDL design in the following cases:

- To access various design elements, such as design hierarchy, connectivity information, and so on. This type of access is called static access.
- To monitor various states of a design, such as current simulation time, signal value, and so on. This type of access is called dynamic access.
- To call a C/C++ VHPI function/procedure from a VHDL design, and is termed as calling a foreign subprogram.
- To create the body of the VHDL architecture using C/C++ VHPI routines, and is termed as creating a foreign architecture.

For more information on VHPI use model and its features, refer to the *VHPI User Guide* in the Xcelium documentation set.

Restrictions on Using VHPI Applications

Consider the following limitations, while working with the VHPI applications:

- A foreign architecture or foreign subprogram must not be part of the DUT.
- Any VHPI application should not access the signals/variables falling in the DUT. This implies that VHPI application should not try to get the handle of any object present in the DUT.

© 2021 All Rights Reserved.

Example: Using VHPI Applications

The following example illustrates the compilation and usage of VHPI applications with IXCOM. This example includes the following source files:

`test.c` - Contains the implementation of VHDL function.

`test.vhd` - VHDL file that calls the VHPI function/procedure

Perform the following steps to use VHPI application:

1. Compile the C/C++ source files of the VHPI application into a shared library.

   ```
   cc -fpic -c -g -I$CDS_INST_DIR/tools/include test.c
   cc -shared -o libvhpi.so test.o
   ```

2. Analyze the VHDL part of the design.

   ```
   vhan test.vhd
   ```

3. Analyze the Verilog part of the design. This step is not applicable for this example.

4. Run the `ixcom` command to generate a snapshot.

   ```
   ixcom -top <top_module> [other_options]
   ```

5. Use the `xrun` or `xeDebug` command for simulation.

   ```
   xrun -R [other_options]
   ```

   Or,

   ```
   xeDebug run.xel
   ```

   Note: Most of this VHPI application has references in the VHDL part of design. So, it is linked at elaboration time. But, if you want to execute any flags or functions at the start of simulation time, pass these during simulation.

6. If you want to use ***xrun*** or xeDebug for simulation, create the `uxc_hdl.var` file in the test directory in which you want to run simulation and add the following line in it:

   ```
   DEFINE XMSIMOPTS $XMSIMOPTS -loadvhpi ./libvhpi.so:start_function
   ```

   Note: This is a sample to call `start_function` from `libvhpi.so` at simulation start time. You can set the library and function names as applicable.

VHPI Example Files

Code for the files referred in the above example is given below:

© 2021 All Rights Reserved.

run.sh - command script

```
cc -fpic -c -g -I$CDS_INST_DIR/tools/include test.c
cc -shared -o libvhpi.so test.o
vhan test.vhd
ixcom -top e +dut+dut -ua
xeDebug run.qel
```

run.xel - XEL command script run by xeDebug

```
debug .
host .
run 100
xc xt0 zt0 run
run 2000
run 2000
run 2000
exit
```

test.c - VHPI application file

```
#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include "vhpi_user.h"
void start_func()
{
  vhpi_printf (" Hello there in the VHPI Start function \n");
}
void My_proc(const struct vhpiCbDataS *data)
{
 vhpiValueT val, val_arr[2];
 vhpiHandleT ParamIt, refHdl, subPHdl;
 int i = 0;
 vhpi_printf(" \n Handling Variables \n\n");
 subPHdl = data->obj;
 ParamIt = vhpi_iterator(vhpiParamDecls, subPHdl);
 while (refHdl = vhpi_scan(ParamIt)) {
   val.format = vhpiObjTypeVal;
   vhpi_get_value(refHdl, &val);
   switch (val.format) {
      case vhpiIntVal :
       vhpi_printf(" Integer :  %d \n", val.value.intg);
       break;
      case vhpiRealVal :
       vhpi_printf(" Real :  %f \n", val.value.real);
       break;
      default :
       vhpi_printf("Unknwon Type \n");
   }
 }
 /* Putting the object value */
 val_arr[0].format = vhpiIntVal;
 val_arr[1].format = vhpiRealVal;
 val_arr[0].value.intg = 23;
 val_arr[1].value.real = 32.34;
 ParamIt = vhpi_iterator(vhpiParamDecls, subPHdl);
 while (refHdl = vhpi_scan(ParamIt))
```


© 2021 All Rights Reserved.

```
   vhpi_put_value(refHdl, &val_arr[i++], vhpiDeposit);
}
```

test.vhd - VHDL design file

```
library ieee, std;
use ieee.std_logic_1164.all;
use std.standard.all;
package pack is
end pack;

library ieee;
use ieee.std_logic_1164.all;
entity DUT is
  port (in1 : in std_logic; out1 : out std_logic);
end;

architecture A of DUT is
begin
 out1 <= in1;
end a;

-------------------------------------------------------------------------------

library std,ieee;
use std.standard.all;
use ieee.std_logic_1164.all;
entity e is
  shared variable s_j : real := 5.7;
begin
end e;

architecture a of e is
  signal s_i : integer := 3;
  signal s_k, out_k : std_logic;
 procedure My_proc(signal i : inout integer; variable j : inout real);
 attribute FOREIGN of My_proc : procedure is "VHPIDIRECT libvhpi : ";
begin
 inst1 : entity work.DUT port map(s_k, out_k);
 var : process
  variable s_l : boolean;
  variable s_n : time := 2 ns;
 begin
  My_proc(s_i, s_j);
  wait;
 end process;

end a;
```


© 2021 All Rights Reserved.

31

Compiling Designs with SystemVerilog DPI

IXCOM supports a subset of the IEEE-P1800 version of the *SystemVerilog Direct Programming Interface (DPI)*.

This **chapter** contains the following topics:

- Overview of SystemVerilog DPI
- SystemVerilog Layer
- Foreign Language (C/C++) Layer
  - Supported C/C++ Functions
  - Unsupported C/C++ Functions
- import Functions and Tasks
- export Functions and Tasks
- Using SystemVerilog DPI
- Generating C Header Files for Exported DPI
- Support for Open Arrays in DPI

Note: Please review standard SystemVerilog DPI documentation for a more complete description of the SystemVerilog DPI. This chapter will focus primarily on how SystemVerilog DPI has been implemented for the environment.

Important

The content of this chapter pertains only to SA mode.

© 2021 All Rights Reserved.

Overview of SystemVerilog DPI

The SystemVerilog Direct Programming Interface (DPI) is an interface between SystemVerilog and any foreign language, although C/C++ is the only one supported at this time. SystemVerilog DPI greatly simplifies connecting a Verilog-based design or test environment to C/C++.

Earlier interfaces, such as the Programming Language Interface (PLI), required you to set up an explicit table to link Verilog with C/C++. This table was usually created as a `veriusertf` table in user C/C++ code or using a `TAB` file. You also had to access Verilog design objects from within C/C++ using access functions provided by the PLI, such as `tf_getp()`, `acc_set_value()`, or `tf_putp()`, and so on.

SystemVerilog DPI allows the HDL-ICE Compiler to create the interface and data access, using two *layers*:

- SystemVerilog Layer which can call `import` functions/tasks that have been defined in the Foreign language (C/C++) layer.
- **Foreign Language (C/C++) Layer** which can call `export` functions/tasks that have been defined in the SystemVerilog layer.

SystemVerilog Layer

The SystemVerilog layer of the DPI is the layer from which you `import` C/C++ tasks or functions. It is also the layer in which you define `export` tasks or functions that can be called by the C/C++ layer.

All import DPI tasks and functions have to be executed on the simulator. All export DPI tasks and functions, by default, are also executed on the simulator unless the `tb_export` compiler directive is used to map the export DPI task to the emulator.

Foreign Language (C/C++) Layer

The foreign language, or C/C++, layer of the DPI is the layer in which you call an exported Verilog task or function from C/C++ code. The task or function must have an `extern` declaration on the C/C++ side. (Refer to Supported/Unsupported C/C++ Functions on page 1393.)

This layer specifies how:

- Actual arguments are passed

© 2021 All Rights Reserved.

- Arguments are accessed from the foreign (C/C++) code
- Verilog data types are represented
- Data types are translated to and from some predefined C/C++-like types

IXCOM allows SystemVerilog DPI only in the context of the Verilog-2001 subset. The following table maps Verilog data types to C/C++ data types:

Table 31-1 Verilog-2001 Data Types Mapped to C/C++ Data Types

Verilog-2001 Data Type	C/C++ Data Type
four-value scalar	unsigned char
bit	unsigned char
four-value vector (packed array, time, integer)	svLogicVec32, svLogicPackedArrRef
bit vector (packed array)	svBitVec32, svBitPackedArrRef
real	double

Return types of the DPI `import` functions are limited to the following Verilog types:

- bit (two-value, vector up to 32 bits)
- logic (scalar four-valued logic)
- real

Supported/Unsupported C/C++ Functions

This section contains lists of:

- Supported C/C++ Functions
- Unsupported C/C++ Functions

© 2021 All Rights Reserved.

Supported C/C++ Functions

IXCOM supports the following functions, whose `extern` declarations are found in the `svdpi.h` file (the standard file supplied with SystemVerilog DPI):

```
extern int svSizeOfBitPackedArr(int width);
extern int svSizeOfLogicPackedArr(int width);
extern void svPutBitVec32(svBitPackedArrRef d, const svBitVec32* s, int w);
extern void svPutLogicVec32(svLogicPackedArrRef d, const svLogicVec32* s, int w);
extern void svGetBitVec32(svBitVec32* d, const svBitPackedArrRef s, int w);
extern void svGetLogicVec32(svLogicVec32* d, const svLogicPackedArrRef s, int w);
extern svBit svGetSelectBit(const svBitPackedArrRef s, int i);
extern svLogic svGetSelectLogic(const svLogicPackedArrRef s, int i);
extern void svPutSelectBit(svBitPackedArrRef d, int i, svBit s);
extern void svPutSelectLogic(svLogicPackedArrRef d, int i, svLogic s);
extern void svGetPartSelectBit(svBitVec32* d, const svBitPackedArrRef s, int i, int
w);
extern svBitVec32 svGetBits(const svBitPackedArrRef s, int i, int w);
extern svBitVec32 svGet32Bits(const svBitPackedArrRef s, int i);
/* 32-bits */
extern unsigned long long svGet64Bits(const svBitPackedArrRef s, int i);
extern void svGetPartSelectLogic(svLogicVec32* d,
const svLogicPackedArrRef s, int i, int w);
extern void svPutPartSelectBit(svBitPackedArrRef d, const svBitVec32 s,
int i, int w);
extern void svPutPartSelectLogic(svLogicPackedArrRef d,
const svLogicVec32 s, int i, int w);
extern svScope svGetScope();
extern const char* svGetNameFromScope(const svScope);
extern svScope svGetScopeFromName(const char* scopeName);
extern int svPutUserData(const svScope scope, void *userKey,
void* userData);
extern void* svGetUserData(const svScope scope, void* userKey);
extern int svGetCallerInfo(char **fileName, int *lineNumber);
extern svScope svSetScope(const svScope scope);
```

Unsupported C/C++ Functions

IXCOM does not currently support the following functions, whose `extern` declarations are *not* included in the `svdpi.h` file:

```
extern int svLeft(const svOpenArrayHandle h, int d);
extern int svRight(const svOpenArrayHandle h, int d);
extern int svLow(const svOpenArrayHandle h, int d);
extern int svHigh(const svOpenArrayHandle h, int d);
extern int svIncrement(const svOpenArrayHandle h, int d);
extern int svLength(const svOpenArrayHandle h, int d);
extern int svDimensions(const svOpenArrayHandle h);
extern void *svGetArrayPtr(const svOpenArrayHandle);
extern int svSizeOfArray(const svOpenArrayHandle);
extern void *svGetArrElemPtr(const svOpenArrayHandle, int indx1, ...);
extern void *svGetArrElemPtr1(const svOpenArrayHandle, int indx1);
extern void *svGetArrElemPtr2(const svOpenArrayHandle, int indx1,
int indx2);
extern void *svGetArrElemPtr3(const svOpenArrayHandle, int indx1,
int indx2, int indx3);
extern void svPutBitArrElemVec32(const svOpenArrayHandle d,
```


© 2021 All Rights Reserved.

```
const svBitVec32* s, int indx1, ...);
extern void svPutBitArrElem1Vec32(const svOpenArrayHandle d,
const svBitVec32* s, int indx1);
extern void svPutBitArrElem2Vec32(const svOpenArrayHandle d,
const svBitVec32* s, int  indx1,int indx2);
extern void svPutBitArrElem3Vec32(const svOpenArrayHandle d,
const svBitVec32* s, int indx1, int indx2, int indx3);
extern void svPutLogicArrElemVec32(const svOpenArrayHandle d,
const svLogicVec32* s, int indx1, ...);
extern void svPutLogicArrElem1Vec32(const svOpenArrayHandle d,
const svLogicVec32* s, int indx1);
extern void svPutLogicArrElem2Vec32(const svOpenArrayHandle d,
const svLogicVec32* s,int indx1, int indx2);
extern void svPutLogicArrElem3Vec32(const svOpenArrayHandle d,
const svLogicVec32* s, int indx1, int indx2, int indx3);
xtern void svGetBitArrElemVec32(svBitVec32* d,
const svOpenArrayHandle s, int indx1, ...);
extern void svGetBitArrElem1Vec32(svBitVec32* d,
const svOpenArrayHandle s, int indx1);
extern void svGetBitArrElem2Vec32(svBitVec32* d,
const svOpenArrayHandle s, int indx1, int indx2);
extern void svGetBitArrElem3Vec32(svBitVec32* d,
const svOpenArrayHandle s, int indx1, int indx2, int indx3);
extern void svGetLogicArrElemVec32(svLogicVec32* d,
const svOpenArrayHandle s, int indx1, ...);
extern void svGetLogicArrElem1Vec32(svLogicVec32* d,
const svOpenArrayHandle s, int indx1);
extern void svGetLogicArrElem2Vec32(svLogicVec32* d,
const svOpenArrayHandle s, int indx1, int indx2);
extern void svGetLogicArrElem3Vec32(svLogicVec32* d,
const svOpenArrayHandle s, int indx1, int indx2, int indx3);
extern svBit svGetBitArrElem(const svOpenArrayHandle s, int indx1, ...);
extern svBit svGetBitArrElem1(const svOpenArrayHandle s, int indx1);
extern svBit svGetBitArrElem2(const svOpenArrayHandle s, int indx1,
int indx2);
extern svBit svGetBitArrElem3(const svOpenArrayHandle s, int indx1,
int indx2, int indx3);
extern svLogic svGetLogicArrElem(const svOpenArrayHandle s,
int indx1, ...);
extern svLogic svGetLogicArrElem1(const svOpenArrayHandle s, int indx1);
extern svLogic svGetLogicArrElem2(const svOpenArrayHandle s, int indx1,
int indx2);
extern svLogic svGetLogicArrElem3(const svOpenArrayHandle s, int indx1,
int indx2, int indx3);
extern void svPutLogicArrElem(const svOpenArrayHandle d, svLogic value,
int indx1, ...);
extern void svPutLogicArrElem1(const svOpenArrayHandle d, svLogic value, int
indx1);
extern void svPutLogicArrElem2(const svOpenArrayHandle d, svLogic value, int
indx1,int indx2);
extern void svPutLogicArrElem3(const svOpenArrayHandle d, svLogic value, int indx1,
int  indx2, int indx3);
extern void svPutBitArrElem(const svOpenArrayHandle d, svBit value,
int indx1, ...);
extern void svPutBitArrElem1(const svOpenArrayHandle d, svBit value,
int indx1);
extern void svPutBitArrElem2(const svOpenArrayHandle d, svBit value,
int indx1, intindx2);
extern void svPutBitArrElem3(const svOpenArrayHandle d, svBit value,
int indx1, int indx2, int indx3);
```


© 2021 All Rights Reserved.

```
extern int svIsDisabledState();
extern void svAckDisabledState();
```

import Functions and Tasks

`import` functions or tasks are defined in the C/C++ layer and are called in the SystemVerilog layer.

To use `import` functions or tasks, you must declare them in your Verilog code using an *import declaration*, for example:

```
import "DPI-C" context task          c_rand (output int x, output int y);
import "DPI-C" context function int pytha_check \
        (input int inx, input int iny, input int mod, input int angle);
```

This section has the following subsections:

- Definition of Import Declaration
- Example: SystemVerilog Layer Calling C/C++ Function
- Accessing Two-Value Bit Values in C/C++
- Accessing Four-Value Logic Values in C/C++

Definition of Import Declaration

- Task or function: A DPI call can be of the type task or function. The main difference between a task and function is:
  - A DPI function is equivalent to a normal C or Verilog function – it has a return value and appears on the RHS of the expressions. Conversely, a DPI task does not return any value to the RHS – it returns a value from C to HDL using an argument.
  - A function can have arguments only of type `input`; a task can have arguments of type `input`, as well as `output` and `inout`.
- Function return type: A function's return type can be one of the following:
  - `bit` [signed|unsigned] ['[x:y]']: two-state, optional signed or unsigned (default) specification, optional range specification for a vector (limited up to only 32 bits)
  - `int` [signed|unsigned]: two-state, 32 bits, optional signed (default) or unsigned specification
  - `logic`: default if unspecified, four-state scalar value
  - `real`: 64 bit floating point value

© 2021 All Rights Reserved.

- Task or function arguments: Task or function arguments have a mandatory direction specification and an optional type.
  - Direction: Direction can be of type `input`, `output` or `inout`. The direction is defined from the C/C++ interface perspective.

    `input` type arguments are set from the HDL and read by C/C++.

    `output` type arguments are set from C/C++ and read by the HDL.

    `inout` type arguments can be read or set from either the HDL or C/C++.

    The mechanism for passing values for `output` or `inout` type arguments to/from C/C++ is in the form of a reference on the C/C++ side.
  - Type: The following types are supported for arguments:

    [`logic`] ['[x:y]']: This is the default type if unspecified, 4-state unsigned value, an optional range specification for a vector.

    `bit` [signed|unsigned] ['[x,y]']: two-state value, an optional signed or unsigned specification (default unsigned); an optional range specification for a vector.

    `int` [signed|unsigned]: two-state, 32-bit value, an optional signed or unsigned (default signed).

    `real`: 64 bit floating point value.
- Following are some more examples of import declarations:

```
import "DPI" context function myDpi(input [0:5]  first); // function returning
scalar 4 value, one input 6 bits wide logic type argument
```

```
import "DPI" context function int unsigned myDpi(input bit signed [0:7]  first,
input second); // function returning an unsigned int, one input 8 bits wide
signed bit type argument, second input 4-value logic scalar
```

```
import "DPI" context task myDpi(input bit [0:7]  first, output int second); /
/ task, one input 8 bits wide unsigned bit type argument, one output signed int
type argument
```

- SystemVerilog has two types of imported functions: `pure` and `context`. Tasks can be only `context`. To be `pure`, the result of a function must depend only on the values of its inputs, and it must not change the value of any variable directly or indirectly (by calling other functions), that is, it must not have any *side effects*.

  In contrast, a `context` task or function accesses SystemVerilog data (other than arguments passed to the function) by the PLI, VPI, and so on. You must identify `context` functions in the declaration to optimize the code.

© 2021 All Rights Reserved.

Example: SystemVerilog Layer Calling C/C++ Function

In the following DPI-based implementation of a Pythagorean processor testbench, the DPI function `c_rand` provides two random data values to test code.

The test process, then, applies the inputs to the DUT and sets up a control signal `do_pytha` to trigger a computation. The test process waits for the status signal `done_pytha` to be set, which indicates that results are available from the DUT. Then, the DPI function call `pytha_check` checks the results against those computed using C/C++ and reports errors exceeding the limit set in the verification environment.

In addition to the DPI functions, behavioral Verilog code, including `wait` conditions and dynamic `while` loops, is used as a part of the test process.

As previously mentioned, you must declare any `import` functions in your Verilog code using an *import declaration*:

```
import "DPI-C" context task        c_rand (output int x, output int y);
import "DPI-C" context function int pytha_check \
        (input int inx, input int iny, input int mod, input int angle);
```

After these `import` functions have been declared, they can be used as regular functions or tasks in Verilog:

```
always @(posedge clk)
  begin : test
    while(!reset_)
      @(posedge clk);
    @(negedge clk);
    if(count < 10)
      begin
        c_rand(inx,iny);
        do_pytha <= 1;
        @(posedge clk)
          do_pytha <= 0;
        while(done_pytha)
          @(posedge clk);
        count = count+1;
        while(!done_pytha)
          @(posedge clk);
        i=pytha_check(inx, iny, mod, angle);
        errors = errors+i;
      end
    else
      begin // ixc tbcall_region
        $display("Done with %0d computations, %0d errors", count, errors);
        $finish;
      end
end
```

For this example, the DPI functions are defined as:

```
#include "veriuser.h"
#include <stdio.h>
```


© 2021 All Rights Reserved.

```
#include <stdlib.h>
#include <math.h>

// Function corresponding to SystemVerilog DPI declaration
// import "DPI-C" context task         c_rand (output int x, output int y);
void c_rand(int* x, int* y)
{
  // "output" type values are passed by reference
  *x=(1+rand()%(1<<15));
  *y=(1+rand()%(1<<15));
}

// Function corresponding to SystemVerilog DPI declaration
// import "DPI-C" context function int pytha_check (input int inx, input int iny,
input int mod, input int angle);
int pytha_check(int x, int y, int mod, int ang) {
  int inx, iny, imod, iang;
  if ((x>>15) & 1) inx = -(x&0x7fff); else inx = x; // sign extension from 16 bits
to int
  if ((y>>15) & 1) iny = -(y&0x7fff); else iny = y; // sign extension as above
  imod = (int) sqrt((double) (inx*inx+iny*iny)); // C computed modulus
  iang = 180 - (int) (180*atan2(iny, inx))/(M_PI); // C computed angle
  if(iang < 0) iang -= 360;
  if((abs(imod-mod)>2) | (abs(iang-ang)>15))
    {
      // DUT generated result deviates too much from C computed
      io_printf("PYTHA-DPI Inx: %d Iny: %d Mod: %d Ang: %d C-Mod: %d C-Ang:
      %d\n", inx, iny, mod, ang, imod, iang);
      return(1);
    }
  else
    return(0);
}
```

To compile this code, use the following `ixcom` command:

```
ixcom tpytha.v +dut+test_pytha +sv +bc+test_pytha ...
```

The above command includes the following options:

- `+sv`

  Enables the SystemVerilog DPI in IXCOM.

  Note: Refer to the *+sv* section in the *Commands and System Tasks for SA Mode* chapter of the *VXE Command Reference Manual* for more detail.

- `+bc+<module>+<module>...`

  Enables behavioral compilation of the specified module(s). This option is required when the module contains either SystemVerilog DPI, behavioral Verilog, or both (as in this case).

  Note: Refer to the `+bc` section in the *Commands and System Tasks for SA Mode* chapter of the *VXE Command Reference Manual* for more detail.

© 2021 All Rights Reserved.

Accessing Two-Value Bit Values in C/C++

Up to 32-bit values are directly accessed as signed or unsigned `int`. When a task or function specifies a two-value bit vector type argument of width > 32 bits, its values are accessed as an array of 32-bit values on C/C++. In this case, the argument in C/C++ should be declared as `svBitVecVal *`, which is defined as a reference to an array of 32-bit unsigned `int` in `svdpi.h`.

Each element of the array starting from 0 represents the next 32 bits of the bit vector. The number of elements in the array is dependent on the size of the vector, such that it is split into 32-bit elements. The bit-wise values of an argument of type `svVecVal` can also be accessed using utility functions such as `svGetBitselLogic()`, `svPutBitselLogic()`, `svGetPartselLogic()`, `svPutPartselLogic()`, and so on. These and other such functions are defined in `svdpi.h`.

Consider the following examples

Verilog Example:

```
import "DPI" context function bit equal \
       (input bit [47:0] ina, input bit [47:0] inb);
```

C Example:

```
// Implement logical equality function in C
int equal (svBitVecVal *ina, svBitVecVal *inb)
{
  unsigned int p[2]; // two unsigned ints to store 48 bits of vector
                 //information
  p[0] = ina[0];  // lower 32 bits
  p[1] = ina[1];  // upper 16 bits
  unsigned int q[2];
  q[0] = inb[0];
  q[1] = inb[1];
  return((p[0] == q[0]) & (p[1] == q[1])); // return value of function
}
```

Accessing Four-Value Logic Values in C/C++

When a task or function specifies a four-value logic type argument, its four values are accessed as 2-bit information (known as "`aval`", "`bval`") per bit on C/C++.

- For a scalar input argument in HDL, the argument in C/C++ should be declared as unsigned `char` or predefined `svLogic` type. The four values of the scalar are enumerated in `svdpi.h` as `sv_0`, `sv_1`, `sv_z`, and `sv_x` corresponding to unsigned values of `0x0`, `0x1`, `0x2`, and `0x3`, respectively.
- For a scalar output argument of a task or scalar return value of a function in HDL, the argument in C/C++ should be declared as unsigned `char*` or `svLogic *`. The value

© 2021 All Rights Reserved.

is then assigned into the dereferenced variable using the same four values' system or enumerated types, as shown previously.

- For a vector input/output argument in HDL, the argument in C/C++ should be declared as `svLogicVecVal *`, which is defined as a reference to an array of `p_vpi_vecval` structures in `svdpi.h`. Each element of the array starting from 0 represents the 32 bits of vector four-value with the structure member *a* having *avalues* and member *b* with *bvalues* to make the four-value logic. The number of elements in the array is dependent on the size of the vector such that it is split into 32 bits of the vector per element. The bit wise values of an argument of type `svLogicVecVal` can also be accessed using utility functions such as `svGetBitselLogic()`, `svPutBitselLogic()`,`svGetPartselLogic()`,`svPutPartselLogic()`, and so on. These functions are defined in `svdpi.h`.

For example:

Verilog:

```
import "DPI" context task mult (input [3:0] ina, input [3:0] inb, output [4:0] out);
```

C:

```
// implement multiplier task in C
void mult (svLogicVecVal *ina, svLogicVecVal *inb, svLogicVecVal *out)
{
  unsigned int p = ina->a;   // Assume logical 0,1 value in aval
  unsigned int q = inb->a;
  out->a = p*q;     // sets aval of return value
  out->b = 0;       // make sure bval is zero (no z, x)
}
```

export Functions and Tasks

`export` functions or tasks are defined in the SystemVerilog layer and called in the C/C++ layer.

Example: C/C++ Layer Calling SystemVerilog Function

The following example shows an exported task, that is, a task declared in SystemVerilog and then called in C/C++.

Here is the declaration:

```
export "DPI-C" task          rom_read;
export "DPI-C" task          rom_write;
```

Note: The arguments of exported functions or tasks are not part of the declaration; the arguments are compiled directly based on task definitions in Verilog, for example:

© 2021 All Rights Reserved.

```
// exported tasks
task rom_read(input int addr, output int data);
begin
  $axis_mread(test_pytha.u1.ang1.rom1.rom, addr, data);
  $display("%m: ROM Read from DPI addr=%x, data=%x", addr, data);
end

endtask

task rom_write(input int addr, input int data);
begin
  $display("%m: ROM Write from DPI addr=%x, data=%x", addr, data);
  $axis_mwrite(test_pytha.u1.ang1.rom1.rom,addr,data);
end

endtask
```

When these tasks have been exported from Verilog, they can be called from C/C++ as follows:

```
extern void rom_read(int addr, int *data);
extern void rom_write(int addr, int data);
void setup_atan_rom()
{
  int i;
  unsigned romd;
  // for a 15 bit (inx/iny) division result of x & y values, calculate atan ROM
values
  for(i=1;i<=31;i++)
  {
    romd = 90-180*(atan(i))/M_PI;
    io_printf("PYTHA-DPI Rom Setup @ addr=%x data=%x\n",i, romd);
    rom_write(i,romd);
  }
}
```

In this example, the C/C++ function `setup_atan_rom()` calls the `rom_write()` task exported from Verilog to perform a write into a design memory using a direct cross-module reference in Verilog. The `setup_atan_rom()` is used in the Pythagorean example to set up the lookup ROM for converting from X/Y ratio to angle in degrees.

The C/C++ function `setup_atan_rom()` itself is called from Verilog at initial time as an SV-DPI task using `import`:

```
import "DPI-C" context task         setup_atan_rom;
  initial
    begin
      $axis_set_clkgen("clk", 0, 50, 100, 100);
        setup_atan_rom;
        count = 0; errors = 0; i=0; reset_=0;
        reset_=0;#1812 reset_=1;
    end
```


© 2021 All Rights Reserved.

Using SystemVerilog DPI

Using the SystemVerilog DPI is similar to using the Programming Language Interface (PLI). As with the PLI, first compile your C/C++ code using your preferred C/C++ compiler; then, pass the generated object file to the compiler. Unlike PLI, you do not need the PLI table file (`veriuser.c`).

You need to pass a new option, `+sv`, when you start ixcom.

```
ixcom [options] <verilog_files> -pl "<dpi_object_files_or_archives>" +sv
```

Typically, a module with export/import DPI scopes is not considered for behavioral compilation unless it includes other constructs requiring behavioral compilation. If such a module needs to undergo behavioral compilation, explicitly specify its name with the `+bc` option of the `ixcom` command, or include the `$map_bc` directive in that module.

Following are the steps to run a DPI example for 32-bit mode:

1. Ensure that your environment has been set up for 32-bit compilation. Set the library path variable `LD_LIBRARY_PATH` to the 32-bit path for IXCOM run-time library, for example:

   ```
   /tools.lnx86/lib/32bit:/grid/cva/test_ius/ius.9.2.s14/tools.lnx86
   /tools.lnx86/lib
   ```

2. Compile the C/C++ code and pass the generated object file to the compiler.

   ```
   gcc -m32 -c -o counter.o counter.c
   gcc -m32 -shared -o libdpi.so counter.o
   ```

3. Invoke `ixcom`.

   ```
   ixcom -f list.f +dut+counter_t_ff +timescale+1ns/10ps -32 +sv
   ```

   Note: By default, compilation occurs in 64-bit mode. You must explicitly pass `-32` option to force the 32-bit compilation.

4. Invoke xrun, passing the `libdpi.so` file using the `-sv_lib` option.

   ```
   xrun -R -TCL -sv_lib libdpi.so
   ```

SystemVerilog DPI Keywords in Verilog

SystemVerilog DPI introduces the following keywords to Verilog:

- `DPI`
- `import`
- `export`
- `pure`

© 2021 All Rights Reserved.

- `context`

Using SystemVerilog DPI with Emulator

When you are mapping your design for the emulator, the method you use depends on the type of SystemVerilog DPI function you are using — `import` or `export`.

SystemVerilog DPI Import Functions with Emulator

You can call a SystemVerilog DPI import function from the module under the DUT. A special flag is not required to map the design, and the module is automatically treated as behavioral module. This is same as using the `+bc+<module>` option. The DPI import functions are always executed on the simulator. Refer to `+bc` in the Tabular Reference for more details about the `+bc` option.

SystemVerilog DPI Export Functions with Emulator

By default, compiler treats a SystemVerilog DPI export function in the same way as a DPI import function. A special flag is not required to map the design, and the module is executed on the simulator. To map an export function or task to the emulator, use the `tb_export` compiler directive. In this case, SystemVerilog DPI export functions and tasks have the same language restrictions as normal tasks and functions in DUT.

Generating C Header Files for Exported DPI

DPI allows direct inter-language calls between the languages on either side of the interface. The subroutines (tasks or functions) implemented in the foreign language can be called from SystemVerilog and are referred to as imported DPI routines. On the other hand, the SystemVerilog subroutines that are to be called from a foreign code are referred to as exported DPI routines. DPI allows for passing SystemVerilog data between the two domains through arguments and results.

This section describes how you can generate the C/C++ prototypes of the import and export DPI routines so that the definition of import DPI routines and the invocation of export DPI routines in the C/C++ domain can match the declarations of those DPI routines in the SystemVerilog domain.

IXCOM provides support for the `-dpiimpheader` and `-dpiheader` Xcelium options. These options generate the prototypes of the DPI routines and the referenced user-defined types of the arguments in the header file.

© 2021 All Rights Reserved.

Consider the following example of import and export DPI routines.

SystemVerilog Code

```
import "DPI-C" function int C_func(logic ctrl, bit[31:0] bv);
export "DPI-C" function SV_func;
function int SV_func(inout byte B, logic[31:0] lv);
...
endfunction

reg C;
bit[31:0] bv32;
int out;
always @(posedge clk) begin
       out = C_func(C, bv32);
end
```

C Code

```
// Prototype of export DPI function SV_func
extern int SV_func(char *B, svLogicVecVal *lv);
// Prototype of import DPI function C_func
extern int C_func(unsigned char ctrl, svBitVecVal *bv);

int C_func(unsigned char ctrl, svBitVecVal *bv)
{
svLogicVecVal lv;
char B = 0;
lv.a = *bv; // data bit(s)
lv.b = 0;   // control bit(s)
int ret = SV_func(&B, &lv); // export DPI call
return ret;
}
```

In the above example, `C_func` is an import DPI function, which is called from a SystemVerilog `always` block, but defined in the C domain. A SystemVerilog function `SV_func` is declared as an export DPI function, and can be called (in the `C_func` function) from the C domain.

The declarations of DPI routines (import and export) are provided in SystemVerilog. However, the definition of the import DPI routines and the invocation of the export DPI routines are in the C/C++ domain. Most SystemVerilog simulators have the functionality of dumping the prototypes of all import or export DPI routines in one C/C++ header file so that you can implement the import DPI routine and invoke the export DPI routine with the correct arguments.

In addition to the prototypes of the DPI routines, the referenced user-defined types of the arguments are dumped to the header file as well. For example, consider the following DPI declarations:

```
typedef struct {
    logic[63:0] mem32x64[32];
    bit[63:0]   bv64;
```


© 2021 All Rights Reserved.

```
    byte        B;
    longint     li;
} MyStruct;

import "DPI-C" function int C_struct(MyStruct s)
import "DPI-C" function int C_func(logic ctrl, bit[31:0] bv);
export "DPI-C" function SV_func;
export "DPI-C" function SV_struct;
function int SV_func(inout byte B, logic[31:0] lv);
function int SV_struct(MyStruct st);
```

The `-dpiheader` option generates the following header file:

```
#include "svdpi.h"
typedef struct {
    svLogicVecVal mem32x64[32][2];
    svBitVecVal   bv64[2];
    char          B;
    long long     li;
} MyStruct;

extern int SV_struct(MyStruct *s);
extern int SV_func(char *B, svLogicVecVal *lv);
```

The `-dpiimpheader` option generates the following header file:

```
#include "svdpi.h"

typedef struct {
    svLogicVecVal mem32x64[32][2];
    svBitVecVal   bv64[2];
    char          B;
    long long     li;
} MyStruct;

extern int C_struct(MyStruct *s);
extern int C_func(unsigned char ctrl, svBitVecVal *bv);
```

These options generate the DPI header files for the C/C++ prototypes and referenced user-defined types for all of the DPI routines declared in the SystemVerilog code and support all the LRM data types for DPI. The generated prototypes of the DPI routines comply with the DPI interface specified in SystemVerilog LRM IEEE 1800-2012 Ch. 35 and Annex H.

These prototypes are divided into two files:

- One for import DPI routines
- One for export DPI routines

The names of these files are as per the file names specified with the respective options (`-dpiimpheader` and `-dpiheader`). If the file name is omitted, the default header file names for the import and export DPI routine(s) are `xedpiimpheader.h` and `xedpiheader.h`, respectively. The header file for the prototypes of import (or export) DPI routines is generated only if the `-dpiimpheader` (or `-dpiheader`) option is specified.

```
ixcom ... -dpiimpheader <import_DPI_header_file> \
          -dpiheader <export_DPI_header_file> ...
```


© 2021 All Rights Reserved.

If the specified file exists, it is overwritten with the new output.

According to the LRM, the prototype of the C layer of a DPI task should have int return type. To make the C layer have no return value, use the `-dpi_void_task` option to make the prototype of a DPI task, a void function.

Both Xcelium and IXCOM generate LRM-compliant prototypes of the DPI routine. If the LRM does not provide a clear specification, IXCOM adheres to the behavior of Xcelium. If a DPI routine contains unsupported data types or constructs, an error is generated.

According to the LRM, every subroutine which is imported to SystemVerilog, eventually resolves to a global symbol. Similarly, every subroutine exported from SystemVerilog defines a global symbol. Global names of imported and exported tasks and functions are unique (no overloading is allowed) and follow C conventions for naming. Also, all user-defined data types of the same name should have the same definition. Otherwise, an error is issued.

For example, the following three declarations of the same import DPI function `foo` are regarded as different declarations. Argument `mem` in the second declaration is different from that of the first declaration because these arguments have different element types (one is `int`, and the other is `bit[31:0]`). Also, the argument `mem` in the third declaration is different from that of the first one because of a different number of unpacked dimensions.

```
import "DPI-C" function int foo(bit[31:0] mem[7:0]);
import "DPI-C" function int foo(int mem[7:0]);
import "DPI-C" function int foo(bit[31:0] mem[3:0][1:0]);
```

If an argument of a DPI routine is of an array type (packed or unpacked) and its types in all declarations of the same DPI routine have the same data types and matching number of dimensions, but the widths or the directions of some dimensions (packed or unpacked) are different, then, a warning message is issued. The following is an example of the warning:

```
import "DPI-C" function int bar(bit[31:0] mem[7:0]);
import "DPI-C" function int bar(int[31:0] mem[0:7]);
import "DPI-C" function int bar(bit[32:1] mem[7:0]);
```

The generated DPI declarations include the referenced user-defined unpacked struct types. Only one definition for each user-defined struct type identified by its name, is generated. This means that all of the user-defined data types of the same name referenced in all of the DPI declarations should have the same definition. Otherwise, an error is issued. Two user-defined struct types are considered as same only if each member of the struct types has the same member name, the same data types, and the array types have the same width, range, and direction of each dimension as that of the other struct type.

Following is an example of this error:

```
module M1;
typedef struct {
int       tag;
bit[31:0] data1;
```


© 2021 All Rights Reserved.

```
} MyStruct;
import "DPI-C" function int C1(MyStruct s)
...
endmodule

module M2;
typedef struct {
bit[31:0] data2;
int       tag;
} MyStruct;
import "DPI-C" function int C2(MyStruct s)
...
endmodule
```

In the above example, both `M1` and `M2` modules have DPI declarations with reference to a user-defined datatype MyStruct. However, the definitions of the `MyStruct` datatype in the two modules do not match, resulting in an error.

The packed struct types referenced in DPI declarations are mapped to either `svBitVecVal` if the struct types are 2-state or to `svLogicVecVal` if the struct types are 4-state in C. The original struct is not dumped. This check is only to determine if the 2-state or 4-state of two packed struct types of the same argument match.

Support for Open Arrays in DPI

This section describes the support for open arrays for packed and unpacked dimensions in import DPI tasks or functions. This support is particularly useful in TBA (Transaction-Based Analysis) applications because the SCE-MI standard requires the use of open arrays for supporting SystemVerilog adaptors.

Example: Using Open Arrays in DPI

Following is a basic example to illustrate the use of open arrays in an import DPI function. The second parameter, `v`, of the function `foo` has an open-packed dimension. So, the SystemVerilog code can call the import function with an arbitrary-sized bit vector as its argument. Notice the two function calls inside the initial process to pass arguments of different sizes.

```
/* SystemVerilog code */
module top;
    import "DPI-C" function void foo(int id, bit [] v);
    bit [31:0] b1 = 32'h1234;
    bit [63:0] b2 = {32'h5678, 32'h1234};
    initial begin
        foo(1, b1); // call foo with 32-bit vector
        foo(2, b2); // call foo with 64-bit vector
    end

endmodule
```


© 2021 All Rights Reserved.

The C function `foo` should be defined so that the variable-sized open-array argument is properly handled. This can be achieved by the use of query functions, such as `svGetArrayPtr`, `svLength`, `svLeft`, `svRight`, and so on.

```
/* C code */
#include "stdio.h"
#include "svdpi.h"
int foo(int id, svOpenArrayHandle h)
{
    int i = 0;
    unsigned *v = svGetArrayPtr(h);
    int sz = svLength(h, 0);
    printf("Call DPI %d %d ", id, sz);
    while(sz > 0){
           printf(" %x ", v[i]);
           sz -= 32; i++;
    }
    printf("\n");
    return 0;
}
```

Querying and Accessing Open Arrays from the C Code

To support the use of open-array dimensions in import DPI scopes, methods for querying information about open dimensions from C code must be provided. Otherwise, the C code cannot handle variable-sized arguments adaptively.

As an example, consider a DPI `import` function declared as follows:

```
import "DPI-C" function void foo(bit [] v);
```

Given this declaration, the SystemVerilog code can call the function `foo`, with bit vectors of different sizes (for example, `bit [3:0] arg1`, `bit [31:0] arg2`). To support the actual arguments of different sizes, the definition of the function `foo` needs to figure out the size of the open-packed dimension. For this, a query function `svLength` can be used.

Typically, the actual call from the SystemVerilog code to the import C function is fully controlled by Xcelium, no IXCOM support is required in the implementation of import DPI function calls. Xcelium, at the time of function call, it will transfer control from the SystemVerilog code to the body of the called C function, marshall or unmarshall all arguments to and from the C function, and support any query functions that can be used inside the C function. So, you can use any query functions as defined in the `svdpi.h` file supported by Xcelium.

To access the elements of open unpacked arrays, the `svGetArrElemPtr` function needs to be used:

```
void *svGetArrElemPtr(const svOpenArrayHandle h, int idx);
```


© 2021 All Rights Reserved.

To access the elements of open unpacked arrays, a different access method is required because the actual layout of the SystemVerilog array passed as an argument for open unpacked arrays is different from the C layout. This function returns the pointer to the element at the given index of the open-unpacked dimension. Only one unpacked dimension is supported.

```
// import "DPI-C" function void foo(input bit [3:0] p []);
int foo(const svOpenArrayHandle p)
{
    int p_low = svLow(p, 1); // lower bound of unpacked dimension
    int p_high = svHigh(p, 1); // higher bound of unpacked dimension
    for (int i = p_low; i <= p_high; i++) {
             unsigned *elem = (unsigned *) svGetArrElemPtr(a, i);
             printf("p[%d] = %x\n", i, *elem);
}
```

When a pointer to the array element is obtained, it is possible to both read and write to the memory location of the array element.

```
void svPutBitArrElemVecVal(const svOpenArrayHandle h, const svBitVecVal*,
                           int idx);
```

This function allows the packed element of an open array to be updated. Note that only one dimension is required because only one open-unpacked dimension is supported.

Given a handle and a dimension, the following function returns the left bound of the actual argument in the given dimension. When dimension `d` is `0`, the query refers to the packed part of the array. When dimension `d` is `1`, it refers to the unpacked part of an array.

```
int svLeft(svOpenArrayHandle h, int d)
```

Given a handle and a dimension, the following function returns the right bound of the actual argument in the given open dimension.

```
int svRight(svOpenArrayHandle h, int d)
```

Given a handle and a dimension, the following function returns the lower value of the left and right bound of the actual argument in the given dimension.

```
int svLow(svOpenArrayHandle h, int d)
```

Given a handle and a dimension, the following function returns the higher value of the left and right bound of the actual argument in the given dimension.

```
int svHigh(svOpenArrayHandle h, int d)
```

Given a handle and a dimension, the following function returns the size of the range in the given open dimension. The return value is equivalent to |`svLeft(h, d) - svRight(h, d)+1`|.

```
int svSize(svOpenArrayHandle h, int d)
```


© 2021 All Rights Reserved.

Limitations in Using Open Arrays in DPI

Due to language constraints, as specified in SystemVerilog LRM, limitations in Xcelium or IXCOM, only a subset of all possible open arrays can be used as formal parameters in DPI scope declarations. This section summarizes such restrictions assuming that DPI scope is used inside a DUT module.

- Open arrays can be only used in imported DPI scopes.
- Only formal parameters of imported DPI functions or tasks can be specified as open arrays.
- An exported function or task cannot have open arrays as its formal parameters.

When using open arrays in import DPI calls, the C function definitions need to use query functions, such as svLeft and svRight to determine the range of the open dimension in actual arguments. Typically, these query functions are supported by Xcelium. For Pthreads and SystemC CVPs, IXCOM provides its own query functions which thread functions or SystemC functions can use. However, When the CVP is in a third-party SystemVerilog code, there is no way to override the DPI functionality embedded in the third-party SystemVerilog simulator.

Using Open Arrays in Packed Dimensions

The SystemVerilog LRM states that when an open dimension is used in a packed dimension, the open dimension must be the only packed dimension in the formal parameter. Therefore, it is not possible to have two or more open-packed dimensions. Also, Xcelium further constrains that such packed dimension must be either a bit vector or a logic vector.

```
bit [] p;               // allowed;
reg [] p;               // allowed
logic [] p;             // allowed
byte [] p;              // not allowed; illegal as per the LRM
bit [3:0][] p;          // not allowed;
                        // open-packed dimension must be the only packed dimension
```

Using Open Arrays in Unpacked Dimensions

Xcelium allows up to one open unpacked dimension in formal parameters of import DPI scopes.

```
int p []; // allowed
shortint p [];            // allowed
longint p [];             // allowed
byte p [];                // allowed
bit p [];                 // not allowed; use bit [] p
reg p [];                 // not allowed; use reg [] p
logic p [];               // not allowed; use logic [] p
bit [3:0] p [];           // allowed
reg [3:0] p [];           // allowed
```


© 2021 All Rights Reserved.

```
logic [3:0] p [];         // allowed
bit [] p [];              // not allowed
reg [] p [];              // not allowed
logic [] p [];            // not allowed
int p [][];               // not allowed; only one unpacked dimension is allowed
logic [] p [][];          // not allowed
```

Supported Set of DPI Functions

SystemVerilog LRM defines the set of DPI functions that C code can use. The list of such functions can be found in the `svdpi.h` file in the Xcelium product installation hierarchy.

```
void *svGetArrayPtr(svOpenArrayHandle h);
void *svGetArrElemPtr(svOpenArrayHandle h, int idx);
void svPutLogicArrElemVecVal(const svOpenArrayHandle h,
                             const svLogicVecVal *v, int indx1, ...);
void svGetLogicArrElemVecVal(svLogicVecVal *v, const svOpenArrayHandle h,
                             int indx1, ...);
void svPutBitArrElemVecVal(const svOpenArrayHandle h, const svBitVecVal *v,
                           int indx1, ...);
void svGetBitArrElemVecVal(svBitVecVal *v, const svOpenArrayHandle h,
                           int indx1, ...);
int svLeft(svOpenArrayHandle h, int d);
int svRight(svOpenArrayHandle h, int d);
int svLow(svOpenArrayHandle h, int d);
int svHigh(svOpenArrayHandle h, int d);
int svLength(svOpenArrayHandle h, int d);
```


© 2021 All Rights Reserved.

32

Analyzing Power Consumption of the Designs

You can analyze and feed power information back into the design at every step. The underlying principle of power analysis to calculate the power consumed by a circuit that has the following three contributing parts:

- **Leakage power** that reflects the ***logic state*** of the device. It is dependent on the leakage current that flows between the source and drain.
- **Internal power** that is the measure of power used by the logic gates (cells) when a driver is switching, that is, ***toggle rate*** * *cell transition power* (which is the power used by a cell when there are transitions on the input pins).
- **Net power** that shows the impact of net loading when a driver is switching, that is, *1/2 * capacitive loading * (voltage level)*^{2}* ***toggle rate***.

Dynamic power is the sum of *internal power* and *net power*; while total power is the sum of *dynamic power* and *leakage power*.

So, when performing Dynamic Power Analysis (DPA), you estimate the total power consumed by a design to identify the power peaks that can break the functionality of the design and segregate the highly active modules. There are following two basic approaches to DPA:

- Toggle Count Analysis (TCA) that generates a toggle count profile capturing the design activity and analyzes the histograms of the activity over time using the xeDebug DPA GUI.

  Palladium Z1 supports analysis of normal and weighted toggle counts to enable identification of areas for detailed power analysis. For details, refer to RTL Toggle Count Analysis (that uses normal toggle counts) and Weighted Toggle Count (WTC) Analysis.
- Palladium Z1 also enables you to generate Switch Activity Interchange Format (SAIF) files for analyzing power using third-party tools. This chapter describes the process of generating SAIF files in Generating Backward SAIF Files for Power Analysis by Third-Party Tools, but does not covers the process of reading SAIF files using third-party tools because that depends on the tool used.

© 2021 All Rights Reserved.

Considerations when Using DPA

- DPA operations are supported in both ICE and SA modes. However, in SA mode, DPA support is limited to the DUT part; the testbench part is not supported. Consequently, you can perform DPA operations in SA mode only after the design has been swapped to the hardware.
- DPA does not work for designs compiled in 0.5X mode.
- DPA can be used only with the FullVision vision mode.
- DPA is not supported with uncontrolled clocks.
- DPA is not supported in dual clock domain wherein a design has both controlled and uncontrolled clocks.
- You require a special license to use the Palladium Z1 DPA functionality. This license includes access to five licensed features for different DPA functions and a SimVision license. Contact the Cadence representative to obtain the necessary licenses. For more information, refer to DPA Licenses.

Identifying Activity-Based Power Analysis Components

For DPA, you need the following three components:

- A netlist that describes the interconnectivity of cells. The netlist can be in either of the following two forms:
  - RTL files generated after synthesizing the RTL designs using the Palladium Z1 HDL-ICE Compiler or IXCOM Compiler
  - Gate-level netlist that also has the associated device library

  The following sections of the chapter discuss the different methods of DPA based on the type of netlist you have:
  - Analyzing Power Consumption of RTL Designs
    - RTL Toggle Count Analysis
  - Analyzing Power Consumption of Gate-Level Netlists
    - Normal Toggle Count (NTC) Analysis
    - Weighted Toggle Count (WTC) Analysis
- An activity file that describes the toggle rate and logic state of nets.

© 2021 All Rights Reserved.

The power estimation tool, Joules, accepts the activity file in TCF format. The TCF files contain toggle count, high count, and net name for every net.

Note: You can also generate SAIF files for third-party tools used for power analysis.

- A library that describes how much power is consumed when an input causes a cell to switch its outputs.

  The information about power consumed by a cell when inputs and outputs change state is recorded in device-specific library (`lib`) files that are often in Liberty file format. The library file contains detailed information, such as leakage power for every combination of input to output, and internal power consumed by each type of transition on every input pin.

Important

If you do not have device-specific library files, use the power estimation techniques discussed under Analyzing Power Consumption of RTL Designs.

Figure 32-1 Components Needed for Power Analysis

Palladium Z1 and the power estimation tool, Joules, operate concurrently to generate the final power estimation reports. This release supports Joules 16.21 or later release. For complete information on how to use Joules for power analysis, refer to the *Joules User Guide*.

Palladium Z1 emulator runs at a high speed compared to simulation. It uses the real world stimulus to capture the activity information, and eventually generate either TCF or SAIF

© 2021 All Rights Reserved.

format files over a specific time range. You can also use Palladium Z1 for direct analysis of switching activity with Toggle Analysis tools.

Joules uses the activity information captured by Palladium Z1 and the switching information contained in the device-specific library files to estimate power.

Using DPA Methods in Design Flow

You can follow a multi-step approach to analyze power consumption, starting from initial results derived through RTL toggle count analysis to final results derived through gate-level power analysis.

At each phase of design flow, you can analyze the power information and feed it back into the design. As you progress through the design flow, the accuracy of results derived during power analysis and the processing time involved increase.

Figure 32-2 Design and Power Analysis Flow

Performing DPA in Offline Mode

DPA operations are generally time consuming. Therefore, to free up the emulator hardware for other designs or users, performing DPA operations in offline session is recommended.

You can create a `*.phy` file in online session using the `database -prepareoffline` command or the *Prepare Offline* button on the *Control* tab of xeDebug GUI. The `*.phy` file

© 2021 All Rights Reserved.

contains FullVision information that can be used to get waveforms for all signals in the design and perform DPA operations.

Note: You can still perform DPA operations in an online session. However, you need to first select an existing `*.phy` file using the `dpa -phyfile` command or the *Setup* tab of the xeDebug GUI. For example, the following set of commands enable you to create and then open the `t1.phy` file:

```
run 1000
# Specify the name of the .phy file as t1
database -open t1
# Create the t1.phy file
database -prepareoffline
# Open the t1.phy file
dpa -phyfile t1.phy
# other dpa commands
```

You can save more data to a `*.phy` file even after opening it. However, the DPA operations will not be able to examine the new data. You will have to issue the `dpa -phyfile` command again to save the new data in the `*.phy` file.

Analyzing Power Consumption of RTL Designs

Starting with an RTL design saves time by getting results earlier during the design development life cycle. However, RTL design analysis generates less accurate estimates as compared to gate-level netlists. This is because the netlist examined by Palladium Z1 might not match the synthesized netlist. Power consumption of RTL designs can be analyzed using RTL Toggle Count Analysis.

RTL Toggle Count Analysis

Toggle Count Analysis (TCA) is the most direct method for toggle analysis. This is possible because you can avoid the time-consuming synthesis step and do not require inputs, such as Liberty (`lib`) format files that might not be available. The toggle count captured during TCA shows the total number of nets in the design that toggle between the `logic1` and `logic0` states at a specific cycle. Each cycle during a time window has a toggle count number.

TCA also captures the total number of nets in the design that have high values at a specific cycle. The high values represent a high count.

© 2021 All Rights Reserved.

Important

Both normal and weighted toggle counts are supported. The normal toggle counts show the raw number of internal nets switching at a time unit and are derived from the RTL designs. Weighted toggle counts are derived from the gate-level netlists. For details about the weighted toggle counts, refer to Weighted Toggle Count (WTC) Analysis.

The captured toggle count and high count numbers are plotted as a histogram that represents the switching activity over a certain time. These histograms let you identify the areas of potential high power consumption. Both toggle count and high count correlate to the power consumption of the design. Large toggle count and high count values generally show a high power consumption.

When you perform TCA before doing a full power analysis using Joules, you must isolate logic blocks or time regions with potential power problems. You can trim out the time windows with low design activity and identify windows with high activity out of millions of design cycles. You then zoom into time segments of high activity for a detailed power analysis. Using toggle counts to identify active time windows is faster than using waveforms or VCD files. However, it is less accurate than estimating power consumption with Joules.

Figure 32-3 shows the general flow of how an RTL design moves through the TCA process.

Figure 32-3 Process of Toggle Count Analysis for RTL Design

© 2021
All Rights Reserved.

After compiling the design using the normal compile flow, invoke xeDebug in online session to run the emulation and create the `*.phy` file. Then, open an offline session of xeDebug to do toggle count analysis. The basic steps required for TCA are:

1. Selecting Instances to Collect Toggle Data
2. Collecting Toggle Count Data in an Offline Session
3. Analyzing Toggle Count Data in an Offline Session

Selecting Instances to Collect Toggle Data

Toggle counts are collected on a per instance basis. On each clock cycle, the following information is available in a set of bins:

- Number of nets in an instance that are high
- Number of nets in an instance that toggled

The software can automatically select a default set of instances for toggle count analysis or you can explicitly specify a set of instances using the following `dpa` command:

```
dpa -setupToggleCount -addinst [-top <n>][-depth <n>] [-min <n>] <insts>
```

This command accepts a list of instance names, and adds the specified instances and all their children. If a dot (`.`) is specified for the instance name parameter, then toggle count data is collected for the entire design.

Use the `-top`, `-depth`, and `-min` options to filter children of each specified instance.

For example, the following command adds all the instances within the top seven levels below `dut`, the top instance:

```
dpa -setupToggleCount -addinst dut -depth 7
```

If no instances were selected before starting toggle count data collection, by default, the top `1000` largest instances are selected from the entire design. These instances can be at any level of the hierarchy, and the size of the instances should be more than one percent of the whole design. In most cases, this is adequate to cover all modules that might be of interest.

Collecting Toggle Count Data in an Offline Session

Use the following `dpa` command to generate toggle count data from the `*.phy` file and save the data in the `<name>.ppfdata` file required for performing toggle count analysis:

```
dpa -getToggleCount [-append] [-weighted] [-file <name>]
```


© 2021 All Rights Reserved.

This command can be issued multiple times. For example, the following commands get `1000` cycles of the toggle count data and save it into the file named `ntc.ppfdata`:

```
dpa -tracewindow -start 0 -end 1000
dpa -getToggleCount -file ntc -append
```

Then, issue the following commands:

```
dpa -tracewindow -start 1000 -size 10000
dpa -getToggleCount -file ntc -append
```

The commands above get `10000` cycles of toggle count data and append it into the `ntc.ppfdata` file that now records `11000` cycles of toggle count data.

If this command is given without previously executing a `dpa -setupToggleCount` command, the default values are used.

By default, the existing file data is overwritten every time new data is uploaded. To retain the toggle count data from previous uploads, specify the `-append` option.

When analyzing power for RTL designs, both normal and weighted toggle count data are collected.

Analyzing Toggle Count Data in an Offline Session

Different instances in the design have separate histograms for toggle count and high count. You can compare the histograms of the different instances for better estimates of power patterns and identify the instances that consume the maximum power. You can also request either the average number of nets toggling for a time range or the cycle with largest number of toggling nets to find peak counts for peak analysis.

The `*.ppfdata` file and the histogram data it contains can be analyzed using the xeDebug GUI or the command prompt.

If you opt to analyze the data from the command prompt, open the `*.ppfdata` file using the following `tca` command:

```
tca -setFile <name>.ppfdata
```

The filename should use the absolute path if it is not in the current directory and should include the `.ppfdata` extension.

Then, you can use the following `tca` command options for the analysis:

```
tca -getAllInstanceNames [-maxDisplay <number>]
tca -getTimeRange
tca -getValues <list_of_inst_names> -start <start_time> [-end <end_time>]
[-type toggle|high] [-count ave|max]
tca -getTopNInst <number> [-start <start_time>] [-end <end_time>]
[-type toggle|high] [-count ave|max]
```


© 2021 All Rights Reserved.

```
tca -getTopNPeak <number> [-sensitivity <N>] [-start <start_time>]
[-end <end_time>] [-type toggle|high] [<list_of_instance_names>]
tca -getTopNSlope <topN_number> <instance_list> [-start <start_time>] [-end
<end_time>] [-type toggle | high] [-step m] [<[fileID] inst>…]
tca -export <waveform_name> [-start <time>] [-end <time>] [-format sst2]
[-filter <filtername><filter_arguments>}] <list_of_instance_names> [-depth
<n>] <list_of_instance_names> [-depth <n>] <list_of_instance_names>
```

For detailed information about the syntax and options of the `tca` command, refer to the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

The waveform can be exported for viewing in SimVision using the `tca -export` command.

Note: In SA mode, the waveform shows the time advances in unevenly steps. Also, when there are multiple toggle samples captured for the same point of time, all except the last sample are ignored. For example, consider the following table that shows the data captured in a waveform:

Time	0ns	0ns	0ns	5ns	10ns	10ns	30ns	30ns	35ns	50ns	80ns	100ns
Value	0	1	1	1	0	1	0	0	1	1	0	1

Here, only the following values are considered: 1 (at 0ns), 1 (at 5ns), 1 (at 10ns), 0 (at 30ns), 1 (at 35ns), 1 (at 50ns), 0 (at 80ns), and 1 (at 100ns). This shows that there were four toggles during the time window starting from 0ns to 100ns.

Viewing Toggle Percentage of State Elements

You can view the toggle percentage of state elements (that is, flops, latch and memories outputs) for the specified instances. The system has been enhanced to enable you to observe this data through the `*.ppfdata` file during Normal Toggle Count (NTC) analysis.

To get the toggle percentage of state elements in the `*.ppfdata` file:

1. Compile your design in SA or ICE mode, as required.

   Note: `powerdb` files or liberty files are not required in the compile script for xeCompile. That is, you do not need to specify the `compilerOption -add {powerdb <powerdb_directory>}` command.

2. In the xeDebug offline session, do the following:

   a. Execute the following command to collect the instances for which the toggle percentage data needs to be computed for state elements:

      `dpa -setupToggleCount-addinst <instances_names>`

© 2021 All Rights Reserved.

b. Execute the following command with the `-sntc` option to generate the `ppfdata` file that will contain the toggle percentage data for the state elements pertaining to the instances specified with the `dpa -setupToggleCount` command:

```
dpa -getToggleCount -sntc -file <ppf_file_name>
```

3. Observe the toggle percentage by issuing the `tca` commands, as follows:

```
tca -setFile  <ppf_file_name>.ppfdata
tca -export  <result_file_name> -format text  <list_of_instance_names>
```

Example:

The following code shows an example of a run script in offline debug mode:

```
    debug .
host -offline trace.phy
…
dpa -setupToggleCount -addinst dut.instanceA
dpa -getToggleCount -sntc -file ppffileXYZ>
tca -setFile ppffileXYZ.ppfdata
tca -export resultfileABC -format text dut.instanceA
```

The following figure shows the toggle percentage of state nets on each cycle for the specified instance `dut.instanceA`. For example, the toggle value 30 on cycle 2192 indicates that 30% of state elements toggle on cycle 2192 in `dut.instanceA`. The high field is set to zero and could be ignored.

```
# realTime - currentTime - toggle - high
# instance: dut.instanceA
..
...
2188 2188 20 0
2189 2189 0 0
2190 2190 10 0
2191 2191 0 0
2192 2192 30 0
2193 2193 0 0
2194 2194 10 0
2195 2195 0 0
2196 2196 20 0
2197 2197 0 0
2198 2198 10 0
2199 2199 0 0
2200 2200 40 0
```

See Also:

- Selecting Instances to Collect Toggle Data
- Collecting Toggle Count Data in an Offline Session
- Analyzing Toggle Count Data in an Offline Session

© 2021 All Rights Reserved.

For detailed information about the syntax and options of the `dpa` and `tca` commands, refer to the description of these commands in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Analyzing Power Consumption of Gate-Level Netlists

Generally a multi-step approach is followed in analyzing power consumption from gate-level netlists. The gate-level netlist can be created by Joules or any other synthesis tool.

To achieve early results use toggle analysis that involves collection of toggle rates over different time ranges for different logic blocks. Analyzing both normal and weighted toggle counts help you to identify areas that require detailed power estimation. You can do a final power analysis on the post-layout structural netlist with complete clock trees, wire load models, parasitic, and layout related timing.

Gate-level netlists enable you to achieve various levels of accuracy. As you progress through the sequence, the accuracy of results increases as does the processing time.

This section covers the following methods of power analysis that you can perform with gate-level netlists:

- Normal Toggle Count (NTC) Analysis
- Weighted Toggle Count (WTC) Analysis

Normal Toggle Count (NTC) Analysis

Analysis of normal toggle counts can be used to get an early look at the design activity. It provides the most direct method for analyzing the power consumption. It can be done using just Palladium Z1 and does not require Joules.

Gate-level normal toggle count analysis is the same as RTL toggle count analysis. The only difference is that it begins with a gate-level netlist. For detailed information about the process, refer to RTL Toggle Count Analysis.

Weighted Toggle Count (WTC) Analysis

The **weighted Toggle Count** (WTC) is the sum of the weights on internal nets switching at a time unit. WTC gives a higher toggle rate to a net transition that consumes more power. For instance, a state transition on a memory address pin consumes more power than a transition on a `NAND` gate.

© 2021 All Rights Reserved.

In WTC analysis, each internal net driven by a gate is assigned a weight based on its power arc. The assignment of weights is done using the power profile database that comprises the power reports derived from the Liberty format (`lib`) files by Joules and the `powerdb.index` file. If a net driven by gate is not described in any Joules-generated power report, the net is assigned a default base weight `1` for WTC analysis. You can change the power characteristics of a specific net to affect its weight by defining either `gate_power` or `power_file` keyword in the `powerdb.index` file. For information about these keywords and usage details, refer to the following sections:

- Adding gate_power Keyword to the powerdb.index File
- Adding power_file Keyword to the powerdb.index File

The basic steps for WTC analysis are:

1. Generate power reports from the Liberty format files using Joules and create the `powerdb.index` file using a text editor. The `powerdb.index` file labels each of the Liberty files as either gate or blackbox. For procedural details, refer to Creating a Power Profile Database.
2. Analyze, elaborate and compile the gate-level netlist. Refer to Applying the Power Database.
3. Start the emulation and create the probe file.
4. Select the modules for which toggle data needs to be collected:

   ```
   dpa -setupToggleCount -addinst [-top <n>][-depth <n>] [-min <n>] <insts>
   ```

5. Collect toggle count data, using the following command:

   ```
   dpa -getToggleCount -weighted [-append] [-file <name>]
   ```

   This creates a file named `<name>.ppfdata`.

   Note: When the compiler implements the ASIC cells in the design, it can create internal nets that do not exist in the design, below the ASIC level. When collecting normal toggle count data, the toggle count includes those internal nets. However, when collecting weighted toggle counts, those internal nets will be excluded.

6. Use the xeDebug GUI or `tca` command to analyze the toggle count data as with normal toggle counts. Refer to Analyzing Toggle Count Data in an Offline Session.

Creating a Power Profile Database

Palladium Z1 uses the information from the power profile database directory to add weights and netlist information to the emulator database. To create a power profile database, you must have the following:

© 2021 All Rights Reserved.

- Power models (Liberty format `*.lib` files) for ASIC cells and macros in the design
- Palladium Z1 models for the ASIC cells and macros

Note: Macros are complex cells, such as memories, which have complex power models.

Then, synthesize the design for power profile database using a tool like Joules. This process differs from synthesizing to Palladium Z1 primitives using xeCompile because a power profile does not include the testbench. When you have a testbench, you must first create a power table, and then use xeCompile to synthesize your testbench, ASIC cell and macro Palladium Z1 models, followed by compile.

Note: A power table needs to be generated only once for every Liberty file used in the design.

Generating a Power Table File

For generating a power table file, first collect all power models (Liberty format `*.lib` files) for the ASIC cells and complex cells in the design. The power models describe the power characteristics of the cells that can be of two types: simple and complex. Simple cells include logic gates such as AND, OR, XOR, INV, BUF, and so on.

Then, sort the `.lib` files based on cell type. A file can contain multiple cell descriptions, but simple cells and complex cells must not be mixed in a file.

If a cell is marked as a simple cell, the weighted toggle count of that cell is computed in a faster manner. If a cell is marked as a complex cell, the weighted toggle count of that cell is computed in a slower but more accurate manner. Cadence recommends that you mark simple combinatorial gates as simple cells and mark memory as complex cells.

When the cells are sorted, execute the `create_power_table` command as follows to generate the power table file from the `*.lib` file:

```
create_power_table <*.lib_file_name> <power_table_file_name>
```

The generated power table is encrypted.

Creating a Power Database File (powerdb.index)

Save all power table files in a single directory that becomes your power profile database. In that same directory, create a `powerdb.index` file that describes how the Palladium Z1 compile process should treat the cells in each power table (`*.rpt`) file, that is gates or blackbox. Each power table file must be described in the `powerdb.index` file by using the following format for the entries:

- `gates=<power_table_file_for_simple_cells>`

© 2021 All Rights Reserved.

- `black_box=<power_table_file_for_complex_cells>`

Note: Cadence recommends to mark power table files containing complex cells or memories as blackbox, and power table files containing combinatorial and simple sequential cells as gates.

Following is an example of a `powerdb.index` file:

```
# power database
gates = standard_cell.rpt
black_box = memory_cell1.rpt
black_box = another_macro.rpt
```

You can also add the `gate_power`, and `power_file` keywords into the `powerdb.index` file for changing the power characteristics of a specific pin of a cell to affect its weight during WTC analysis. The sections below describe the syntax of these keywords.

Adding gate_power Keyword to the powerdb.index File

You can use the `gate_power` keyword to define the power values of only simple gate cells for WTC analysis.

The `gate_power` keyword can be specified using either of the following syntax:

- `gate_power=<Lib_Name>:<Cell_Name>:<Pin_Name>:<PowerValue>`

  This means, assign the `<PowerValue>` as the power value for the pin, `<Pin_Name>`, of the cell, `<Cell_Name>`, in the library, `<Lib_Name>`.

  For example, to assign power value `15.0` to the pin `Y` of the cell `nand2` in the library `ASIC_LIB`, define the following in `powerdb.index` file:

  ```
  gate_power=ASIC_LIB:nand2:Y:15.0
  ```

  An error is reported in either of the following conditions:

  - The specified pin does not exist in the specified cell.
  - The specified cell does not exist in the specified library.
  - The specified cell is a blackbox cell defined in an imported power table file.

- `gate_power=<Lib_Name>:<Cell_Name>:*:<PowerValue>`

  This means, assign the `<PowerValue>` as the power value for ***all*** pins (as defined using the asterisk) of the cell, `<Cell_Name>`, in the library, `<Lib_Name>`.

  For example, the following definition will assign `15.0` as power value for every pin of the cell `nand2` in the library `ASIC_LIB:`

  ```
  gate_power=ASIC_LIB:nand2:*:15.0
  ```


© 2021 All Rights Reserved.

In the two formats discussed above, you have to specify `<Lib_Name>`. Therefore, you should know the `<Lib_Name>` in your compilation script, for example the library name specified in the `refLib` command in the compilation script. `<Lib_Name>` is required to resolve the ambiguity when a `<Cell_Name>` appears in multiple libraries. If a cell does not have a duplicate in other libraries, you can use an asterisk (`*`) as `<Lib_Name>` to avoid specifying a real name, as follows:

- `gate_power=*:<Cell_Name>:<Pin_Name>:<PowerValue>`

  This means, assign the `<PowerValue>` as the power value for the pin, `<Pin_Name>`, of the cell, `<Cell_Name>`.

  An error is reported in either of the following conditions:

  - The specified pin does not exist in the specified cell.
  - The power value is zero or negative.
  - The specified cell is not a simple gate cell.
  - The specified cell exists in multiple libraries. In this case, you need to specify the `<Lib_Name>`.

- `gate_power=*:<Cell_Name>:*:<PowerValue>`

  This means, assign the `<PowerValue>` as the power value for ***all*** pins of the cell, `<Cell_Name>`.

  An error is reported in either of the following conditions:

  - The power value is zero or negative.
  - The specified cell is a blackbox cell defined in an imported power table file.
  - The specified cell exists in multiple libraries. In this case, you need to specify the `<Lib_Name>`.

Note: If you define multiple `gate_power` keywords in the `powerdb.index` file, the definitions that specify the `<Pin_Name>` along with the `<Cell_Name>` are given the highest priority as compared to those that specify only the `<Cell_Name>`. For example, when you define the following statements, the pin `Y` of `nand2` will have power value `15.0`, and other pins in `nand2` will have power value `10.0`:

```
gate_power=ASIC_LIB:nand2:Y:15.0
gate_power=ASIC_LIB:nand2:*:10.0
```


© 2021 All Rights Reserved.

Adding power_file Keyword to the powerdb.index File

The `power_file` keyword can import a text-format file that includes power values of blackbox cells and simple gate cells for WTC analysis.

- `power_file=<Power_File_Name>`

  Here, `<Power_File_Name>` is a text-format file.

  For the content of the `<Power_File_Name>`, you can refer to the `powerdb.dump` file in the `dbFiles` directory. This is a text-format file that includes the entire power data collected in the internal database after running `dbgPrepare` using the following command:

  ```
  dbgPrepare <lib_name> <proto_name> -checkpowerdb
  ```

  Copy the contents of the `powerdb.dump` file to a new file, modify the content, and then, import the new file using the `power_file` keyword in the `powerdb.index` file. Run the `dbgPrepare <lib_name> <proto_name>` command to overwrite the old values in the internal power database with the power data from the imported file.

  You can also write your own power file. However, make sure that the following mandatory content is added to your power file in the same order:

  ```
  cell name : RAM_4096x32
  QTDB library name : MACRO_LIB
  simple gate : false
  arc # : A[0]    A[0]    10.000000
  arc # :  CLK   CLK   4430.000000 (!PGEN & RETN) & (!CEN & ! (WEN) ) & (!EMA[2])
  & (!EMA[1]) & (!EMA[0])
  ..
  ```

  Note: The keywords in bold are case-sensitive. Use `arc` to describe a power characteristics for a pin, which is equivalent to a module port. The specified pin should exist in the port list of a module definition, but should not be a wire or register.

  For a simple gate cell, one `arc` has two elements: the *pin* name and *power(nW)*, which is the power for the specified *pin*.

  For example:

  ```
  cell name : cell buf1
  ...
  ...
  arc(s) : pin    power(nW)
  arc #0 :  Y     5.033000
  ```

  For a blackbox cell, one `arc` can have four elements: *pin(from)*, *pin(to)*, *power(nW)*, and *when*. However, *when* is optional, and the other three are mandatory.

  For example:

  ```
  cell name : RAM_4096x32
  ...
  ```


© 2021 All Rights Reserved.

```
...
arc(s) : pin(from)   pin(to)    power(nW)    when
arc #13 :  CLK          CLK      4430.000000    (!PGEN & RETN) & (!CEN &
! (WEN) ) & (!EMA[2]) & (!EMA[1]) & (!EMA[0])
```

An error is reported in either of the following conditions:

- The specified cell does not exist in the specified library.
- The specified library is not imported by the design.
- The specified pin does not exist in the specified cell.
- The power value is zero or negative.
- Cell type (described by simple gate) is inconsistent with the definition in the imported power tables.

Important Considerations When Using Keywords in powerdb.index File

Keep the following considerations in mind when using either `gate_power` or `power_file` keyword in the `powerdb.index` file:

- After compiling the design, if you change the definitions of any of these keywords in the `powerdb.index` file, you need to run the `dbgPrepare` command to bring the changes into affect. Following is the syntax of the `dbgPrepare` command:

  ```
  dbgPrepare <lib_name> <proto_name>
  ```

  To find the correct values for *<lib_name>* and *<proto_name>*, access the `PDB` subdirectory in the design directory. The `PDB` subdirectory contains files with names of the format *<lib_name>.<proto_name>.<file_extension>*. For example, in the file named `LIB.testbench.dadb`, `LIB` is the *<lib_name>* and `testbench` is the *<proto_name>*. So, you should run the following command:

  ```
  dbgPrepare LIB testbench
  ```

  The `dbgPrepare` command re-computes the weights for all nets and saves them into a power profile database that can be used for WTC analysis later.

- When you use either the `gate_power` or `power_file` keyword for a cell, if the cell is directly included in an imported power table file, its cell type (simple gate or blackbox gate) should not be changed in the definition of `gate_power` or `power_file`. If the cell is not directly included in any imported power table, you must choose the correct cell type based on the design; otherwise, the WTC data will be incorrect.

- In most cases, only NTC analysis is applied to RTL designs because Liberty format files might not be available, and so, power table files cannot be used. However, you can apply WTC analysis for RTL designs as well by adding either the `gate_power` or `power_file` keyword to the `powerdb.index` file. Following are the necessary steps:

© 2021 All Rights Reserved.

a. Add the following compilerOption command to your compilation script:

```
compilerOption -add {powerdb <powerdb_directory>}
```

The *<powerdb_directory>* parameter in the above syntax is the power profile database directory name specified with either the full path or path relative to the working directory. This power profile database directory will contain the powerdb.index file.

b. Create the powerdb.index file in the power profile database directory specified using the *<powerdb_directory>* parameter in the above step.

Add the gate_power or power_file keyword definitions to this powerdb.index file.

c. Compile the design.

d. At run time, use the following dpa command to collect the WTC data:

```
dpa -getToggleCount -file <ppfdata_file> -weighted
```

Applying the Power Database

To define your power database to the compiler, use the compilerOption command to specify the power database:

```
compilerOption -add {powerdb <powerdb_directory>}
```

The directory name can be either the full path or relative to the working directory. This directory must contain all power reports and powerdb.index or powerdb.tcl files.

If both powerdb.index and powerdb.tcl files are present in the powerdb directory, the powerdb.index file is ignored with a warning like the following:

```
WARNING: powerdb.tcl file is detected, then powerdb.index will be ignored.
```

Along with specifying this compilerOption parameter, you can include standard compilation options, and specify the precompile and compile options as normal.

With a gate-level netlist and the power database, you can collect and view weighted toggle counts to get a more accurate view of when the design is likely to be using high amounts of power. You can then generate TCF or SAIF files for these times.

The TCF files are input to Joules along with your gate-level netlist and descriptions of Liberty format cells to determine actual power consumption for those periods. The SAIF files are input for third-party tools used for power analysis. For details related to SAIF files, see Generating Backward SAIF Files for Power Analysis by Third-Party Tools on page 1448.

© 2021 All Rights Reserved.

Weighted Toggle Count for RTL Designs

The system supports Weighted Toggle Count (WTC) analysis for the RTL designs. The generated gate level netlist is based on the qtref cells.

Note: Before using WTC analysis for RTL designs, you must install the Joules software, which is a Cadence product for power estimation. Joules is an RTL power analysis product that provides a unified platform to RTL designers. You also need to specify the liberty (`*.lib`) files for the ASIC cells. For complete information on how to use Joules for power analysis, refer to the *Joules User Guide*. Ensure that you have the Joules 16.21 or later release installed.

To perform WTC analysis for the RTL designs, you need to generate the qtref power table using the `create_qtref_power_table` command. The qtref cells are mapped to the ASIC cells in the liberty file to assign weights for RTL WTC analysis.

For example, in the qtref power table, a qtref cell `Q_XOR2` exists, which is 2 input XOR gate. The system finds the corresponding 2 input XOR cell in the `*.lib` file. If a match is found, the power information from that cell is assigned to the `Q_XOR2` cell.

The qtref cells are mapped to the ASIC cells in the following situations:

- If a qtref cells match one or multiple ASIC cells in the liberty file, the average of the ASIC cells' power value is calculated and then assigned to the qtref cell.
- If no match is found for the qtref cell, the group average value is derived from similar qtref cells. If the group average is not available, a global average power value is computed from all matched qtref cells. The average power value is then assigned to the qtref cell. For complex qtref cells such as `Q_AD04` or `Q_MX08`, the system computes a generic power values based on the similar qtref cells that already have power values.

Creating a qtref Power Table

Execute the `create_qtref_power_table` command as follows to generate the qtref power table from the `*.lib` file.

```
create_qtref_power_table <user ASCI Liberty file> <power table file name>
```

For example:

```
create_qtref_power_table typical.lib qtref.rpt
```

The `create_qtref_power_table` command reads the power liberty file (`typical.lib`) and creates an equivalent power profile using the qtref based cells. Then, you can add the new power profile into the `powerdb.index` file.

© 2021 All Rights Reserved.

Adding qtref Power Table into powerdb.index File

Execute the following statement to add the qtref power table into the `powerdb.index` file:

```
qtref=<qtref_power_table_ file_name>
```

The `qtref=<qtref_power_table_ file_name>` statement must be placed prior to the other power table statements in the `powerdb.index` file.

For example,

```
# powerdb.index
qtref=qtref.rpt //import qtref.rpt
black_box=xxx
```

Note: The `gate` statement is not required for the netlist. However, the `black_block` statement is required for memory cells.

Then, execute the `create_power_table` command to create the power database for memory cells. If you do not provide the power table for memory cells, the system assigns a default value as the weight of the pins.

After the power database has been prepared, use the `compilerOption -add {powerdb <powerdb_directory>}` command in your compile script to define the power database to the xeCompile compiler. For details, refer to the Applying the Power Database section.

Verifying the Cell Power Values

Refer to the `powerdb.dump` file in the `dbFiles` directory to verify the power value of cells. The `powerdb.dump` file also lists the computed power value for the qtref cells that do not have direct power information from the liberty file.

For example, the following qtref cells do not have direct power information from the liberty file. The value of these cells is computed using other similar type of cells.

```
QT_cell_Name     Num_of_instance     Percentage
============================================================
Q_FDP0I0         32                  0.3137%
Total simple gates: 10282
```

The messages are also printed in the `xe.msg` file for the qtref cells whose power weights are assigned an average value from other similar type of qtref cells.

Modifying the Cell Power Values

You can modify the cell power values in the `powerdb.index` file by using `power_file` or `gate_power` keywords.

© 2021 All Rights Reserved.

- To enable the WTC RTL analysis, use the following statement:

  ```
  qtref=<qtref_power_table_ file_name>
  ```

- To import a text-format file that includes power values of cells for WTC analysis, specify the `power_file` statement as follows:

  ```
  power_file=<Power_File_Name>
  ```

  For information about the format of the `power_file` keyword, refer to Adding power_file Keyword to the powerdb.index File section.

- To define the power values of the qtref gate cells for WTC analysis, specify the `gate_power` statement as follows:

  ```
  gate_power=<qtref_library_name>:<qtref_cell_name>:<pin_name>:<arc value>
  ```

  The above statement assigns the `<arc value>` as the power value for the pin, `<pin_name>`, of the qtref cell, `<qtref_cell_name>`, in the library, `<qtref_library_name>`. Use `*` to specify all pins on the cell.

  For example, to assign power value `18.0` to the pin `X` of the cell `nand1` in the library `QTREF_LIB`, define the following in `powerdb.index` file:

  ```
  gate_power=QTREF_LIB:nand1:X:18.0
  ```

  The default power values will be replaced by the specified values. An error is reported in either of the following conditions:

  - The specified pin does not exist in the specified cell.
  - The specified qtref cell does not exist in the specified qtref library.
  - The specified cell is a blackbox cell defined in an imported power table file.

Note: You should not use the `qtref=xxx statement` for the simple gate level WTC analysis.

Example - powerdb.index

```
qtref=qtref.rpt //Loads the power table for qtref cells
power_file =./powerdb/powerdb.dump // Loads powerdb.dump that defines power values.
gate_power=qtref:Q_LDP0:*:1000.00
gate_power=qtref:Q_FDP1:*:1000.00
```

Using Multiple Power Tables in WTC Analysis

You can use multiple power tables in WTC analysis. Using multiple power tables is beneficial in situations when different instances in a design contain same type of cells and operate at different power voltages.

© 2021 All Rights Reserved.

For example, Inst A, Inst B and Inst C all have the cell `nand2`, but operate at different voltages such as 1.0v, 0.9v, and 0.85v respectively. In such situations, you can assign different power tables to different instances.

The `powerdb.index` file in WTC analysis is now enhanced to add keywords to specify the instances to which the corresponding power table file applies. You can create different power tables for RTL designs as well.

Following are the mandatory requirements to use multiple power tables in WTC analysis:

- You need to provide liberty format files (`*.lib` files) of ASIC cells for the required instances.
- These liberty files should contain same type of ASIC cells with different power values.
- One of these liberty files should apply on the top instance globally.

To use multiple power tables in WTC analysis, do the following:

1. Create a power table from each of the given `*.lib` files.
2. Modify the `powerdb.index` file to add the `pwrinst` keyword to specify the instances for various power tables.
3. Modify the compile script for xeCompile to specify the power database.
4. Refer to the `powerdb.dump` to check the cell power values.

*Creating Power Table from *.lib Files*

Create a power table from liberty files using the following command:

```
create_power_table <ASIC Liberty file> <power table file name>
```

For example:

```
create_power_table asic_power_Global.lib asic_power_Global.rpt
create_power_table asic_power_instA.lib asic_power_instA.rpt
```

To create power tables for RTL designs, use `qtref_power_table`, as follows:

```
create_qtref_power_table ../lib/PD_1.10V.lib ./powerdb/PD_1.10V.rpt
create_qtref_power_table ../lib/PD_1.15V.lib ./powerdb/PD_1.15V.rpt
```

For details about creating power tables for WTC analysis, refer to Creating a Power Profile Database section.

© 2021 All Rights Reserved.

Modifying powerdb.index

After the power table is generated, include the power table in the `powerdb.index` file with a new keyword, `pwrinst`, in the required format.

Example,

```
# powerdb.index
```

1. The following power table statement should be provided prior to other power table statements, if any, to apply on top instance of the design with default power value:

   ```
   gates=asic_power_Global.rpt
   ```

2. Use the new `pwrinst` keyword followed by a full path instance name, to specify the instance that the next power table statement will apply to:

   ```
   pwrinst=top_inst.instanceA
   ```

   The following power table statement applies to `top_inst.instanceA`

   ```
   gates=asic_power_instA.rpt
   ```

3. Specify more instances, if required. Each `pwrinst` specified instance should have its corresponding power table in the next line, as follows:

   ```
   pwrinst=top_inst.instanceB
   gates=asic_power_instB.rpt
   ....
   black_box=xxx
   ```

The usage of black box cells in multiple power tables is same as in usual WTC analysis. For details about the usage of black box cells, refer to Creating a Power Database File (powerdb.index) section.

Modifying the Compile Script

After editing the `powerdb.index` file, modify the compile script of xeCompile by adding the `compilerOption -add {powerdb <powerdb_directory>}` command.

For details, refer to Applying the Power Database section. This section discusses the power database directory name and directory contents, and the operations that can be performed using the power database.

Verifying the Cell Power Values

Refer to the `powerdb.dump` file in the `dbFiles` directory to verify the power value of cells. The `powerdb.dump` file contains multiple sets of power values for different instances.

For example,

© 2021 All Rights Reserved.

```
Instance: global
========================================
cell #0 :
cell name : nand2
library name : ./powerdb/asic_power_Global.rpt
QTDB library name : ASIC_LIB
simple gate : true
----------------------------------------
arc(s) : pin power(nW)
arc #0 : Y 4000.000
----------------------------------------
pin(s) :
pin #0 : A
pin #1 : B
pin #2 : Y
..
...
Instance: top_inst.instanceA
========================================
cell #0 :
cell name : nand2
library name : ./powerdb/asic_power_instA.rpt
QTDB library name : ASIC_LIB
simple gate : true
----------------------------------------
arc(s) : pin power(nW)
arc #0 : Y 5000.000
----------------------------------------
pin(s) :
pin #0 : A
pin #1 : B
pin #2 : Y
..
...
```

Using Tcl Scripts to Specify Power Tables for WTC Analysis

You can specify the power tables for WTC analysis using the Tcl scripts. The existing functionality of various `powerdb.index` file commands has been extended to several new TCL format commands, which you can specify in the `powerdb.tcl` script. The `powerdb.tcl` script is provided under the `powerdb` directory. The `powerdb` directory is located in the design path, which is declared by the `compilerOption -add {powerdb ./powerdb}` command.

You can edit the `powerdb.tcl` script to specify the power tables for WTC analysis. The commands in the script offer the same functionality as the commands in the `powerdb.index` file. However, with the Tcl format, you can leverage additional features for WTC analysis, such as, variable substitution, if statement, for statement, file name pattern, and so on. For example, using the environment variable to specify path name for power table, or using the file name pattern, such as “`a*.rpt` while specifying the power table.

© 2021 All Rights Reserved.

This section explains how to use the `powerdb.tcl` script for WTC analysis. If the required settings are done in the `powerdb.tcl` file, this file will be automatically imported instead of the `powerdb.index` file. If, however, the `powerdb.tcl` is not provided, the `powerdb.index` file will be imported by default.

Use the following Tcl commands with the appropriate options in `powerdb.tcl` to import any power information, such as power tables.

The Tcl commands to define the power tables in the `powerdb.tcl` file are as follows:

- `import_power_table -gates <rpt_file_name> [-pwrinst <inst_name>]`
- `import_power_table -black_box <rpt_file_name>`
- `import_power_file <power_file_name>`
- `import_gate_power LIB_name:cell_name:pin_name:weight_value`
- `import_power_table -qtref power_table_name.rpt`

Examples

The following examples show the sample content for `powerDB.tcl` script.

```
import_power_table -gates tutorial.rpt
import_power_table -black_box RAM_4096x32_ss_1v08_125c_syn.rpt
set myrptfile tutorial.rpt
set cell flopdrs
import_power_table -gates $myrptfile
import_power_table -black_box ./RAM_4096x32_ss_1v08_125c_syn.rpt
import_gate_power *:nor2:*:5.0
set f [open ./src/dut.vg]
while {[gets $f line] != -1} {
             if {[regexp "$cell" $line]} {
                   import_gate_power DUT_LIB:$cell:D:2.0
                }
      }
```

The following table lists various commands in the `powerdb.tcl` file and their equivalent commands in the `powerdb.index` file.

Table 32-1 Power Commands in powerdb.index and powerdb.tcl

Commands in powerdb.index	Commands in powerdb.tcl
gates=<*power_table_file_for_simple_cells*>	import_power_table -gates <*rpt_file_name*> [-pwrinst <inst_name>]

© 2021 All Rights Reserved.

Table 32-1 Power Commands in powerdb.index and powerdb.tcl

Commands in powerdb.index	Commands in powerdb.tcl
black_box=<*power_table_file_for_complex_cells*>	import_power_table -black_box <*rpt_file_name*>
gate_power=<*Lib_Name*>:<*Cell_Name*>:<*Pin_Name*>:<*Power Value*>	import_gate_power LIB_name:cell_name:pin_name:weight_value
power_file=<*Power_File_Name*>	import_power_file <*power_file_name*>
qtref=<*qtref_power_table_file_name*>	import_power_table -qtref power_table_name.rpt

import_power_table -gates <rpt_file_name> [-pwrinst <inst_name>]

This command treats the cells in the specified `*.rpt` file as simple gates. The file name could be a relative path name or an absolute path name. This command is equivalent to the `gates` keyword in the `powerdb.index` file.

Specify the `-pwrinst` option only while using the multiple power tables for WTC analysis. The `-pwrinst` option is followed by an instance name, such as `Top_inst.intanceA`, to specify the instance for the power table. Without the `-pwrinst` option, power table is applied globally to all the instances.

For example,

```
set myrptfile tutorial.rpt
set instance dut.cnt_u1
import_power_table -gates $myrptfile -pwrinst dut.cnt_u0
import_power_table -gates tutorial_reverse.rpt -pwrinst $instance
```

import_power_table -black_box <rpt_file_name>

This command treats the cells in the specified `*.rpt` file as black box cells. The file name could be a relative path name or an absolute path name. This command is equivalent to the `black_box` keyword in the `powerdb.index` file.

For example,

```
import_power_table -black_box $env(MY_PATH)/RAM_4096x32_ss_1v08_125c_syn.rpt
```


© 2021 All Rights Reserved.

import_power_file <power_file_name>

This command imports a power file saved under the `powerdb` directory to change values of the cells. The power file should be in the text format. The format of the power file is same as that of the power file for `powerdb.index`. This command is equivalent to the `power_file` keyword in `powerdb.index`.

For example,

```
set myfile ../powerfile
import_power_file $myfile
```

import_gate_power LIB_name:cell_name:pin_name:weight_value

This command is followed by a gate power string to import gate power information and is equivalent to the `gate_power` keyword in `powerdb.index`. Specifying gate power changes the power values of a specific pin of a cell. The format of the gate power string should be same as that of the gate power string in the `powerdb.index` file.

For example,

```
set cell flopdrs
import_gate_power *:nor2:*:5.0
set f [open ./src/dut.vg]
while {[gets $f line] != -1} {
               if {[regexp "$cell" $line]} {
                    import_gate_power DUT_LIB:$cell:D:2.0
                   }
          }
```

In the above example, cell value of the `nor2` cell is being modified using wildcards and the value of the `flopdrs` cell is being modified in a while loop.

import_power_table -qtref <power_table_name.rpt>

The `qtref` option imports the `qtref` power table generated by the `create_qtref_power_table` command. The `import_power_table -qtref` <*`power_table_name.rpt`*>command is equivalent to the `qtref` keyword in `powerdb.index`.

Examples

You can use any Tcl feature to simplify your job such as using variables, loop and file name pattern, and if statements:

- Using variable names:

© 2021 All Rights Reserved.

- Using Tcl variable `myrptfile` to hold the `*.rpt` file name:

  ```
  Import_power_table -gates $myrptfile
  ```

- Using the `MY_PATH` environment variable to specify the path to the `*.rpt` files.

  ```
  import_power_table -gates $env(MY_PATH)/asic_1.rpt
  ```

- Using loop and file name pattern:

  ```
  foreach filename [glob "mylib*.rpt"] {import_power_table -gates $filename}
  ```

  The above command will define `mylib*.rpt` files as simple gates.

- Using if statement:

  ```
  if { $use_new_lib }
      { import_power_table -gates new.rpt }
  else
      { import_power_table -gates old.rpt }
  ```

Modifying Compile Script for xeCompile

After you have successfully created the power database, you can use it by adding the `compilerOption -add {powerdb ./powerdb}` command in your compile script for xeCompile. After compilation, refer to the `dbFiles/powerdb.dump` file to check the cell power values.

Refer to Applying the Power Database section for details on the power database directory and its contents, and the operations that can be performed using the power database. Refer to Weighted Toggle Count (WTC) Analysis for information on the steps required to perform the WTC analysis and for details on `powerdb.index` file.

Enabling Fast Toggle Count (FTC)

Normal Toggle Count (NTC) enables you to generate a toggle count profile for the selected time window and identify peak in the toggle count profile. The peak indicates the highest volume of power consumption. You can then focus on a smaller time window around the peak in toggle count profile to do further analysis to decide whether the peak corresponds to the peak in power consumption.

NTC is based on fullvision, which need to upload large volume of data from the emulator to do fullvision computation. Therefore, generating the NTC profile takes time. The system enables you to add instrumentation logic to compute the toggle count in both SA and ICE modes, such that the toggle count is available immediately in each clock cycle. This method is known as **Fast Toggle Count (FTC)**.

The toggle count is represented by an N-bit register that stands for a N-bit unsigned integer. You can use this register in SDL trigger or probe it to get a toggle count profile.

© 2021 All Rights Reserved.

To accelerate the toggle count profile, you need to compile the design with compilerOption -add {FTCTopNInst <*N*>} or compilerOption -add {SubFTCTopNInst <*filename*>} commands to insert the instrumental logic to compute the toggle count. These commands are supported in both 1X and 2X modes. For details, refer to Specifying Instances to Generate Toggle Count Profile section.

In ICE mode, the toggle count is based on fclk cycle. The signal values for the current fclk cycle are compared with the previous fclk cycle.

In SA mode, the toggle count is based on the simulation time bin. The signal values for the current time bin are compared with the previous time bin. In SA mode, each time bin consists of multiple fclk cycles. Only the last fclk represents the stable value for each time bin. All other fclk cycles are ignored.

The toggle count only counts all flip-flops in the selected instances. All nets are not counted. This behavior reduces the overhead from the instrumental logic.

Specifying Instances to Generate Toggle Count Profile

The compiler pre-selects a set of instances and creates one toggle count register for each instance. The set of instances are automatically selected by the compiler according to the size of the instances. For example, a design has the following hierarchy:

```
Top (5,000)
   Top.A (4,000)
      Top.A.B (2,000)
      Top.A.C (1,900)
   Top.D (100)
```

The number in the parenthesis is the number of nets in each instance. If the compiler selects top three largest instances, the instances would be Top(5,000), Top.A(4,000), and Top.A.B (2,000). Note that the size of Top.A is not equal the size of Top.A.B plus size of Top.A.C because there are some other gates inside Top.A that belong to neither Top.A.B nor Top.A.C.

You can use the following compilerOption commands to pre-select different number of instances and at different design hierarchies. For example, you might pre-select top 50 instances from Top.GPU.Core1.

- compilerOption -add {FTCTopNInst <*N*>}: Specifies the number of instances to select starting from the top level to generate the toggle count.
- compilerOption -add {subFTCTopNInst <*filename*>}: Specifies the instances to select starting from any hierarchical level to generate the toggle count.

© 2021 All Rights Reserved.

The compiler prints a list of pre-selected instances to let you know about the instances that provide the toggle count. The size of the pre-selected instances is also printed.

For example,

```
List of toggle counter modules:
LIB:testbench (123)
LIB:dut (110)
LIB:mem_wrapper (78)
MACRO_LIB:RAM_4096x32 (78)
End of toggle counter modules.
```

Note: The capacity overhead of the instrumental logic is linear to the number of signals that need to be counted. If FTC is enabled on the whole design, the capacity overhead is about 2X. Inserting instrumentation logic creates several levels of combinatorial logics that might result in larger step count — that is, slower emulation speed.

Accessing the Toggle Count

The compiler creates an N-bit register for each pre-selected instances. The size of the N-bit register depends on the number of signals that need to be counted — that is, the size of the register is pre-defined according to the number of signals. For example, if an instance contains 5 millions of signals, then the toggle count register needs 23 bits.

The compiler creates a dedicated name for each toggle count register. The format is: instance name followed by `HSV_ToggleCounter`.

For example, to access the toggle count for `Top.A`, you can use `Top.A.HSV_ToggleCounter`. You can use the `value` command to query the toggle count value. You can also use the same instance name in SDL programs to create triggers. You can also probe the toggle count register to get a waveform. The register is shown as a bus in Simvision and Verdi.

Accelerating WTC Profile Using Hardware-Assisted Weighted Toggle Count (HW-WTC)

Weighted Toggle Count (WTC) is based on fullvision, which needs to upload huge amount of data from emulator for FV computation. When the simulation time is over billion cycles, the generation of the toggle profile for a long simulation is infeasible in terms of time and computing resources. In such cases, you can accelerate the generation of the WTC profile using **Hardware-Assisted Weighted Toggle Count** (**HW-WTC**), and get the WTC data in a faster way.

Note: HW-WTC is supported only for gate-level netlists (with ASIC liberty file). HW-WTC is supported in both ICE and SA modes, and supported in 1X as well as 2X mode.

© 2021 All Rights Reserved.

HW-WTC does not support RTL-level netlists with QTref libery file. For RTL-level netlists, Cadence recommends that you should use the Fast Toggle Count (FTC). For information on how to use FTC, refer to the Enabling Fast Toggle Count (FTC) section.

This section discusses how to perform HW-WTC analysis. It covers the following sections:

- Enabling Hardware-Assisted WTC (HW-WTC)
- Defining Top-N instances
- Accessing HW-WTC Registers
- Importing Power Tables for Compiler
- Generating ppfdata in Offline Sessions
- Observing ppfdata
- Generating Window of Interest
- Restrictions and Limitations

Enabling Hardware-Assisted WTC (HW-WTC)

To enable HW-WTC, modify your compile script to use the `compilerOption -add {hwWTC on}` command. The compiler internally generates instrumentation logic to compute the Weighted Toggle Count for each top N instance. The weight is based on Software-WTC (SW-WTC). If you need to compile the design without HW-WTC, remove the `hwWTC` compiler option or set it to `off`.

Note: You must clean up the *QTDB*, *PDB* and *dbFiles* directory to remove the instrumental logic generated for HW-WTC in the previous compile.

You can access the HW-WTC data by a 40-bit register for each top N instance. Use SDL or DRTL to display the HW-WTC registers, or generate the `.ppfdata` file. The compiler creates a dedicated name for each toggle count register. The format is: instance name followed by `HSV_ToggleCounter`. For example: `dut.HSV_ToggleCounter[39:0]`.

For more details about the `compilerOption -add {hwWTC on}` command, refer to the description of this command in the *User Data Commands* chapter of the *VXE Command Reference Manual*.

Defining Top-N instances

Use the following `compilerOption` commands in your compile script to define the TopN instance set:

© 2021 All Rights Reserved.

- compilerOption -add {DPATopNInst <*N*>}: Adds top N pre-selected biggest instances of the whole design. The instances are sorted based on the number of nets in each instance. The default value of N is 10000 for SW-WTC and 50 for HW-WTC.
- compilerOption -add {subDPATopNInst <*fileName*>}: Adds top N instances under a specified instance, which might not be the root instance, into the pre-selected instances list.

For details on these command options, refer to the description of the compilerOption -add {DPATopNInst <*N*>} and compilerOption -add {subDPATopNInst <*fileName*>} command in the *User Data Commands* chapter of the *VXECommand Reference Manual*.

Note: HW-WTC supports maximum 100 TopN instances due to capacity overhead. Defining more than 100 instances issues the following warning:

```
HW-WTC only supports TopN <=100 for DPA, use 100 as the maximum TopN
```

Accessing HW-WTC Registers

Each topN instance has a corresponding HW_WTC instance, and the HW-WTC data is kept in the output register HSV_ToggleCounter[39:0] of the HW_WTC instance. For example, WTC data for instance dut is maintained in dut.HSV_ToggleCounter[39:0]. The HW-WTC data is used to probe to dump waveform, or generate the .ppfdata file. It can also be used in SDL and DRTL programs.

The generated HW-WTC instances are saved at ./dbFiles/hwtc_counters.lst file. The content of the hwtc_counters.lst file looks similar to the following:

```
testbench
testbench.dut
testbench.dut.cnt_u0
testbench.dut.cnt_u1
testbench.dut.cnt_u2
testbench.dut.cnt_u3
testbench.dut.mem_u
testbench.dut.mem_u.mem_core
```

Importing Power Tables for Compiler

The compiler reads the power table files and assigns the net weights.

1. Import the power table file by editing the powerdb.tcl file under the powerdb directory.

   Following is an example of the import_power_table command in the powerdb.tcl file to support the gates and black_box power table files:

© 2021 All Rights Reserved.

```
#powerdb.tcl
import_power_table -gates <rpt_file_name>
import_power_table -black_box <rpt_file_name>
```

2. Set the `compilerOption -add {powerdb ./powerdb}` in your compile script. Also place the power table files and the `powerdb.tcl` file under the `powerdb` directory.

To view the list of instances list that are instrumented with the toggle counter, refer to the `./dbFiles/hwtc_counters.lst` file. The content of the `hwtc_counters.lst` file looks similar to the following:

```
dut
dut.instanceA
dut.instanceB
dut.instanceC
…
```

For more details on how to import the power tables, refer to Using Tcl Scripts to Specify Power Tables for WTC Analysis section.

Generating ppfdata in Offline Sessions

To generate the ppfdata file for HW-WTC counters, follow the steps given below:

1. Open the `*.phy` in an offline session:

   For example,

   ```
   debug .
   host -offline trace.phy
   …
   ```

2. Execute the `dpa -setupToggleCount` with the `-hw` option to collect the specified instances with toggle counters, as follows:

   ```
   dpa -setupToggleCount -addinst -hw <instance1_name>
   <instance2_name>...<instanceN__name>
   ```

   Note: The instance names are the same as in your design. To view the instance names, refer to the `./dbFiles/hwtc_counters.lst` file.

   For example:

   ```
   dpa -setupToggleCount -addinst -hw dut dut.InstA
   ```

   For details on the `-hw` option, refer to the description of the `dpa -setupToggleCount` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

3. Execute the `dpa -getToggleCount` command to generate the ppfdata from the `*.phy` file and save the data in the `<name>.ppfdata` file:

   ```
   dpa -getToggleCount -file <ppf_file_name>
   ```


© 2021 All Rights Reserved.

Note: You can also use the `dpa -getToggleCount -file <ppf_file_name>` command before the `dpa -setupToggleCount` command, which adds all valid toggle counter instances by default in the specified `.ppfdata` file.

To view the collected instance names, use the `dpa -setupToggleCount -listinst` command. These names can be used later in the `tca` command.

Note: The HW- WTC registers do not include the high count value. Therefore, the high counts in the ppfdata are all zero.

Observing ppfdata

Execute the following commands to observe the toggle field in the waveforms for toggle count results:

```
tca -setFile <ppf_file_name>.ppfdata
tca -export <waveform_name> <list_of_instances>
```

For details on various options of the `tca` command, refer to the description of the `tca` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Generating Window of Interest

You can generate a *Window Of interest* (WOI) for real power peaks. WOIs are smaller and multiple time windows within the whole running time window and contains the specified number of power peaks.

Use the `tca -getWOI` command to generate the WOI:

```
tca -getWOI -hwwtc_peak <number>  [-sensitivity <N>]  [-maxNumOfWOI <M>]
[-maxPercentageOfWOI <C>] [-start <start_time>]  [-end <end_time>] inst_name
```

For information on different options of the `tca -getWOI` command, refer to the description of the `tca` command in the *Run-Time Command*s chapters of the *VXE Command Reference Manual*.

Example 1: Generating Window of Interest (WOI)

```
tca -getWOI -hwwtc_peak 100 -maxNumOfWOI 10  dut
```

```
Output:
10 WOIs,  Percentage=73.77%
{dut {[1182 1970] fclk}} {dut {[2206 3026] fclk}} {dut {[3198 4050] fclk}} {dut
{[4222 5074] fclk}} {dut {[5246 6098] fclk}} {dut {[6270 7122] fclk}} {dut
{[7294 8146] fclk}} {dut {[8318 9170] fclk}} {dut {[9374 10194] fclk}} {dut
{[10398 10962] fclk}}
```


© 2021 All Rights Reserved.

Example 2: Generating WOI in Offline Sessions

```
debug .
tca -setFile stream_hw-wtc1.ppfdata
tca -getWOI -hwwtc_peak 100  -maxNumOfWOI 3 dut
....
exit
```

```
3 WOIs,  Percentage=75%
{dut {[1182 1970] fclk}} {dut {[2206 3026] fclk}} {dut {[3198 4050] fclk}}
```

Restrictions and Limitations

- If a design is compiled with the `compilerOption -add` {`hwWTC on`} command, the added instrumental logic cannot be removed unless you recompile the design without this command. You must also clean the `QTDB`, `PDB` and `dbFiles` directory to remove the instrumental logic.
- To resize TopN or change TopN instances, a recompile is required.
- HW-WTC is not supported for designs compiled in modular compilation mode.
- HW-WTC cannot be used if power tables are not imported.
- Fast Toggle Count (FTC) is not supported with HW-WTC.
- HW-WTC counters might have one or half clock cycle delay because the toggles are detected and computed on the positive edge of fast clocking.
- HW-WTC ppfdata can be generated only in the offline mode. The online mode to generate HW-WTC ppfdata using streaming probes is not supported.
- HW-WTC does not support incremental compilation. An error message is displayed when you perform incremental compilation if the design was previously with the `compilerOption -add` {`hwWTC on`} command:

  ```
  ERROR (legacy-53089): The current design database has included HW-WTC
  instrumental logic, failed to buildDB with HW-WTC enabled.
  To proceed, please delete "QTDB" directory and recompile the design.
  ```

- HW-WTC is not supported using the *TCA Preparation* Tab of the DPA GUI.

Using Joules for Power Analysis

Joules is an RTL power analysis product that provides a unified platform to RTL designers. Joules `read_stimulus` command is used to read stimulus for power estimation. It can support the following formats:

- TCF

© 2021 All Rights Reserved.

- VCD
- SAIF
- FSDB
- SDB
- PHY

When the format is PHY, Joules reads stimulus directly from the `*.phy` file, which is generated by Palladium Z1. An xeDebug process will be automatically started to compute the stimulus from the `*.phy` file. The stimulus is then passed to Joules through a socket connection between xeDebug and Joules.

The syntax of this command is:

```
read_stimulus [-file] <stimulus-file-name>  [-format fsdb|vcd|saif|tcf|sdb|phy]
```

For information on the complete syntax and usage, refer to the description of the `read_stimulus` command in *Joules Command Reference* on the Cadence Online Support site.

Examples:

```
read_stimulus -file trace.phy -start 5ns -end 35ns -out 5_35_phy.sdb
read_stimulus -file trace.phy -dut_instance /dut -start 100ps -end 5000ps -out
phy_ps.sdb
read_stimulus -file trace.phy -dut_instance /dut -frame_count 10 -out phy.sdb
read_stimulus -file trace.phy -dut_instance /dut -interval_size 99ns -out phy.sdb
```

For information on how to use the Joules platform for RTL power analysis, refer to the *Joules User Guide* on the Cadence Online Support site.

Generating Backward SAIF Files for Power Analysis by Third-Party Tools

Two types of SAIF files are supported – library forward SAIF and backward SAIF. Palladium Z1 reads the library forward SAIF file containing the state-dependent and path-dependent (SDPD) directives and generates the backward SAIF file containing the SDPD switching activity for the third-party tool used for power analysis.

The process of generating backward SAIF files involves the following steps:

1. Loading the Library Forward SAIF File into Palladium Z1.

© 2021 All Rights Reserved.

2. Specifying Signals in Backward SAIF File.
3. Defining the Scope of Design Hierarchy Search for Instances.
4. Specifying the Time Windows for Backward SAIF File Generation.
5. Generating the Backward SAIF File.

Loading the Library Forward SAIF File into Palladium Z1

Use the following command to load the SDPD directives from the library forward SAIF file into Palladium Z1:

```
dpa -saif -load <saif_file_name> [-module <module_name_pattern>]
```

The library forward SAIF file defines a set of modules. Palladium Z1 runs a search for all instantiations of these modules in the entire design hierarchy and performs calculations on those instances to identify the toggle counts. Therefore, select the modules carefully to avoid any negative impact on the performance.

Inclusion of memory cells and IP cells is recommended. You can also include flip-flops because they are important to make accurate power estimation. However, inclusion of flip-flops can cause additional overhead because the number of flip-flops are usually large.

Including simple gates, such as AND, OR, and XOR, in the library forward SAIF file is not recommended. This is because they have too many instances and do not improve the power estimation accuracy significantly.

Note: The SDPD forward SAIF for backward SAIF generation is not supported with partial `swfvDB` files. For information on how to leverage the partial `swfvDB` files to accelerate the generation of SAIF files, refer to the Accelerating SAIF Generation Using FullVision Partitioning section.

Specifying Signals in Backward SAIF File

Signals inserted in the backward SAIF file include the set of instances selected based on the module names in the loaded library forward SAIF file.

In addition, if required, specify a set of normal signals selected using the following `dpa` commands:

```
dpa -addinstoutputpin <instanceName>
dpa -addinstreg <instanceName>
dpa -input_port <instanceName>
```


© 2021 All Rights Reserved.

You can add all output pins of the cells and hardmacros within the specified instance using the `-addinstoutputpin` option. The `-addinstreg` option adds all register outputs of the specified module instance and its underneath hierarchies to the probe set for TCF/SAIF generation. To add all input ports of the specified module instance and its underneath hierarchies to the probe set for TCF/SAIF generation, use the `-input_port` option.

These normal signals are written as `NET` in the backward SAIF file.

When generating the backward SAIF file, the two types of signals will be combined into the same design hierarchy. If an instance matches a module specified in the library forward SAIF file, then that instance is written to the backward SAIF file. It is possible that some of the internal nets also get selected as normal signals. In such cases, the internal nets will not be written to the backward SAIF file.

For complete details on various options of the `dpa` command, refer to the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Specifying Instances to Generate Module-Based Backward SAIF Files

To generate module-based SAIF files, use the following command:

```
dpa -outfile -saif <saif_file_name> -instance <instanceName>
```

The `<instanceName>` specified in this command must be a subset of the instances selected in the library forward SAIF file. Module-based SAIF files use the names starting from the instance name specified with the `-instance` option.

When the `-instance` option is used, the third-party tool can read from the netlist only that part which corresponds to the specified instance. Also, the generated backward SAIF file works with the partial netlist.

The signal names saved in the SAIF file are trimmed at the beginning to remove the instance name. For example, if you specified `-instance Top.A` option, the signal named `Top.A.n1` will be saved with name `n1` and the `Top.A` part will be removed.

Defining the Scope of Design Hierarchy Search for Instances

The signals are searched through the entire design hierarchy by default. To define a scope to restrict the search operation to only the specified signals, use the following command:

```
dpa -saif -addscope <instance_name>
```

Setting of scope saves the processing time and significantly reduces the size of the backward SAIF file.

© 2021 All Rights Reserved.

Specifying the Time Windows for Backward SAIF File Generation

One backward SAIF file is generated for each time window. So, to generate multiple SAIF files, evenly split the time window into segments using the following command:

```
dpa -outfile -saif <saif_file_name> -segment {<cycleNumber> | <percent>}
```

Segments can be split based on either the number of cycles captured in the backward SAIF file or the percentage of the whole time window. For example, 10% means to split the time window into 10 segments.

You can also specify `-start` and `-end` options with the `dpa -outfile -saif` command to generate multiple SAIF files.

Generating the Backward SAIF File

Use the following command to generate the backward SAIF files:

```
dpa -upload
```

When SAIF files are generated, corresponding SAIF log files are also created. These log files record a summary of toggles and high probabilities, and the start/end time.

Merging Multiple SAIF Files

You can merge the output from multiple SAIF files into a single one, using the `mergeSAIF` run-time command. You can do this in the following ways:

- Specify a list of files to be merged.
- Specify a file pattern to search for, and merge the files matching the specified criteria.

For more information, refer to the description of the `mergeSAIF` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Accelerating SAIF Generation Using FullVision Partitioning

SAIF generation uses multiple fvCompute hosts. In case of large designs, the fvCompute processes on remote hosts consume a lot of memory. Sometimes, you might have difficulty finding hosts with large enough memory in your compute farm.

In such cases, you can leverage the *Parallel Partition Compiler* (PPC) technology to generate multiple software FullVision partitions (`swfvDB`) and then generate a SAIF file for each partition. These partial SAIF files can later be merged into a single SAIF file. Partitioning

© 2021 All Rights Reserved.

the design limits the memory footprint to 256 GB for fvCompute hosts. Though you might also observe some reduction in the xeDebug memory. Using partial `swfvdb` with partial SAIF files is only supported for generating a backward SAIF file without depending on the forward SAIF file.

Note: Multiple `swfvDB` files are not supported for stand-alone fvCompute because saving partial data back to the `*.phy` file is not supported. Multiple `swfvDB` files are generated during compilation and are used at run time. When computing partial data, fvCompute automatically runs in the *Freeze* mode, wherein, the growing size of the `*.phy` file is restricted by not saving the FullVision data to the file.

By default, fvCompute runs on local xeDebug host. To reduce the memory usage on xeDebug, Cadence recommends that you should use the `CDN_FV_NO_LOCAL_HOST 1` environment variable. This variable ensures that the computation tasks for FullVision and DPA are run on the remote hosts. For more details on this variable, refer to Setting Up the Environment for Palladium Z1 chapter.

Follow the given steps to accelerate the generation of SAIF files:

1. Partition your design with the PPC.
2. Enable FV partitioning on PPC enabled designs.
3. Generate the SAIF files partially.
4. Merge the partial SAIF files.

This section explains how to partition the designs using PPC and how to generate the partial `swfvDB` and SAIF files, and covers the following topics:

- Partitioning the Design
- Enabling FV Partitioning for PPC Designs
- Generating *.phy File during Emulation
- Generating SAIF Files in Offline Mode
  - Specifying Partitions to Generate SAIF Files
  - Merging Multiple SAIF Files

Partitioning the Design

Compile the design with PPC to generate the partitions. You can do this using auto-partitioning, by defining your own partitions using the `partitionGroup` user data commands, or by advanced partitioning.

© 2021 All Rights Reserved.

- **Enabling auto-partitioning**: To enable auto-partitioning, use the xeCompile `autoPart` command at compile time.
- **Using the partitionGroup user data commands**:

  `partitionGroup -add {<instance>[.][<prer>][*]<partition>}`

  For example:

  ```
  partitionGroup -add {dut.a.b. part1}
  partitionGroup -add {dut.a.b.r* part2}
  partitionGroup -add {dut.a.c* part3}
  partitionGroup -add {* part3}
  ```

  Note: Manual partitioning does not guarantee memory footprint of less than 256 GB for fvCompute hosts.
- **Using advanced partitioning**: To enable advanced partitioning, use the `compilerOption -add {advancedPart ON | OFF | <N>}` in your compile script.

With PPC compilation, the design is partitioned into various smaller partitions. The typical size for each partition is about three boards on Palladium Z1, which corresponds to about 100 million gates. The compiler identifies all the signals on the boundary of the PPC partitions. Those signals are defined as PI probes. The full `swfvDB` is generated during dbgPrepare Pass1 and is saved in `dbFiles/swfvDB`.

Enabling FV Partitioning for PPC Designs

Enable the FullVision partitioning on PPC designs using the following command in your compile script:

```
compilerOption -add {fvPartition on}
```

Partial `swfvDB` files will be generated if `compilerOption -add {fvPartition on}` is specified. The partial `swfvDB` files are generated using the full `swfvDB` and saved in `dbFiles/swfvDB_[n]`, where n is `0` to <*number of partitions-1*>. Partial `swfvDB_n` will be generated for each PPC partition. Therefore, the number of partial `swfvDB` files is same as the number of PPC partitions.

For details on how to use the PPC, refer to Using Parallel Partition Compiler for Large Designs section. For details on the partitioning commands, refer to the descriptions of the `autoPart` and `partitionGroup` commands in the *User-Data Commands* chapter in the *VXE Command Reference Manual*.

© 2021 All Rights Reserved.

Generating *.phy File during Emulation

After successful creation of the partial swfvDB files, no extra step is required at run time and you can run the design like a regular flow, and generate the *.phy file. The *.phy file is mapped to the full swfvDB — that is, data is captured for all partitions in the design.

Generating SAIF Files in Offline Mode

After the *.phy file is generated in an online session, you can start the offline session to generate the SAIF file from the *.phy file. You can also use your own SAIF generation script. The SAIF generation script has one or multiple dpa -add commands to define probes and the dpa -outfile -saif command to specify the file name for SAIF file, similar to the following:

```
dpa -addinst [-excludelibrary <.llib names>] [-allnets] -depth <n> <instance>
…
dpa -addinstoutputpin <instance> [-gatecell <gatecell_filename>]
dpa -addinstpin <instance>
dpa -addcone -depth <n> <signal> …
dpa -addpath -depth <n> {<net1> <net2>}
dpa -readNetNames <dpa_info_file_name> [<instanceName>]
```

Specifying Partitions to Generate SAIF Files

SAIF files can be generated individually for each partition and session. You can specify the partitions for which the SAIF files should be generated using the -partition option with the debug command:

```
debug . -partition <part_id> -session <session_name>
```

The -partition option specifies the swfvDB partition to process. The <*part_id*> refers to the partition ID, which starts from 0. It corresponds to the postfix of dbFiles/swfvDB_<*n*>. When the -partition option is specified, xeDebug generates the SAIF file for only the specified partition.

You can also specify the session name with the -partition option. For example, debug . -partition -session <*session_name*>. A session will be created for the specified partition and SAIF file generated for the partition will be saved in the session directory.

The dpa -add* command filters out all probes that are not in the current partition.

The dpa -outfile -saif command generates the SAIF files with the following naming convention: <*original_SAIF_file_name*>_<*n*> — that is, the original SAIF file name with a postfix _<*n*>.

© 2021 All Rights Reserved.

Note: You must run xeDebug for all the partitions in the design. You can run these partitions sequentially or run them in parallel. SAIF files will be generated separately for each partition. Cadence recommends you to run these partitions in parallel if you have sufficient resources. You can launch the partitions according to your requirements. All xeDebug sessions are independent of each other.

Merging Multiple SAIF Files

You can create a single SAIF file for the entire design by combining all SAIF files from all session directories and partitions. Use the following command to merge the SAIF files into a single SAIF file.

```
saif_merge  [Options]...
```

For example,

```
saif_merge -o outFile.saif.gz -i inFile_0.saif.gz inFile_1.saif.gz
inFile_2.saif.gz
```

For details on the various options for the `saif_merge` command, refer to its description in the *Programs and Utilities* chapter of the *VXE Command Reference Manual*.

The `saif_merge` command merges the partitions for the same time window. The `-o` option specifies the output file name that has the results for all the merged SAIF files.

You must specify the complete path of the input files. When the partial SAIF files are ready in all the partitions, execute the commands as follows.

```
saif_merge -i partition_*.session/InSaifFilename_*.saif.gz -o outFile.saif.gz
```

If you have used the session IDs, execute the `saif_merge` command as follows:

```
saif_merge -i sessionID*.session/InSaifFilename_*.saif.gz -o outFile.saif.gz
```

Adding [`.gz`] to the end of the output filename compresses the file. If `.gz` extension is not appended, the `saif_merge` command creates a non-zipped merged file. For example, the following command will create a non-zipped merged file, `outFile.saif`.

```
saif_merge -i InSaifFilename_*.saif.gz -o outFile.saif
```

Limitations of Using Partial swfvdb with Partial SAIF Files

Using partial `swfvdb` with partial SAIF files is only supported for backward SAIF generation. This functionality is not supported if you need to include forward SAIF files into the SAIF generation.

© 2021 All Rights Reserved.

Parallel Computing for DPA Using Multiple CPUs

To improve the performance of DPA, the VXE software uses multi-threading in its DPA computation engine to make use of multiple CPUs on a workstation.

The core computation engine uses variable number of CPUs, while the other part of the DPA computation engine uses fixed number of CPUs. The VXE software can use multiple hosts to do computation as described in Running DPA with Multiple Hosts. When using multiple hosts, the xeDebug process on the local host distributes the computation tasks to the fvCompute processes on multiple remote hosts. The core computation engine exists in the xeDebug process on the local host and the fvCompute process on the remote hosts.

The xeDebug process can use up to five more CPUs in addition to the CPUs used by the core computation engine. While the fvCompute process can use up to 1 more CPU in addition to the CPUs used by the core computation engine. The number can change in future releases if the software architecture changes.

The core computation engine can detect the number of CPUs installed on a workstation and decide how many CPUs to use. If a workstation has eight or fewer CPUs, the core computation engine uses all of them. Otherwise, the core computation engine uses only eight CPUs. The number 8 is a preselected number that you can change by setting the following environment variable:

```
setenv CDN_FV_MAX_THREAD <number>
```

Note: In addition to the core computation engine, the VXE software uses multithreading for other features as well, such as computation of FullVision probes.

Running DPA with Multiple Hosts

DPA operations need a huge amount of computation. The computation work can be distributed to multiple hosts to speed up the process. You only need to specify a set of hosts to use. The software will then automatically start the fvCompute process on those hosts. The fvCompute process creates a socket connection to the xeDebug process on local host after start up. Then, the tasks are distributed from xeDebug to fvCompute through this socket connection. xeDebug detects load on the remote hosts and accordingly distributes the tasks. If there are no tasks to compute, the fvCompute process remains idle.

When using multiple hosts to do waveform generation or DPA operations, the software needs to use the file locking system to synchronize access to the `*.phy` file. This requires that the RPC `nlockmgr` service is running on each host. If the `nlockmgr` service is not running on a host, the software displays a warning message as follows and the host is not used.

```
WARNING (legacy-51682): Failed to start fvCompute on host <host_name> because RPC
service nlockmgr is not running (ask system manager to fix it).
```


© 2021 All Rights Reserved.

You might use the `rpcinfo` command to check the services running on a host.

Note: The setting of multiple hosts using the `multi_host_specification_string` will also help the `fvCompute` command and waveform generation.

There are two ways to specify the set of hosts to use. Use either the `rsh` remote execution command or LSF, which is described later.

Requirement on Remote Hosts

The remote hosts must satisfy the following conditions:

- The remote hosts must have VXE software installed on the same path as on the local host.
- The remote hosts must be able to access the design directory by the same path as on the local host.
- As the remote hosts are expected to carry out intensive computation, the CPU speed is critical. Generally, workstations with Intel/AMD CPU are much faster.

Specifying Multiple Hosts Using Commands

Use either of the following commands to specify multiple hosts:

- Specify the following `xeset mtHost` XEL command:

  ```
  xeset mtHost "multi_host_specification_string"
  ```

- Specify the following UNIX environment variable:

  ```
  setenv CDN_MT_HOST "multi_host_specification_string"
  ```

  Note: The `xeset mtHost` command is given preference over the `CDN_MT_HOST` environment variable.

Specifying Multiple Hosts with rsh

To use `rsh`, the `multi_host_specification_string` is a list of host names separated with colon (`:`). Also, make sure that you can do `rsh` to the specified hosts without being asked for a password.

Note: If your IT environment does not allow remote login (rsh) without password, this method cannot be used.

© 2021 All Rights Reserved.

Specifying Multiple Hosts with LSF

You need to define a dedicated queue for DPA purpose. There should be no task per host, and task per user limitation on the queue. The fvCompute processes will not exit after they are started and will remain idle if there is no computation task. If there is a limitation on the queue, other users will not be able to start fvCompute after there are some fvCompute processes running in the queue. The fvCompute process will exit automatically when their parent xeDebug process exits.

To use LSF in DPA on the Palladium Z1 platform, both LSF and xeDebug need to be set up. The configuration of LSF is typically done by the system administrator. Therefore, this document assumes that the basic LSF settings are already in place, and concentrates mainly on the setting up of xeDebug to use LSF.

The `multi_host_specification_string` should follow the following format that consists of three parts separated by a colon punctuation mark:

`LSF:<number_of_hosts>:<bsub_command>`

where,

- The first part of the string is the text `LSF` which is used as the keyword.
- The second part is a numeric value that indicates the number of hosts that will be used for the computation work.
- The third part is a command string for the `bsub` command. It can include any character except for '`:`', which is used as a separator for the LSF string. You can add any options or arguments in this command string. For example, add the `bsub -q my_dpa_que` command string to specify the `bsub` queue name. You can also specify the complete path for the `bsub` command using the `/<LSF_install_dir>/bsub -q my_dpa_queue` command string.

Note: You can try to use any other tool that is similar to LSF by specifying an equivalent command to the `bsub` command. You can try the command-line options specific to the tool.

Setting Limitations on Remote Hosts

When multiple hosts are specified with `rsh` or LSF, a minimum requirement set on remote hosts allows you to identify powerful hosts that will take less time to compute a task. A remote host that does not qualify the minimum requirement is not used.

You can use the following environment variables to set a minimum requirement on remote hosts:

© 2021 All Rights Reserved.

- `CDN_FV_MIN_CPU`: Specify the minimum number of CPUs that a host must have. It must be a number larger than `0`.

  ```
  setenv CDN_FV_MIN_CPU <integer>
  ```

- `CDN_FV_MIN_MEM`: Specify the minimum amount of physical memory that a host must have. It must be a number larger than `0` and represent number of `GB`.

  ```
  setenv CDN_FV_MIN_MEM <integer>
  ```

- `CDN_FV_MIN_FREE_MEM`: Specify the minimum amount of free physical memory that a host must have. It must be a number larger than `0` and represent number of `GB`.

  ```
  setenv CDN_FV_MIN_FREE_MEM <integer>
  ```

- `CDN_FV_MIN_TMP_SPACE`: Specify the minimum amount of free disk space in the `/tmp` directory that a host must have. It must be a number larger than `0` and represent number of `GB`.

  ```
  setenv CDN_FV_MIN_TMP_SPACE <integer>
  ```

These environment variables have no default value. If an environment variable is not set, then the corresponding condition is not checked.

Analyzing Power Consumption Using xeDebug GUI

The xeDebug GUI can be used to collect information during a design run, and analyze the results for the dynamic power consumed by the design. The xeDebug tabs and windows used for DPA are discussed in the following sections.

After completing the design compile, start an xeDebug session and download the design database using the *Setup* tab. If you have a DPA license and the compiled design has power information, the *Power Tower* icon (as shown in Figure 32-4) is enabled on the toolbar. Click this icon to bring up the DPA GUI (as shown in Figure 32-5).

Note: The DPA GUI is not supported while debugging in the regression mode.

Figure 32-4 Toolbar with Power Tower Icon for Dynamic Power Analysis

Note: In addition to needing a separate DPA license to access the *Power Tower* icon, if the license server is not running, or insufficient DPA licenses are available, the *Power Tower* icon will not be enabled.

The DPA GUI that has the following tabs:

© 2021 All Rights Reserved.

- Setup Tab - Enables you to select the `.phy` file on which the DPA operations will be performed.
- TCA Preparation Tab - Enables you to set up the parameters for collecting toggle count data. This is equivalent to using the `dpa -setupToggleCount` command.
- TCA Tab - Enables you to analyze the toggle count data once a `*.ppfdata` file has been generated. This is equivalent to using the `tca` command.
- DPA Probe Tab - Enables you to add DPA probes for generating TCF or SAIF files.
- TCF Tab - Enables you to generate TCF files independently.
- SAIF Tab - Enables you to load forward SAIF files and/or generate backward SAIF files.
- PowerDebug Tab - Enables you to debug power information of a design imported from the `.ppfdata` and TCF files.

Each tab in the DPA GUI has a set of three general buttons that function as *minimize window*, *maximize/unmaximize window*, and *close window*. These additional buttons are provided because some operating systems do not provide these on the top-right of the DPA window.

Setup Tab

When using the xeDebug GUI, Cadence recommends that you invoke an offline session for the DPA operations. An offline session requires selection of a `.phy` file in the *Offline Debugging* group box of the xeDebug *Setup* tab (refer to Figure 42-34 and Figure 42-36 of Appendix 42, “Using the xeDebug GUI”). This `.phy` file is used for waveform generation and DPA operations.

When you open the *Dynamic Power Analysis* window by clicking the *Power Tower* icon from the xeDebug toolbar, the *Setup* tab for DPA operations is displayed (as shown in Figure 32-5). This tab allows you to select a `.phy` file if you do not want to work on the `*.phy` file selected when opening the offline session.

Note: Selecting a `.phy` file on the DPA *Setup* tab is equivalent to using the `dpa -phyfile <phyFileName>` command on the xeDebug command prompt. If no `.phy` file is opened for DPA operations, the remaining tabs in the *Dynamic Power Analysis* window remain disabled.

1. Initially, the non-editable *Current DPA Phy File* field displays the name of the `.phy` file selected on the xeDebug *Setup* tab by default. The value adjacent to this field changes when you select a separate `.phy` file for the DPA operations.

© 2021 All Rights Reserved.

2. Browse to select a separate `.phy` file for the DPA operations by using the directory button adjacent to the *Filename* text field.
3. Below the *Filename* text field, a list of all `.phy` files under the current directory and subdirectories is displayed based on the selection of the *Local Session* or *All Sessions* radio button. The list displays the path where the `.phy` file exists, the date when it was last modified, and intervals of cycle which were captured in it.
4. Select the required `.phy` file from the list and click *Open*. This action will update the filename corresponding to the *Current DPA Phy File* field.

   Note: Alternatively, to open the required `.phy` file, select the `.phy` file and double-click the filename.

Figure 32-5 DPA Setup Tab

Note: You can also perform DPA operations in an online session. In this case, explicitly open a `.phy` file before starting DPA. You can continue to update this `.phy` file with new data

© 2021 All Rights Reserved.

during the online session even after opening it for the DPA operations. However, the new data will not be visible for DPA. For example, you opened a `.phy` file for DPA when it contained data for cycles `0` to `1000`. Then, the `.phy` file is updated with cycles `1000` to `2000`. Despite the update, the DPA operations will continue to see only cycles `0` to `1000`.

TCA Preparation Tab

The *TCA Preparation* tab (as shown in Figure 32-6) enables you to select the instances to be included for analysis.

Figure 32-6 TCA Preparation Tab

Use the *TCA Preparation* tab to do the following:

1. Use the *TCA Probes Add/Remove* group box to add and remove instance names for TCA by using the *Instance* field and *Add*, *Remove*, and *RemoveAll* buttons.

   You can limit the number of instances by using the parameter fields below the *Instance* field, taking the top instances, listing how deep you want to probe, and giving a minimum size in terms of percent of the design.

© 2021 All Rights Reserved.

2. Use the *TCA Probes Display* group box to view the name list of the currently added modules (instances). Module names can be filtered to match a pattern specified in the *Filter* field.

3. Use the *TCA Generation* group box to generate and upload a toggle count file by performing the following steps:

   a. Specify the *Toggle Count File Name*.

   b. Set the *Start Time* and *End Time*.

   When a `.phy` is opened for DPA operations, the *Start Time* and *End Time* fields get initially set to the entire time range of the opened `.phy` file.

   The *End Time* field name is clickable. You can switch this field to *Size* by clicking it.

   These fields are equivalent to using the following command:

   ```
   dpa -tracewindow [{-start <time>} -end <time> | -size <num_cycles>]
   ```

   Note: The button next to the *End Time* or *Size* field can be used to reset the trace window to the entire time range of the opened `.phy` file.

   c. Select the *Append Trace* check box to show trace data captured during the first run and add to it the data for the current run.

   d. Select the *Weighted TC* check box to collect weighted toggle counts. Use this only if you compiled using a power database; otherwise, normal toggle counts are collected.

   e. Click *Upload ToggleCount*. The toggle count data will be collected and placed in a file with the suffix `.ppfdata`.

   Note: Multiple hosts can be used to speed up the collection of the toggle count data by setting the `mtHost` run-time parameter or the `CDN_MT_HOST` environment variable.

TCA Tab

The *TCA* tab is used for examining the toggle count data.

This tab allows you to open multiple `.ppfdata` files in the same session. This feature is useful for analyzing multiple `.ppfdata` files generated for the same design to compare the power consumption of the same module at different run times.

In addition, the local graphing feature adds support to plot both the high count data and toggle count data for the same instance at the same time.

© 2021 All Rights Reserved.

The *TCA* tab allows the data analysis for the current working files and provides a quick way to identify instances and times of interest before exporting the data to waveform. By default, ***all*** currently opened `.ppfdata` files are treated as working files. To close an opened `.ppfdata` file, in the Import TCA Data group box, specify the name of the file in the *TCA File Name* field and then click **X** (cross) icon button next to the text field. Otherwise, change the selection to any one of the opened files by selecting the name of the specific file using the down-arrow button next to the *File* field, which is the first field in the left pane. Consequently, the subsequent operations will only use that selected file.

The following sections describe the use of each group box on the *TCA* tab:

- Import TCA Data on page 1465
- Data Processing on page 1465
- TopN Graph Export Tabs on page 1466
- Tree Structure in Left Pane on page 1472

Figure 32-7 TCA Tab Showing Top Peaks

© 2021 All Rights Reserved.

Import TCA Data

Specify the *TCA File Name* in the *Import TCA Data* group box to open the specified toggle data file for analysis. For example, Figure 32-7 shows an opened a data file named `ntc5.ppfdata`.

Note: If no files are opened, all fields except *TCA File Name* remain disabled.

The left side of the tab is a hierarchy viewer that shows all the instances saved in the data file in a tree-like organization. Double click on a node in the tree to expand and collapse that node. The number associated with the name of each node is the average or peak (depending on the *Value Type*) toggle count or high count (depending on the *Data Type*) over the specified period of time.

Data Processing

Use the *Data Processing* group box to specify the following:

1. ***Data Type***: Click to select whether you want to use toggle counts or high counts in DPA. Toggle count is the number of nets within a specific instance at any given FCLK cycle that change value. High count is the number of nets within a specific instance at any given FCLK cycle that have a high value.

2. ***Value Type***: Click to select whether you want to use averaged data or examine peak data.

   Time Average is the count number (toggle or high) averaged over the given time range. *Peak Values* is the count number at individual FCLK cycles. They affect the count numbers displayed for each node on the hierarchical tree. You can choose to show time average or peak value over the time window.

   This parameter affects the *GetTopNInst*. However, it does not affect *GetTopNPeak*. *GetTopNPeak* always uses the peak number.

   This parameter does not affect the export of the toggle count and high count information using the `tca -export` command.

   In addition, this parameter affects the results on the *Graph* display because of the resolution of the canvas, independent of the time range; the graph uses either the time average or the peak value for each pixel. *Time Average* means that, on the *Graph* display, the data is divided into time intervals with each data point on the graph being an average of its interval. *Peak Value* indicates to use the maximum value within each time interval. For a smoother waveform, you should use the *Time Average*.

   For example, the `*.ppfdata` file has `1M` samples and the *Graph* display has `500` pixels horizontally. If you zoom to 100%, then each pixel stands for `2000` samples. You can

© 2021 All Rights Reserved.

select to use time average or peak value over the `2000` samples. If you zoom in to show only `500` samples, then each pixel corresponds to `1` sample. In this case the setting of time average versus peak value does not have effect on the graph.

3. ***Time Range***: The *Start Time* and *End Time* fields limit the time range to gain finer resolution. This affects the *GetTopNInst* and *GetTopNPeak*, graphing, the browser display (values), and the export function. The GUI is refreshed upon clicking the button (same as clicking the *Refresh TCA Display* button).

   a. ***Reset Time Range Icon***: Clicking this button resets the time range to the minimum and maximum time ranges specified by all open files. The open files might have overlapping time ranges. Also, the average counts calculated for a time range that includes a point without any data are the same as the ones calculated when the time range is limited to the point until which data exists. Clicking this button is the same as clicking the *Refresh TCA Display* button. Consequently, the GUI gets refreshed.

   b. ***Get Time Range from Graph Markers Icon***: If the start and end time markers are set in the graph, the time range within those markers is written into the *Start Time* and *End Time* fields.

4. ***Refresh TCA Display***: Clicking this button updates the browser, graphs, and export list. It does not update the topN data. Following are examples to illustrate situations when refresh is useful:

   - If graphs are deleted and you want to rescale the remaining plots to increase the vertical resolution.
   - When the window is resized, the plots are not resized automatically. So, click this button after manually resizing the window.

TopN Graph Export Tabs

The lower right hand side of the *TCA* tab contains the following tabs:

- The *TopN* tab allows you to identify peaks or instances for analysis. For more information, refer to Finding Peaks.
- The *Graph* tab displays waveforms for the toggle data. For more information, refer to Displaying Toggle Count Data Graphically.
- The *Export* tab allows you to create toggle profiles to be viewed in SimVision or Verdi, or in a text file. For more information, refer to Exporting Data to Waveform Viewer.

© 2021 All Rights Reserved.

Finding Peaks

The *TopN* tab (as shown in Figure 32-7) has the following buttons:

- ***AddTopNInst***: Clicking this button finds the top N instances by examining the value calculated based on the selection of *Data Type, Value Type*, and specified *Time Range*.

  It selects `N` instances having the largest values. The values in the *Count Value* column correspond to the values in the hierarchy viewer. If *Time Average* is selected, the *AddTopNInst* button returns the top `N` number of instances with the highest average toggle or high count. If *Peak Values* is selected, the top `N` number of instances with the highest peak values are returned.

  It also automatically adds these to the export list.

  Note: You can select any one or all of the opened `.ppfdata` files.

- ***AddTopNPeak***: Clicking this button examines the toggle data over a time domain to figure out the peak values. First select one or more instances in the hierarchy viewer and click the *AddTopNPeak* button. The selected value(s) will be searched for among the selected instances. If no instances are selected before clicking the button, all instances will be searched. The peaks are arranged in decreasing order, according to data value. Each line shows instance name, peak value, and the time for the peak value.

  Note: More than one peak value can be found from the same instance. The *AddTopNPeak* value is not affected by the *Time Average* or *Peak Values* settings. You can select any one or all of the opened `.ppfdata` files to use, and also highlight in the browser a subset of the instances between all the opened `.ppfdata` files.

- ***PlotTopN***: On clicking this button, the GUI graphs all the instances from the displayed list. If selected and applicable, that is, the times of the peaks are available because you selected the *AddTopNPeak* button, the peak values are also marked on the graphs with a red-colored **X** (cross mark).

- ***Name Style***: You can select this check box to view the instance names with the additional information of filename and/or data type.

Displaying Toggle Count Data Graphically

On the *Graph* tab (as shown in Figure 32-8), you can view the toggle count data (normal or weighted, depending on how they were collected at run time) graphically over a time range and zoom in on areas of interest.

© 2021 All Rights Reserved.

Figure 32-8 Toggle Count Histograms

The *Graph* tab will show the graphs for multiple instances for the specified time window.

1. ***AddPlot***: Select one or more instances in the hierarchy viewer and click the *AddPlot* button to add instances to histogram viewer. You can also use drag-and-drop to add instances from the hierarchy viewer to the histogram viewer.
2. ***DelPlot***: Clicking this button brings up a pop-up window that shows all the instance names on the graph and allows you to delete instances from the graph. There is also a delete button beside each graph, which can be used to remove an instance from the graph.
3. ***DeleteAll***: Clicking this button clears the entire canvas.
4. ***#Plots/Scr***: Sets the number of plots per screen. It determines the height of each plot given the fixed canvas size. The default is 4.
5. ***Overlay***: Maps all the plots to the same shared y-axis for comparison purposes. The names of the plots and control buttons are removed to avoid cluttering the plot. The individual plots can be identified by matching the color to the name in the pop-up window invoked by clicking *DelPlot*.

© 2021 All Rights Reserved.

6. ***Copy***: Copies the current screen to another pop-up canvas.
7. ***Name Style***: Select to view the instance names with the additional information of filename and/or data type.
8. ***Zoom Buttons (+,-, =)***: You can use the following zoom buttons to perform the zoom operations on the graph:
   - **+**: Zoom in the graph along x axis
   - **-**: Zoom out the graph along x axis
   - **=**: Zoom out the graph fully along x axis
9. ***Buttons on the Plots***:

   Each plot has its own set of control buttons. In general, there are three buttons:

   a. X (Cross mark): This deletes the plot for a specific instance from the canvas.

   b. Up arrow - This swaps positions with the plot above it.

   c. Down arrow - This swaps positions with the plot below it.

The graphs can be very sensitive to cursor position because each pixel can represent thousands of samples.

Clicking once in the graph viewer draws a red vertical line on the plot, called a time marker, and also updates the left bottom status field with the corresponding time.

Dragging the mouse with the left button pressed draws two red vertical lines on the plots, one at the point of button press and another at the point of button release. The status field on the left bottom of the graph viewer shows the time corresponding to the two time markers.

A right click on a specific instance plot, only updates the status bar on the right, showing the following:

- Time
- Real data value and instance name

You can also zoom in and out to view the information displayed on the graph. Use the time marker icon (red flags with a small green arrow) to grab the times from the graph (lower left status bar) and put them in the time range fields, *Start Time* and *End Time*. Then, click the *Refresh TCA Display* button to zoom in on the specified time range.

Note:

© 2021 All Rights Reserved.

- While performing the zooming operation, the values shown on hierarchy view are also updated according the current time range.
- The graph viewer is a global view of the toggle count data. It has limited ability to perform zooming and scrolling by specifying the time range. Toggle data can be exported to a waveform viewer for greater viewing capabilities. Use the *Export* tab (shown in Figure 32-9) to perform this operation.

Exporting Data to Waveform Viewer

Use the *Export* tab (as shown in Figure 32-9) to select a set of instances to be exported to a waveform. This set of instances is displayed on the *Export* tab and can also be viewed in the browser.

You can perform the following operations on this tab after selecting an instance or set of instances from the hierarchy viewer in the left pane:

1. ***Add***: Clicking this button adds the selected instance(s) from the browser to the export list for exporting to a waveform. The browser is updated to display this item as italicized to indicate that it is for export.
2. ***Rm***: Clicking this button removes the selected instance from the displayed export list. If nothing is selected from the export list and something is highlighted in the browser, then the browser item is removed. The browser updates the style to be not italicized, indicating the selected item is not for export.
3. ***Add All***: Clicking this button adds to the export list all the instances from either all opened `.ppfdata` files or any one of the opened files. The browser is updated to display all added instances as italicized.
4. ***Rm All***: Clicking this button removes from the export list all the instances from either all opened `.ppfdata` files or any one of the opened files. The browser updates the style to be not italicized for all removed instances.
5. ***Save***: Clicking this button saves the *current* list of instances selected for export to a file. This enables you to restore the selection later by using the *Import* button.

© 2021 All Rights Reserved.

Figure 32-9 Export to Waveform Viewer

6. ***Import***: Clicking this button enables you to import a saved file and all the instances listed in that file, which are also found in the current working files. The export list and the style in the browser will change accordingly.

7. ***Data Filter***: Define different types of filters while performing the export. The raw data will first be processed by the filter before it is saved to the SST2 file. Currently, there are two types of filters. The first filter is for *IntervalAverage*. You must first specify a segment size by clicking the *Properties* button, which is displayed after you select a *User Data Filter for Export*. The specified time range is split into segments according to the specified segment size. An average number is calculated from all data points inside a segment. Then, the average number is saved for each segments. The intention of this filter is to reduce the number of data points in the SST2 waveform.

   The second filter is the *RunningAverage*. It calculates an average number for each sample of the raw data. The time window for the average is centered on the specified sample. The window size can be set by clicking the *Properties* button. The running average does not reduce data size, but produces a smoother waveform by transforming each raw data point into an average with its neighboring data points; you can specify the number of neighboring data points as a time window.

© 2021 All Rights Reserved.

8. ***Waveform***: Invokes SimVision or Verdi, and displays the exported waveform.
9. ***Export Format***: This list box allows lists the format for exported data (that is, either SST2 for SimVision or FSDB for Verdi).

   The export format library is loaded only for one format in a debug session. If xeDebug is opened using the-*sst2* option or by default, the *Export Format* menu shows only *TEXT* and *sst2* options. If, however, xeDebug is opened using the *-fsdb* option, the *Export Format* menu shows the *TEXT* and *fsdb* options.
10. ***Trace File Name***: Specify the name of the trace file to be created and click the button to export the data.
11. ***Name Style***: This list box in the Instances Selected for Export group box allows you to select the way you want to view the instance names in the export list.

    Below this list box, the *Export* tab displays a list of the instance names currently selected.

Note: You must specify a time window for export by entering values in the *Start Time* and *End Time* fields in the *Data Processing* group box. Both toggle count and high count will be exported for each of the selected instances.

Tree Structure in Left Pane

The left pane of the *TCA* tab displays the instances in a hierarchal format. Next to each instance in parentheses is the count and number of immediate child instances. The count depends on the current user selections of data type, value type of average or peak, and time range. The count is updated when you click *Refresh TCA Display* from the *Data Processing* group box.

1. ***File***: Shows the current working file selection, which might be name of any of the opened `.ppfdata` files or `all`. By default, its value is `all`, which indicates that instances of all opened `.ppfdata` files will be listed in a hierarchal format.
2. ***Path***: Specify an instance name and press the *Enter* or *Return* key. Consequently, the left pane gets updated to display all instances in the specified entry and its antecedents. This works on the current working file selection, which can be *all* files.
3. ***Filter***: Clicking the button next to this field or pressing the *Enter* or *Return* key when the cursor is placed in this field, sets the filter on expanding children of the instances in the left pane. It also filters the current display in the left pane to all instances which match the filter; it does not expand instances that are not already shown. It works on the current working file selection, which can be *all* files. Filtering follows the preferences that you set globally for *regex* or *glob*, and case sensitivity. Default is *glob* style case-sensitive filtering.

© 2021 All Rights Reserved.

4. ***Find***: Clicking the button next to this field or pressing the *Enter* or *Return* key when the cursor is placed in this field, highlights in the current left-pane display all instances which match the *find* filter. It works on the current working file selection, which can be *all* files. It does not change the tree structure. Filtering the searched items follows the preferences that you set globally for *regex* or *glob*, and case sensitivity. Default is *glob* style case-sensitive filtering.
5. ***View All***: Clicking this button expands the entire hierarchy for the current working file selection, which can be *all* files.
6. ***View Export Only***: Clicking this button shows in the left pane only instances selected for export and all their antecedents. It works on the current working file selection, which can be *all* files.

Note:

- If you click once on any instance, it is displayed in the *Path* filter.
- If you click twice on any instance, it is displayed in the *Path* filter. Also, if no children of that instance were displayed, the tree expands by displaying its children by one level. If children are already displayed, it will collapse to hide the children of that instance.
- Select a range of instances by selecting an entry from the tree structure and dragging with the left mouse button clicked till the entry you want to select.
- Add to the current selection of instances by keeping the *Ctrl* key pressed while clicking a particular instance to select.
- Drag selected instances while keeping the right mouse button clicked and drop them to the graph.

DPA Probe Tab

The *DPA Probe* tab (as shown in Figure 32-10) allows you to add DPA probes before generating TCF or SAIF files.

Note: The *DPA Probe* tab is enabled only after you open a `.phy` file for DPA operations using the *Setup* tab of the *Dynamic Power Analysis* window.

The following sections describe the use of each GUI element on the *DPA Probe* tab:

- Adding DPA Probes
- Saving or Loading DPA Probes
- DPA Probes Display

© 2021 All Rights Reserved.

Figure 32-10 DPA Probe Tab

Adding DPA Probes

Perform the followings steps in the *Add DPA Probes* group box:

1. Click the menu button adjacent to the *Command Type* field to select one of the `dpa` command options listed below. The fields displayed in the *Add DPA Probes* group box change according to the selected command option to enable you to set the related parameter values.

   - `-add`

     Selecting *-add* is an equivalent to the `dpa -add <nets>` command. You need to specify net(s) name. When you want to specify multiple net names, separate each

© 2021 All Rights Reserved.

name by a space. If any net name includes a backslash or space, use a pair of curly braces to enclose it.

You can drag and drop net names from other browsers, like Design Browser, into the *Nets* text field.

- `-addinst`

  Selecting *-addinst* is an equivalent to the `dpa -addinst [-allnets] -depth <n> <instances>` command. You need to specify instance name(s) and depth value. When multiple instance names need to be specified, separate each name by a space. If any instance name includes a backslash or space, use a pair of curly braces to enclose it.

  If `-allnets` check box is selected, the nets starting with `n#` will also be added, which are otherwise filtered and not added by default.

You can drag and drop net names from other browsers, like Design Browser, into the *Instances* text field.

- `-readNetNames`

  Selecting *-readNetNames* is an equivalent to the `dpa -readNetNames <dpa_info_file_name> [<instances>]` command. This command adds the nets or pins that appear in the specified file, `<dpa_info_file_name>`, which is a text file that contains signal names. Each line should contain one signal name.

© 2021
All Rights Reserved.

You can specify the `[<instances>]` argument if the required nets or pins are not under the top cell.

Click the directory button adjacent to the `rc_dpa_info_file_name` text field to bring up a window that lists all directories and allows you to select the required file.

You can drag and drop net names from other browsers, like Design Browser, into the *Instance* text field.

- `-addpath`

  Selecting *-addpath* is an equivalent to the `dpa -addpath -depth <n> <net1> <net2>` command. You must specify *Net1* and *Net2* to trace back from *Net1* to *Net2* and add all nets on the path. The value specified in the *To Depth* text box indicates the number of levels to be traced back.

You can drag and drop net names from other browsers, like Design Browser, into the *Net1* and *Net2* text fields.

- `-addcone`

  Selecting *-addcone* is an equivalent to the `dpa -addcone -depth <n> <signals>` command.

© 2021 All Rights Reserved.

This command allows you to add all signals in the input cone of logic of the given nets.

You can drag and drop net names from other browsers, like Design Browser, into the *Signals* text field.

- `-addinstpin`

  Selecting *-addinstpin* is an equivalent to the `dpa -addinstpin <instance>` command. Specify an instance name. All input and outputs of the specified instance, and all instances within it are added to the DPA probe list.

  If the *Outputpin only* check box is selected, the *Command Type* is an equivalent to the `dpa -addinstoutputpin <instance>` command which is used to add only outputs of the specified instance and all instances within it.

You can drag and drop net names from other browsers, like Design Browser, into the *Instance* text field.

2. Click *Apply* to add DPA probes based on the parameters specified for the selected *Command Type*.

Note: Clicking *Reset options* enables you to reset the parameter values of the selected command option to default values. You can then specify new parameter values for the selected command option.

Saving or Loading DPA Probes

At any time, you can either save DPA probes to a file or load DPA probes from a file.

© 2021 All Rights Reserved.

To save DPA probes into a file that can be loaded later, first specify a filename in the *DPA Probes File Name* text field and then click *Save*. The *Save DPA Probes* dialog box (as illustrated in Figure 32-11) will be displayed to confirm whether or not you want to save all the added DPA probes to the specified file. Click *OK* to continue.

Figure 32-11 Probe Tab - Save DPA Probes to a File

To load DPA probes from a file, you can do either of the following:

- Enter the name of the file in the *DPA Probes File Name* text field.

  OR

- Select the required file from the window invoked by clicking the directory button adjacent to the text field.

Then, click *Load* to list the DPA probes from the selected file in the *DPA Probes Display* group box. The *Load DPA Probes* dialog box (as illustrated in Figure 32-12) will be displayed to confirm whether or not you want to load the DPA probes from the specified file. Click *OK* to continue.

Figure 32-12 Probe Tab - Load DPA Probes from a File

DPA Probes Display

The *DPA Probes Display* group box enables you to view the added DPA probes according to the specified filter. You can also select the probes that are not required and remove those from the display list.

© 2021 All Rights Reserved.

By default, the *Filter* is set to an asterisk (`*`), which indicates to list all added DPA probes. You can specify a new filter pattern and click the button to bring it into effect. This is equivalent to the `dpa -list -glob <pattern>` command.

The content list can show only the top `1000` DPA probes. The purpose of this limitation is to allow you to browse the DPA probes easily. Therefore, if the number of added DPA probes is more than `1000`, a warning message will be displayed at the bottom of the probe list box (refer to Figure 32-13 for an example). In this case, use a more accurate filter pattern to display only specific probes that you want to see.

Figure 32-13 Probe Tab - Display Added DPA Probes

There are two non-editable text boxes at the bottom of the list to show the total number of *Probes added* and *Probes displayed*.

You can select any probe from the DPA probe listbox.

You can use the following buttons as well:

- Click *Remove* to remove the selected DPA probes.
- Click *Select All* to select all probes that match the specified filter pattern (including those that are not displayed in the probes listbox).
- Click *Deselect All* to deselect all probes.

© 2021 All Rights Reserved.

TCF Tab

The *TCF* tab (as shown in Figure 32-14) enables you to independently generate TCF files. This is equivalent to the following command:

```
dpa -outfile -tcf <tcfFileName> [-instance <instanceName>] \
                                [-masterTCF] [-slaveTCF][-noregulartcf] \
                                [-segment <n>% | <cycleNumber>] \
                                [-start <time>] [-end <time>]
```

Note: The *TCF* tab is enabled only after you open a `.phy` file for DPA operations.

Figure 32-14 TCF Tab

Perform the following steps to generate the TCF files using the *TCF* tab:

1. In the xeDebug console, enter any one of the following commands to add probes for generating TCF files:

```
dpa -addinstoutputpin <instanceName>
dpa -addinstReg
dpa -input_port <instanceName>
```


© 2021 All Rights Reserved.

2. On the TCF tab, specify the *TCF Format* for the files to be generated by selecting either the *Master/Slave* or *Regular* radio button.

   If the *Master/Slave* radio button is selected, the *Generate Master* check box is enabled for selection. This is equivalent to giving the `[-masterTCF]`, `[-slaveTCF]`, and `[-noregulartcf]` options to the `dpa -outfile -tcf` command.

3. Specify the name of the required TCF file in the *Filename* field. This is a mandatory field, which is equivalent to giving the `<tcfFileName>` option to the `dpa -outfile -tcf` command.

4. Specify the top instance name in the *TCF Top Instance* field. This is an optional field. Selecting this field is equivalent to specifying the `[-instance <instanceName>]` option.

   Note: This field allows you to drag and drop instance name from other browsers, such as the Design Browser, into the *TCF Top Instance* field.

5. Specify the required cycle time in the *Start Time* and *End Time* fields. The time unit will be displayed adjacent to these fields. The button next to these fields can be used to reset to the entire range. This is equivalent to giving the `[-start <time>]` `[-end <time>]` options.

   Note: If the selected `.phy` file has data for a specific interval of time, the start and end time is by default displayed in these fields. You will be allowed to specify only a time that falls within that range. For example, `trace.phy` has data for interval `0-7999`, the *Start Time* and *End Time* fields will display `0` and `7999`, respectively. You will be allowed to change the *Start Time* and *End Time* to `100` and `500`, but changing to `7000` and `8500` will generate an error.

6. Specify the *Segment Size*. You can switch the unit as `%` (percentage) or `cycles/time` (number) by clicking the button adjacent to the *Segment Size* field. This field is equivalent to giving the `[-segment <n>% | <cycleNumber>]` option.

7. Click *Generate TCF* to generate the TCF file. The TCF files generated in the current session are listed in the table within the *Generated TCF Files* group box.

   Note:

   - Before clicking *Generate TCF*, add the probes manually (refer to step 1 in this section); otherwise, nothing will be uploaded into the TCF files. Also, a warning message will be displayed in the xeDebug console and status bar of the DPA GUI window.

© 2021 All Rights Reserved.

- If you entered a filename which already exists, a dialog box will be displayed to confirm whether or not you want to overwrite it.

SAIF Tab

The *SAIF* tab (as shown in Figure 32-15) contains the following group boxes that enable you to perform the SAIF-related operations equivalent to using the `dpa -saif` and `dpa -outfile -saif` commands:

- Forward SAIF
- Scope
- Backward SAIF

Note: The *SAIF* tab is enabled only after you open a `.phy` file for DPA operations.

Forward SAIF

The *Forward SAIF* group box provides fields and tabs for loading and filtering the forward SAIF files that provide the description to the tool to generate the backward SAIF files. Perform the following steps in this group box:

1. Enter the *Forward SAIF File Name* in the *Load Forward SAIF File* group box. You can also use the directory button to browse and select a forward SAIF file. This is a mandatory field.
2. Enter the *Module Patterns*. Each pattern should be separated by a blank space. This is an optional field.

   If specified, the load operation looks in the loaded forward SAIF file for only those modules whose names match any of the specified patterns.

   Note:

   - If the module pattern includes space, enclose the pattern within a pair of curly braces.

     For example:

     ```
     {M *}
     ```

   - If the module pattern includes a backslash, enclose the pattern within a pair of curly braces and add an extra backslash before each original backslash.

     For example:

© 2021 All Rights Reserved.

```
{M *} {MX. \\A*}
```

3. Click *Load* to load the specified forward SAIF file(s). This is equivalent to using the following command:

```
dpa -saif -load <saif_file_name> [-module <module_name_pattern>]
```

Figure 32-15 SAIF Tab - Forward SAIF Group Box with File Tab Enabled

The *Forward SAIF* group box also has the following four tabs to view the content of the loaded forward SAIF files:

- ***File***:

  Lists all the loaded forward SAIF files. This is equivalent to using the `dpa -listfile` command. Refer to Figure 32-15.

  You can use *Filter* to show only those files whose name match the specified pattern. By default, the *Filter* field has an asterisk (`*`) that means display *all*.

  At the bottom of the results table, you can see the *#files loaded* and *#files displayed* in non-editable fields.

  Clicking *Remove* removes the selected forward SAIF file(s). This is equivalent to using the `dpa -saif -rmfile <file_name>` command.

© 2021 All Rights Reserved.

Clicking *Remove All* removes all displayed files that match the specified filter.

- ***Lib***:

  Lists all libraries that have been loaded from the forward SAIF files. This is equivalent to using the `dpa -listlib` command.

Figure 32-16 SAIF Tab - Forward SAIF Group Box with Lib Tab Enabled

You can use *Filter* to show only those libraries whose name match the specified pattern. By default, the *Filter* field has an asterisk (`*`) that means display *all*.

At the bottom of the results table, you can see the *#libraries loaded* and *#libraries displayed* in non-editable fields.

Clicking *Remove* removes the selected library. This is equivalent to using the `dpa -saif -rmlib <lib_name>` command.

Clicking *Remove All* removes all displayed libraries that match the specified filter.

Clicking *View* displays detailed information about the library and module such as library name, module name, location, and saif file.

© 2021 All Rights Reserved.

Figure 32-17 Module SDPD Viewer

- ***Module***:

  Lists all modules that have been loaded from the forward SAIF files. Additionally, the library name in which the listed module is included is displayed. This is equivalent to using the `dpa -saif -listmodule [-lib]` command.

Figure 32-18 SAIF Tab - Forward SAIF Group Box with Module Tab Enabled

You can use *Filter* to show only those modules whose name match the specified pattern. By default, the *Filter* field has an asterisk (`*`) that means display *all*.

At the bottom of the results table, you can see the *#modules loaded* and *#modules displayed* in non-editable fields.

Clicking *Remove* removes the selected modules. This is equivalent to using the `dpa -saif -rmmodule <module_name_pattern> [<lib_name>]` command.

Clicking *Remove All* removes all displayed modules that match the specified filter.

Note:

- The modules are grouped by library name and the modules that belong to the same library are sorted by the module name.

© 2021 All Rights Reserved.

- If more than one libraries are loaded, the *Module* tab uses colored background to distinguish the modules in different libraries (that is, the modules from the same library are displayed with the same background color).
- If the *Filter* text field on the *Module* tab has a pattern that includes a backslash, add one extra backslash before each original backslash. For example, `\\MX*`.

  Note: Unlike the *Module Pattern* field within the *Load Forward SAIF File* group box (refer to step 2), you need not enclose the pattern within a pair of curly braces even if the pattern includes a backslash or space.
- To deselect, press the `Esc` key on the keyboard. To select multiple modules from the list, you can use the `Ctrl` (Control) and `Shift` keys.

■ ***Duplicate Module***:

Lists only those modules that appear in multiple libraries. The content of the *Duplicate Module* tab is a subset of the content of the *Module* tab. This is equivalent to using the `dpa -saif -listduplicatemodule` command.

Figure 32-19 SAIF Tab - Forward SAIF Group Box with Duplicate Module Tab Enabled

You can use *Filter* to show only those modules whose name match the specified pattern. By default, the *Filter* field has an asterisk (`*`) that means display *all*.

At the bottom of the results table, you can see the *#duplicate modules loaded* and *#duplicate modules displayed* in non-editable fields.

Note:

- On the *Duplicate Module* tab, if modules are removed, the modules left might no longer have duplicates. In this case, the modules whose duplicates have been removed will disappear automatically from the *Display Duplicate Module* list.
- Lists on all four tabs – *File*, *Lib*, *Module*, and *Duplicate Module*, will be updated as soon as you load a new forward SAIF file.

© 2021 All Rights Reserved.

- Removing elements from one list will also update the other three lists. For example, if you remove one file, other three lists will also be updated by automatically removing libraries or modules within that file.
- Select multiple elements from a list by clicking the required items while keeping either the `Ctrl` or `Shift` key pressed on the keyboard.
- To deselect the item selected last, press the `Esc` key on the keyboard.

Scope

The *Scope* group box (as shown in Figure 32-20) enables you to specify only those instances that should be included in the backward SAIF file. This group box is equivalent to using the `dpa -saif -addscope <instance_name>` command.

Figure 32-20 SAIF Tab - Scope Group Box

Perform the following operations to define the scope:

1. Enter an *Instance Name*(s). Each instance name should be separated by a blank space.

   Note:

   - If the instance name includes a space or backslash, enclose the instance name within a pair of curly braces.

     For example:

     `{dut.cnt_u2.\dat_reg[7] } dut`

   - You can drag and drop instance name from other browsers, such as Design Browser, into the *Instance Name* field.

2. Click *Add* to add the specified instance into scope that needs to apply Forward SAIF. Alternatively, you can press the *Enter* or *Return* key on the keyboard.

All added instances are displayed in the scope listbox.

© 2021 All Rights Reserved.

You can use *Filter* to show only those instances whose name match the specified pattern. By default, the *Filter* field has an asterisk (`*`) that means display *all*. This field is an equivalent to using the `dpa -saif -listscope` command.

Clicking *Remove* removes the selected instances from the scope. This is equivalent to using the `dpa -saif -rmscope <instance_name>` command.

Clicking *Remove All* removes all displayed instances from scope that match the specified filter.

Note: You can select multiple instances from the list by clicking the required instances while keeping either the `Ctrl` or `Shift` key pressed on the keyboard.

Backward SAIF

The *Backward SAIF* group box (as shown in Figure 32-21) is an equivalent to using the following command to generate the backward SAIF file:

```
dpa -outfile -saif <saif_file_name> [-instance <instance_name>] \
                                    [-segment <cycleNumber> | <percent>] \
                                    [-start <time>] [-end <time>]
```

Figure 32-21 SAIF Tab - Backward SAIF Group Box

Perform the following operations using this group box:

- Enter the output filename in the *Backward SAIF File Name* field.This is equivalent to using the `<saif_file_name>` option. This is a mandatory field.
- Enter the *Instance Name*. This is an optional field. Selecting this field is equivalent to using the `[-instance <instance_name>]` option.

  Note: You can drag and drop instance name from other browsers, such as Design Browser, into the *Instance Name* field.
- Enter the *Start Time* and *End Time*. The time unit is displayed next to these fields. Also, there is a button next to these fields that can be used to reset to the entire range. These fields are equivalent to giving the `[-start <time>] [-end <time>]` option.

© 2021 All Rights Reserved.

- Enter the *Segment Size.*This is equivalent to using the `[-segment <cycleNumber>|<percent>]` option. You can switch `%` (percent) and `cycles/time` (number) as unit by clicking the button adjacent to the *Segment Size* field.

- Click *Generate Backward SAIF File* to generate the output file according to the currently specified settings.

  Note: If you entered a backward SAIF filename which already exists, a dialog box will be displayed to confirm whether or not you want to overwrite it.

PowerDebug Tab

DPA enables you to generate the toggle or high count data of instances into the `.ppfdata` file and generate the TCF file which contains the toggle count, high count, and net name for every net.

The *PowerDebug* tab (as shown in 32-22) provides an integrated environment to display the power information of the design including toggle or high number for instances and nets. The data comes from the `.ppfdata` and TCF files that you specify. The *PowerDebug* tab remains grey until you open a `.phy` file or download a design in xeDebug, or open a vvm session in vvmDebug, It provides controls for the following actions:

- Viewing Instance Hierarchy
- Viewing Nets
- Importing Data from the ppfdata and TCF Files
- Finding Top N Instances and Nets
- Normalizing the Toggle or High Number
- Filtering Instances and Nets
- Resetting the Values
- Probing Nets

© 2021 All Rights Reserved.

Figure 32-22 PowerDebug Tab

Viewing Instance Hierarchy

The *PowerDebug* tab shows a tree view of all instances of the entire design hierarchy in the ICE mode, or only the DUT portion of the design in the SA mode. Under the tree view, a list box shows all the nets of a highlighted instance. You can double click on a node in the tree to expand and collapse the node.

Figure 32-23 PowerDebug Tab - Instance Hierarchy View

© 2021
All Rights Reserved.

The *Path:*, *Filter:*, and the other buttons on the top of the *PowerDebug* tab work same as the DUV Design Browser.

For each instance, an instruction is displayed following the instance name. The toggle number is not displayed for the instances that are not recorded in the imported `.ppfdata` file.

For example, an instance is displayed as `cnt_u1 (counter, insts: 48, toggle:41843)`. The instruction following the instance name shows that `cnt_u0` is an instance of module `counter`. It has `48` sub instances. The total toggle number is `41843` in the whole time range.

Viewing Nets

The Nets group box shows all nets of the selected instance.

By default, the nets are displayed in alphabetical order. You can click the *sortByCount* button to see these nets sorted by the toggle or high count. When you click the button, it changes to *sortByName*, and nets are displayed by toggle or high number orders.

Figure 32-24 PowerDebug Tab - Nets

For each net, a toggle or high number is displayed following the net name.

For example, a net is displayed as `clk (regular, toggle: 14998)`. The instruction shows that the `clk` net of the selected instance toggles `14998` times.

Importing Data from the ppfdata and TCF Files

When you specify the `.ppfdata` and TCF files, the toggle or high count data in these files is imported and you can see the toggle or high count data for instances and nets. The toggle or high number for instance comes from the `.ppfdata` file. The toggle or high number for net comes from the TCF file.

© 2021 All Rights Reserved.

On the *PowerDebug* tab, first you should import the `.ppfdata` and TCF files to show toggle or high number in the design.

Figure 32-25 PowerDebug Tab - Importing and Probing

In the *Importing & Probing* group box, you can bring up a directory and select the `.ppfdata` and TCF files. Click the button to import the `.ppfdata` and TCF files. After the files are imported, the toggle or high number is shown in the instance and nets hierarchy view. To clear the imported toggle or high number for instances or nets, you can click the button.

The *Data Type* radio button enables you to select whether you want to see toggle counts or high counts. The default type is *Toggle Count*.

When the `.ppfdata` file is imported, you can see the time range of the imported `.ppfdata` file by moving the mouse cursor over the *Time Range* label.

The following are the requirements for importing a TCF file:

- You can import a regular TCF or a slave TCF file. However, for a slave TCF file, a corresponding master TCF file must exist in the same directory.
- You need to keep the TCF naming style like `xxx.tcf.s.n` or `xxx.tcf.m` for a slave or master file, respectively.

If the imported TCF file is a `.gz` file, the TCF file is automatically extracted from the `.gz` file before importing the file.

Note: If the TCF and `.ppfdata` files are imported together, the time range of the TCF file should be within the time range of the `.ppfdata` file. The instance tree view and *Nets* tree view show power information of the time range in the TCF file.

For example, you import a `.ppfdata` file named `abc.ppfdata` and the time range of the file is `[1fclk, 19999fclk]`. Now, the instance tree view shows the toggle count in `[1fclk, 19999fclk]`. And then, you import a TCF file named `test.tcf.1`, and the time range of the file is `[5000fclk,15000fclk]`. After you import the TCF file, the previous imported `.ppfdata` file is reloaded, and only the toggle count of `[5000fclk, 15000fclk]` is displayed.

© 2021 All Rights Reserved.

If you are trying to import a TCF file whose time range is not covered by the time range of the imported `.ppfdata` file, an error is displayed on the status bar and the import fails.

Finding Top N Instances and Nets

You can define a top `N` number to find the top `N` instances or nets with more toggle or high number in the whole design.

Note: The Instance display list and Nets display list use a different top `N` number.

You can enter an integer number in the *TopN* text box and then click the *Confirm* button. The default value is `0`, which means all instances or nets. Only the top `N` instances or nets which have more toggle or high number are displayed in the hierarchy tree.

You can use the following buttons to browse the top N instances or nets:

- Click the << button to browse the first instance or net with the largest toggle or high number.
- Click the < button to browse the previous instance or net with the larger toggle or high number.
- Click the > button to browse the next instance or net with the smaller toggle or high number.
- Click the >> button to browse the last instance or net with the smallest toggle or high number.

When you click one of these buttons, the respective instance or net is highlighted and the hierarchy tree is extracted around the instance or net.

When you are browsing the top `N` instances or nets, the current index is displayed. You can also directly input an index number and press `ENTER` on the keyboard to go to the specified instance or net.

The index is based on the toggle or high count number sorting, not the instance size. The instances whose toggle or high count number is not recorded in the `.ppfdata` file are not counted. Power domains are not displayed in the instance hierarchy view.

When you are browsing top `N` instances, the full name of the current instance is displayed in the *Path:* text box. When you are browsing top `N` nets, the full name of the current net is displayed in the *TopN Net Name* text box.

For example, if you are browsing the number `9` instance, the number `9` instance is highlighted and expanded, and the full instance name is shown in the *Path:* text box.

© 2021 All Rights Reserved.

Normalizing the Toggle or High Number

You can define a toggle or high number as base number. The toggle or high numbers of all instances or nets are normalized by the base number.

Note: The Instance display list and Nets display list use a different base number.

You can enter a toggle or high number as base number in the *Base:* text box and then click the *Confirm* button to normalize all toggle or high number by this base. To be more accurate, a floating number with five digits after the decimal point is used.

You can also highlight an instance or net and click the button next to the *Base:* text box to use the count number of the selected instance or net as the base value.

For example, if the instance `Top.A.n1` has `11111` toggles and the instance `Top.A.n2` has `33333` toggles. You can set `11111` as the base number. And then, the toggle of `Top.A.n1` shows `1` and of `Top.A.n2` shows `3`. The instance `Top.A.n2` has `3X` toggles than the instance `Top.A.n1`.

Filtering Instances and Nets

You can define a toggle or high threshold number. Only the instances or nets that toggle or keep high more than the specified threshold number are displayed.

Note: The Instance display list and Nets display list use a different threshold number.

You can input a toggle or high number threshold value in the *Thres:* text box. The default value is `0`, which means no threshold. If threshold value is set, the instances or nets that have a fewer number of toggle or high count less than the threshold value are not shown. Only the instances or nets whose toggle or high count is not less than the threshold value are shown.

You can also highlight an instance or net and click the button next to the *Thres:* text box to use the count number of the selected instance or net as the threshold value.

Resetting the Values

The Reset buttons in the Instance hierarchy view and the Nets group box enable you to reset the top `N`, base, and threshold values to the default values for an instance or net, respectively.

You can click the *Reset* button to reset the *TopN*, *Base:*, and *Thres:* options to the default values. Click the *Confirm* button to make the *TopN*, *Base:*, and *Thres:* values effective. Otherwise, the new values are not used.

© 2021 All Rights Reserved.

If all of the *TopN*, *Base:*, and *Thres:* options are defined, all the options are effective together and only the instances or nets that match all of the restrictions are displayed. Toggle number is first normalized by the base number and then filtered by the threshold number.

Probing Nets

You can probe the selected nets and display these probes in the waveform.

In the *Importing & Probing* group box, you can click the *Probe* button to add the objects that you have selected in the *Nets* group box.

For example, when you select nets in the *Nets* group box for an instance `dut.cnt_u0` and click the *Probe* button, the selected nets are added as probes.

The *Display* checkbox is an equivalent to the `xeset -gui dbAutoWave 1` command.

The *AutoUpload* checkbox is an equivalent to the `xeset -gui dbAutoUpload 1` command.

The *Upload* button is an equivalent to the `database -upload` command.

Analyzing Power Consumption with CPF

DPA recognizes the power constraints defined for low-power applications using the Common Power Format (CPF). CPF-aware DPA provides the mechanism to identify the logic applied to the power domains and adjust the power calculations to reduce the dynamic power.

For detailed information on the CPF implementation, refer to Chapter 33, “Specifying CPF Low-Power Intent on Palladium Z1.”

DPA supports CPF while performing the following operations:

- Calculating NTC or WTC

  The NTC or WTC of a power domain can be calculated by summing up toggle count data and high count data from the instances that are included in the power domain.

- Generating power profile in TCF or Joules flow

Working with Power Domain in DPA GUI

You can define instances on the *TCA* tab of the *Dynamic Power Analysis* GUI window. To add a power domain into `*.ppfdata` file, add the power domain name in the DPA GUI.

© 2021 All Rights Reserved.

In the DPA GUI, power domain is displayed as *virtual instance* which is integrated into the hierarchical tree of normal instances in the design, displayed in Figure 32-26. Cadence recommends that you prefix the name of the power domain with `PowerDomain` in order to avoid confusing power domains with normal instances. For example, in Figure 32-26, the names of the power domains are `PowerDomain_PD_1`, `PowerDomain_PD_2`, and `PowerDomain_PD_3`. According to the CPF standard, a power domain can have the same name as the normal instance in the design. The hierarchy for normal instances is shown under *dut* and power domains are shown below normal instances under the Power Domain icon.

On double-clicking the power domain the hierarchical tree is expanded, displaying the instance names for that power domain. This allows you to explore instance's hierarchy starting from a power domain.

Power domains are treated as *virtual instance* and you can perform any operations on the power domain as they perform on normal instances. For example, in Figure 32-26, the power profile graph for power domain `PowerDomain_PD_1` and `PowerDomain_PD_2` is displayed.

© 2021 All Rights Reserved.

Figure 32-26 DPA Power Domain

Creating *.ppfdata File for Power Domain

Specify the instances to be included in the `*.ppfdata` file for power domain using the following command:

```
dpa -setupToggleCount
```

The power domain information is saved in the `*.ppfdata` file. Use the following command to create the `*.ppfdata` file:

```
dpa -getToggleCount
```

If the `dpa -setupToggleCount` command is not specified, automatically a default set of instances is included in the `*.ppfdata` file. All the power domains are included in the default set.

© 2021 All Rights Reserved.

Both the power domains and the normal instances name can be specified with the `dpa -setupToggleCount` command option. For example:

```
dpa -setupToggleCount -addinst normal_inst1 \
normal_inst2 power_domain_1
```

Using Sparse Sampling with DPA

The sparse sampling mechanism allows you to sample once every `N` cycles instead of sampling every `FCLK` cycles. This improves the run-time performance and increases the DPA coverage in LA mode.

Note: Sparse sampling is not supported when performing DPA operations in the Vector Debug mode.

The DPA features of capturing the toggle count data (both NTC and WTC) and generating the TCF/SAIF files take time. Consequently, when DPA needs to be performed on large trace windows with millions of `FCLK` cycles, it takes a long time to upload each cycle. In such situations, using sparse sampling for DPA reduces the amount of data that needs to be uploaded and thus, improves the performance.

In LA mode, Data Capture Card (DCC) can capture only once. Therefore, capturing multiple times and merging them into a large trace window is not possible. This limits DPA in LA mode to the maximal trace depth, which is typically `1M` cycles. With sparse sampling, the trace window can cover more cycles.

The sparse sampling is useful to analyze average power; but it is not good at peak power. Select an appropriate sample rate according to the performance and power analysis requirements. With higher `N` value, you can get larger improvement on performance and trace window size for LA mode. However, there is a higher risk of missing some import power usage change, such as peak.

Sparse sampling is transparent when the NTC or WTC feature is used. You can run all XEL commands and GUI as if sparse sampling was not enabled.

Sampling rate is set at multiple of frame size, which is typically 32 cycles for most designs.

Note: This document uses 32 in the following sub-section. However, in VXE, the frame size need not be always equal to 32.

Enabling and Disabling Sparse Sampling

Use the following XEL command to enable or disable sparse sampling:

```
dpa -sparse_sampling [<N>]
```


© 2021 All Rights Reserved.

In the above command, the numeric value specifies the sample rate at which a frame should be captured after every <*N*> frames. When a frame is captured it contains typically 32 cycles, from which 31 toggle counts and 32 high counts are generated. When capturing NTC or WTC data, an average is calculated and only one toggle count number and one high count number is provided for those 32 cycles.

Specifying <*N*> as `0` or `1` disables sparse sampling.

When the `dpa -sparse sampling` command is used without specifying a value for <*N*>, the current sample rate is returned.

After enabling sparse sampling, you can start the emulation using the `run` command. However, after running the emulation process, any conditional acquisition statements in your SDL program will be ignored.

Note: When using the xeDebug GUI for DPA, before running the emulation, specify a sample rate on either the *Control* tab or the *InfiniTrace* tab. For information about these two tabs, refer to Control Tab and InfiniTrace Tab in Appendix 42, "Using the xeDebug GUI."

Estimating Power Using TCF/SAIF Files with Sparse Sampling

You can perform TCF and SAIF-based power estimation even when using sparse sampling for DPA. This approach speeds up the power estimation by sampling only a part of the run activity at a fixed rate.

Before generating the `*.phy` file during run time, you should specify the sparse sampling ratio using the `dpa -sparseSampling <N>` command.

Improving Accuracy of Power Estimation

Though gate-level netlists provide more accurate power estimates than netlists generated from RTL designs, there are considerations and commands that can help you improve accuracy.

Generating High Quality TCF Files

To generate high quality TCF files, consider the following:

- Ensure that the time taken for generating a particular TCF file is the same as expected. The segment time can be retrieved from the TCF file header or TCF log file.

© 2021 All Rights Reserved.

- Always specify the real clock frequency because Palladium Z1 uses cycle-based method for calculating toggle information.
- If you are using ICE, take into account the mode of clocking that you are using, that is, CAKE 2X or CAKE 1X.
- Adjust the `traceMemSize` run-time parameter to cover the number of samples needed. This helps because the physical trace memory available in Palladium Z1 limits the time range covered by TCF files. If the time range is larger than what the trace buffer can cover, then run just enough cycles to fill the trace memory, and perform upload for each run.

Estimating Accurate Power Consumption of RTL Designs

Use the power estimation tool, Joules, to perform the following operations to improve accuracy of the RTL-level power estimation:

- Build an RTL power model that puts additional information into the Joules database for calculating power. Do this operation only for one TCF file because the model remains the same with the activity. After power has been reported once based on the model, acquire a scaling factor used in subsequent power calculations as shown in the following example:

```
build_rtl_power_models -clean_up_netlist // Enhance database with model
information
report_power -tcf_summary // report TCF summary
set rtlpe_factors [get_attr lp_rtlpa_scale/des*/* ]// Grab scaling factor
```

- Move to a generic gate-level netlist using the following command:

```
synthesize -to_generic
```

- Include wire load models from the Liberty files, as shown below. This can result in multiple net name mismatches, but there will also be a lot of nets that are recognized.

```
set wlm [find / -wireload_selection *]
set_attr wireload_selection [lindex $wlm 1]
```

Estimating Accurate Power Consumption of Gate-Level Netlists

The following ways can help you improve accuracy for gate-level power analysis:

- Include clock tree and physical transmission delays. Always use this information, if available. Reporting flat power instead of hierarchical power allows you to analyze the local contribution of clock buffers in a better way. The hierarchical power report will likely place all clock tree power in the top level instance.

© 2021 All Rights Reserved.

- Use the `read_spf` command to add SPEF file information, especially when using wire load models. The SPEF files contain resistance, capacitance, and inductance for wires based on the physical layout.
- Use the `read_sdc` command to retrieve timing and clocking information from an SDC file.

Cadence recommends that you always add wire load models, SPEF files, and SDC files when available.

DPA Licenses

The DPA license control requires 39SDPA.

When using the xeDebug tool, you have the choice of using DPA licenses or other licenses while running SimVision and Joules:

- For viewing waveforms, either use the `DPA_SimVision_2` license or the Xcelium `Affirma_sim_analysis_env license`.
- For Joules usage, either use the `DPA_Power_Table_Generation_2 license` or the Joules license.

39SDPA contains the following five licensed features for different DPA functions:

- `DPAPXP_ToggleCountAnalysis_2`
- `DPAPXP_TCAExport_2`
- `DPAPXP_ToggleCountFormat_2`
- `DPAPXP_SimVision_2`
- `DPA_PowerTable_Generator_2`

These licensed features provide the following functionalities:

- By default, the *Power Tower* icon is set to ON in the xeDebug GUI, irrespective of the availability of the licenses. Therefore, you can simultaneously work on the different DPA functions from different xeDebug processes.
- The xeDebug tool supports invoking SimVision directly. The `DPAPXP_SimVision_2` license is released when the SimVision process exits, without requiring you to exit from xeDebug.
- The `DPAPXP_SimVision_2` license is used for the `dpawave` program.

© 2021 All Rights Reserved.

The `DPA_PowerTable_Generator_2` license is used for `dpa_joules` program.

DPA Tutorial Examples

A tutorial example for DPA can be found in the `<installation_dir>/gift/tutorial/dpa` directory. The `README` file in this directory provides the instructions on the use of the tutorial.

© 2021 All Rights Reserved.

33

Specifying CPF Low-Power Intent on Palladium Z1

Common Power Format (CPF) is the standard format for specifying design- and technology-related power constraints for low-power applications. CPF provides the infrastructure to combine the RTL design files with CPF commands and simulate operations of a power saving circuit.

The supported low-power verification techniques are Power Shut-Off (PSO), Multiple Supply Voltage (MSV), and Dynamic Voltage Frequency Scaling (DVFS). For details, refer to Selecting Low-Power-Aware Verification Method Based on Design Type.

Palladium Z1 supports a limited set of CPF 1.1 and 2.0 standards. Design Considerations for Low-Power Verification discusses the deviations from the supported CPF standards and restrictions that need to be observed while verifying a low-power-aware design on Palladium Z1. For the list of supported and unsupported CPF commands, refer to Table 33-1 in Support for CPF Commands on Palladium Z1.

For information on the CPF command syntax, usage of CPF, and changes in the CPF 1.1 standard, refer to the *Common Power Format Language Reference*, *Common Power Format User Guide*, and *What's New in Common Power Format* in the *Encounter RTL Compiler* documentation set.

For information on the CPF 2.0 standard, refer to *Si2 Common Power Format Specification™*, *Version 2.0* document.

Support for CPF Commands on Palladium Z1

VXE does not support the entire set of functionality available in the *Cadence Common Power Format Language Reference, version 1.1* and *version 2.0*. For option-wise details and syntax of the supported CPF commands, refer to the *Common Power Format Commands* chapter in the *VXE Command Reference Manual*.

The following table lists the supported and unsupported CPF commands:

© 2021 All Rights Reserved.

Table 33-1 Supported and Unsupported CPF 1.1 and 2.0 Commands

Category	Supported CPF Command(s)	Unsupported CPF Command(s)
Version command	set_cpf_version	-
Modeling commands	end_design get_parameter set_design set_instance update_design [1]	set_pad_ports
Power mode commands	assert_illegal_domain_configurations create_mode [a] create_mode_transition create_power_mode	set_power_mode_control_group end_power_mode_control_group update_power_mode
General-purpose commands	find_design_objects [a] include set_hierarchy_separator	set_power_unit set_time_unit set_array_naming_style set_register_naming_style
Macro support commands	set_macro_model end_macro_model	set_floating_ports set_input_voltage_tolerance set_wire_feedthrough_ports

© 2021 All Rights Reserved.

Category	Supported CPF Command(s)	Unsupported CPF Command(s)
Design constraint commands	create_isolation_rule create_nominal_condition create_power_domain create_state_retention_rule	create_analysis_view create_bias_net create_global_connection create_ground_net create_level_shift_rule create_operating_corner create_pad_rule create_power_nets create_power_switch_rule define_library_set identify_power_logic identify_secondary_domain set_equivalent_control_pins set_power_target set_switching_activity update_level_shifter_rules update_nominal_condition update_power_switch_rule update_isolation_rules update_state_retention_rules

© 2021 All Rights Reserved.

Category	Supported CPF Command(s)	Unsupported CPF Command(s)

Library-related commands	-	`define_always_on_cell`
		`define_isolation_cell`
		`define_state_retention_cell`
		`define_level_shifter_cell`
		`define_open_source_input_pin`
		`define_power_clamp_cell`
		`define_power_switch_cell`
		`define_related_power_pins`

Verification support commands	-	`create_assertion_control`

1. This command has been introduced in the CPF 2.0 standard.

Note: When modeling low-power intention of hard-custom IPs in a CPF file, you can use only a specific set of CPF commands and options in the different parts of the CPF file. For more information, refer to the sub-sections of Modeling Hard-Custom IP Low-Power Intention in a CPF File.

CPF 2.0 Features Supported on Palladium Z1

The section lists the CPF 2.0 features supported on Palladium Z1 in the table below. For detailed information about these features, refer to the *Enhancements and New Capabilities for CPF 2.0* chapter of the *Si2 Common Power Format Specification, Version 2.0*.

Enhancements and new capabilities for CPF 2.0	
Design/instance semantic change	■ `set design` semantic change ■ Append power intent ■ Virtual port expend ■ Instance port mapping

© 2021 All Rights Reserved.

Enhancements and new capabilities for CPF 2.0	
Improve hierarchical flow	■ Port mapping for macro ■ Domain includes leaf cell ■ Multiple cells share model ■ Find design objects
Simulator-related	■ Specify simulation action to control PSO behavior ■ Test bench CPF model ■ Power off corruption model
Exclude instances or pins in power domain	
Additional operators for expression **Note:** Only logical operators ! (unary logical negation), && (logical AND), and || (logical OR) are supported with operands as a port or pin. However, bitwise XOR operator (^) is not supported.	

Design and Instance Semantic Differences in CPF 1.1 and CPF 2.0

Following are the important differences in semantic of design and instance related commands in context of CPF 1.1 and CPF 2.0 standards:

- The semantics of the `set_design` command have been changed as follows:
  - CPF 1.1 specified the power intent for one *module*.
  - CPF 2.0 specifies an entity, referred to as a *power design*, that defines power intent. A power design can be applied to
    - A top design, or
    - Instantiations of any module, or
    - Instantiations of one or more explicitly specified modules
- The rules for appending power intent to an existing design are different as follows:
  - CPF 1.1 appends the power intent to an existing design using additional pairs of the `set_design` and `end_design` commands.

© 2021 All Rights Reserved.

- CPF 2.0 appends the power intent to an existing power design using the new `update_design` command paired with the `end_design` command.

- The semantics for virtual ports have been expanded as follows:
  - In CPF 1.1, the `-ports` option of the `set_design` command specified input ports only.
  - In CPF 2.0, the new `-input_ports`, `-output_ports`, and `-inout_ports` options of the `set_design` command explicitly specify the input, output, and inout ports.

  Note: The `-ports` option still exists for backward compatibility.

- In CPF 2.0, the semantics of the `-port_mapping` option of the `set_instance` command have been clarified to include the specification of real ports in addition to virtual ports.

Processing CPF Information on Palladium Z1

Perform the following steps to use CPF for analyzing low-power-aware designs on Palladium Z1:

1. Create a CPF file to specify the power intent for the design and the technology information. Refer to Creating CPF Files.
2. Add the power information recorded in the CPF files to your design (RTL design or gate-level netlist) by using the `readCPFfile` command. Refer to Adding CPF Information to a Design.
3. Compile your low-power-aware design that comprises the design files and the CPF files. Refer to Compiling a Design with Low-Power Intent.

   The compiler processes these files to generate the emulator executable image.

   Note: On Palladium Z1, the behavior of the CPF files is based on the synthesized gate-level netlist, instead of the original RTL design. Therefore, low-power-aware designs compiled in ICE and SA modes can show differences in behavior because of the tool used to synthesize the design, that is, either the HDL-ICE Compiler or the IXCOM tool.
4. At run time, download the emulator executable image on the debugger. Refer to Running and Debugging a Compiled Design with Low-Power Intent.
5. Check the signal values in the SST2 or FSDB format waveforms by using the SimVision or Novas tools (Verdi or Siloti).

© 2021 All Rights Reserved.

Figure 33-1 illustrates the flow for processing CPF information in VXE.

Figure 33-1 Flow for Processing CPF Files in VXE

Creating CPF Files

Create a CPF file that contains the following information:

- CPF design model defined using the `set_design` and `end_design` CPF commands

  Note: Multiple embedded `set_design` and `end_design` command pairs are not supported.

© 2021 All Rights Reserved.

- Power domains created using the `create_power_domain` CPF command
- Nominal conditions and power modes defined using the `create_nominal_condition` and `create_power_mode` CPF commands, respectively
- Rules created for isolation using the `create_isolation_rule` CPF command
- Rules created for state retention using the `create_state_retention_rule` CPF command

To use multiple CPF files, you must create a master CPF file and use the `include` command to source into it the supporting CPF files, which contain the additional CPF information.

For example, create a `design_TOP.cpf` master CPF file containing the basic information used by the tool. Also, create two supporting CPF files, `library_1.cpf` and `impl_1.cpf`, that contain the additional power information. To add the content of the supporting files in the master CPF file, specify the following two commands in `design_TOP.cpf`:

```
include library_1.cpf
include impl_1.cpf
```

While running and debugging your low-power-aware design using the xeDebug GUI, you can view and edit the content of the associated CPF files in the *Power Format Verifcation* window. For details, refer to Using the xeDebug GUI for Low Power Verification.

Adding CPF Information to a Design

To read the CPF file listing the CPF commands into the database, use the `readCPFfile` command that has the following syntax:

```
readCPFfile [-checkSyntaxOnly | -checkOnly | -noLinter | -force] <cpfFileName>
readCPFfile -clear
readCPFfile [-help]
```

The `readCPFfile [options] <cpfFileName>` command should be issued after the design is imported, but before the `precompile` command; otherwise, the precompile and compile processes do not take in the content of the CPF file. For command syntax and description of the `readCPFfile` command, refer to the *Compiler Commands* chapter of the *VXE Command Reference Manual*.

Note:

- If you only want to check CPF syntax, use `checkCPFfile` command. The `checkCPFfile` command internally invokes the Conformal Low-Power (CLP) integrator to check CPF commands for syntax errors.

© 2021 All Rights Reserved.

- If you want to check both syntax and semantics errors, and save CPF information into database, use `readCPFfile` command. The `readCPFfile` command directly supports hierarchical-CPF files and does not invoke CLP integrator.

The `readCPFfile` command performs the following functions:

- Translates the CPF information into User Data
- Issues warning messages for unsupported CPF commands
- Checks the CPF commands for grammar errors
- Checks the VXE-supported CPF commands for semantic errors

The CPF-related settings can be viewed at both compile and run time by using the following `lp` command:

```
lp [assert_illegal_domain_configurations |
    create_nominal_condition |
    create_isolation_rule |
    create_power_domain |
    create_power_mode |
    create_state_retention_rule |
    set_cpf_version |
    set_design |
    set_instance |
    set_macro_model |
    set_hierarchy_separator
```

Note: At compile time, you can execute the `lp` User Data command at any point after executing the `readCPFfile` command.

Compiling a Design with Low-Power Intent

In ICE mode, use the normal compile flow discussed in Chapter 9, “Compiling Designs for In-Circuit Emulation with xeCompile” to compile a design with low-power intent saved in CPF file(s). In SA mode, you must keep in mind the considerations discussed in Compilation - CPF Constraints - Partitioning and Support for CPF at Compile Time by IXCOM Compiler.

During compilation, the compiler checks the CPF files, interprets the CPF specification, and inserts the derived instrumentation logic in the emulator executable image.

If you decide to not model the low-power behavior in the design, but have already specified the CPF file using the `readCPFfile` command, use the `precompile -noCPF` XEL command to turn off the CPF support during compilation. Consequently, the design is compiled without the CPF information. Alternatively, use the `precompileOption -add {noCPF}` XEL command.

© 2021 All Rights Reserved.

Running and Debugging a Compiled Design with Low-Power Intent

In both ICE and SA modes, perform the steps listed below to run and debug a compiled design with low-power intent. The differences in approach that are specific to SA mode are discussed in Support for CPF at Run Time in SA Mode.

1. Download the emulator executable image on the debugger.

   Note:

   - By default, CPF support is enabled at run time. However, if required, use the `lp lpDisable true` command to disable the CPF support. This command is useful when you do not want to keep different design databases, such as one with CPF and the other without CPF support. If you run into loop-breaking issues, you do not need to fix the loop-breaking issues in both the databases. Therefore, using the ability to enable and disable CPF support at run time, you can generate only one database with CPF during compilation.
   - After download, use the `lp -value -<lpObjectType> <lpObjectName>` command at any time to get the actual value of the CPF objects.

2. Add power domains, power modes, general modes, and rules to the probes set using the following CPF-specific options of the `probe` run-time command:

   - `-addPowerDomain` <*`powerDomainName`*> - probes control signals and instance signal values

     Use the `-addPowerDomain` option to add the `cpf_power_domains` object into the waveform database at the top hierarchy.

     Each power domain is its own instance under the `cpf_power_domains` object. Each power domain has the following properties, shown as instances:

     - `active_state_conditions`
     - `base_domains`
     - `instances`
     - `interfaces`
     - `shutoff_condition`
     - `state_retention` (if applicable)

   - `-addRetention` <*`stateRetentionRuleName`*> - probes control signals and secondary domain

© 2021 All Rights Reserved.

Use the `-addRetention` option to add the `cpf_state_retention_rules` object into the waveform database at the top hierarchy. Each retention rule is its own instance.

- `-addIsolation` <*`isolationRuleName`*> - probes control signals and attributes

  Use the `-addIsolation` option to add the `cpf_isolation_rules` object into the waveform database at the top hierarchy.

- `-addPowerMode` <*`powerModeName`*> - probes the power mode transition waves and attributes

  Use the `-addPowerMode` option to add the `cpf_power_modes` object into the waveform database at the top hierarchy.

- `-addPowerModeTransistion` *`<ModeTransitionName>`*

- `-addMode` *`<modeName>`*

Note: With each of the above-listed options, use the (*) wildcard character.

The following are added into the waveform database at the top hierarchy if any CPF probes are added:

- `cpf_controls` containing the `lpDisable` signals.
- `cpf_power_mode_transition` containing the `power_mode` information.

3. View the selected power information in the SimVision or Verdi waveform viewer.

In xeDebug, probed CPF information is directly generated into the SHM or FSDB database. Therefore, after opening the SHM or FSDB database, you can view the probed CPF information.

For each power domain, the waveform displays the power-domain states to indicate the times when the power is up, down, or standby; and when the state retention happens. Additionally, for each isolation rule specified in the CPF files, the waveform displays the isolation on and off status.

In the waveforms, the values of the potential interface pins in the power domain are displayed as `0`, `1`, or `X` to depict the power-loss state of the pins. The value `X` is displayed for signals in power-loss state. The potential interface pins are the pins of those instances declared in the listed power domain instances using the `create_power_domain` command.

Note: To display the original emulator value of the signals in power-loss state instead of displaying the value `X`, use the `lp powerLossXDisplay [true|false|1|0]` command.

© 2021 All Rights Reserved.

When you specify the `-all` option with the `probe` command, the value of signals in power-loss state is displayed as `1` or `0` instead of `X` in the waveform even if the `lp powerLossXDisplay` command has been set to `true`. The disabling of the feature of displaying the value of signals in power-loss state as `X` happens automatically because the `-all` option might initiate probing of the internal instrumentation nets too. Palladium Z1 cannot gaurantee the correctness of the power-loss information corresponding to the internal instrumentation nets.

To enable the feature again after it has been automatically disabled, perform the following steps:

a. Execute the `lp powerLossXDisplay` command.

b. Execute the `probe` command without using the `-all` option.

c. Execute the `database` command to create a new waveform database.

Suppose, a user executes the following set of commands including the `probe -all` command:

```
XE> lp powerLossXDisplay true
XE> probe -all dut
XE> database t
XE> database -upload
```

In waveform database `t`, if the driver of a signal is in power-loss state, the value of the signal will be displayed as either `0` or `1`, instead of `X`.

To enable display of value `X` for signals in power-loss state, you should instead give the following set of commands:

```
XE> lp powerLossXDisplay true
XE> probe dut
XE> database tX
XE> database -upload
```

In waveform database `tX`, the signals with the driver in power-loss state will be displayed with value `X`.

At the `XE>` prompt, you can also execute the `value` run-time command to know the power-down condition. The value returned will be either `0`, `1`, or `X`; where `X` implies that the net is in shut-off power domain.

Refer to Figure 33-2 that illustrates a waveform depicting power-loss display in `X` value.

© 2021 All Rights Reserved.

Figure 33-2 Waveform Displaying Power Loss in 'X' Value

Using –power_up_states and –power_down_states with Palladium Z1

The `-power_up_states` and `-power_down_states` options of the `create power domain` command affect the values of state devices that do not use retention. The `-power_up_states` option determines the values of state devices when power shut-off condition is removed. When `lp powerLossXDisplay` is used, the values will change from `X` to the value specified in `-power_up_states`. If `-power_down_states` is used with `lp powerLossXDisplay`, the power down values will only show `X` until power is back. Then, as long as there is no `-power_up_states` and no state retention, the state devices will contain the values determined by `-power_down_states`. If `lp powerLossXDisplay` is false, the state devices will show the `-power down states` immediately on power shut-off.

The purpose of `-power_down_states` is to check the effect on the design if power down happens and isolation is not complete.The purpose of `-power_up_states` is to check if any non-retained sequential elements during power off period causes design chaos when power is back.

© 2021 All Rights Reserved.

Xcelium does not support the `-power_down_states` and `-power_up_states` options. Xcelium uses `X` to model power loss corruption in the design. Xcelium handles power back re-initialization by calling initial blocks. For more information on Xcelium solution on power down and power back, refer to the Xcelium documentation.

Timing considerations related to –power_up_states and –power_down_states

In silicon, state devices require some stabilization time after power is applied before they can be clocked with new values. In Palladium, this stabilization time is in the form of `FCLKS`. It can take up to four `FCLKS` for state devices to accept new values when `-power_up_states` is used and up to three `FCLKS` when `-power_down_states` is used. The number of `FCLKS` is less if the value of the state device does not change, that is, `0` remains `0` or `1` remains `1`.

CPF does not specify the delay related to this power back. It depends on each individual tool to determine how to implement the power back re-initialization, either with delay, or without delay.

The waveform in Figure 33-3 shows the clock changing. But this is misleading because the state devices within the shut-off power domain are not allowed to clock until the required `FCLK` delay has passed. This behavior does not match simulation and is a by-product of instrumentation compiled into the emulator for CPF.

In Figure 33-3 the design has used CAKE 2X clocks. (two `FCLKS` cycles per design `CLK` cycle) You can see that at time 2410 ns, the rising edge of the clock does not change the output of the `FF` and `raddr_r` to match the input. The clock pin on the `FF` is effectively disabled until just before 2420 ns. The clock edge at 2420 ns, which is four `FCLKS` after power on, is allowed to clock the `FF` and `raddr_r` changes to match `raddr_in`. If this was a CAKE 1X design, `raddr_r` would not update until 2440 ns. If the `-power_up_states` option is not used with CAKE 2X, then `raddr_r` does follow `raddr_in` at 2410 ns.

If you want to ignore any delay due to `-power_down_states` and `-power_up_states`, CAKE 4X can be used. This puts all four `FCLK` cycles within a single user clock.

© 2021 All Rights Reserved.

Figure 33-3 Waveform Displaying CAKE 2X Clocks

Displaying PSO Status

To probe the power domain PSO conditions, execute the following `probe` command before using a `run` or `database -upload` command:

```
probe -addPowerDomain <powerDomainName>
```

For example:

```
XE> database -open testCPF
XE> probe -addPowerDomain PD1
XE> run 1000
XE> database -upload
```

Here, an SST2 database is generated in `testCPF`. This database stores the power domain `PD1` related information including the PSO condition, save/restore condition, nominal conditions status, and interface pins waveforms. These conditions and pins can be viewed using SimVision.

To probe the *on* and *off* status of the save/restore condition in the specified state retention rule, use the following `probe` command:

```
probe -addRetention <stateRetentionRule>
```

Important

The `probe -addRetention` command should be used before the `run` and `database -upload` commands.

For example:

```
XE> database -open testCPF0
XE> probe -addRetention rt1
```


© 2021 All Rights Reserved.

```
XE> run 1000
XE> database -upload
```

The above set of commands, first generate an SST2 database in `testCPF0`. Then, the state retention rule, `rt1`, records two state-retention conditions, `rt1_save` and `rt1_restore`. Use SimVision waveform viewer to view these conditions from the `testCPF0` SST2 database.

Execute the following command to probe the `on` and `off` isolation conditions of a specific isolation rule:

```
probe -addIsolation <isolationRuleName>
```

The above command should be used only before using a `run` or `database -upload` command.

For example:

```
XE> database -open testCPF1
XE> probe -addisolation iso1
XE> run 1000
XE> database -upload
```

An SST2 database is generated in `testCPF1`. Using SimVision waveform viewer, the special isolation signal, `iso1`, can be viewed from the `testCPF1` file as illustrated in Figure 33-4.

Figure 33-4 Waveform Displaying Power Domains in SimVision

© 2021 All Rights Reserved.

Profiling Power Mode Transition

A power mode is a static state of a design in which each power domain operates on a specific state: up, down, or standby.

If the `create power mode` commands are present in the CPF file, whenever SST2 or FSDB waveforms are generated for probes at run time, the power mode transitions are automatically generated in the same waveform database. Consider the following set of commands:

```
XE> database -open testCPF
XE> run 10000
XE> database -upload
```

In the above example, a special signal, `power_mode_transition`, will be generated in the `testCPF` database. Using the `database -upload` command after the `run` command enables you to view the power mode transitions in the SimVision waveform viewer as illustrated in Figure 33-5.

Note: Use the `probe -addPowerMode <powerModeName>` command to display the status of a specific power mode in the SST2 or FSDB waveform database.

Figure 33-5 Waveform Displaying Power Mode Transition in SimVision

© 2021
All Rights Reserved.

Specifying CPF Conditions in SDL Expressions

Palladium Z1 emulator allows you to specify CPF conditions in SDL expressions for better verification and debugging of the low-power intent in your designs.

The CPF conditions can be used to do the following:

- Identify whether the state of a CPF object is true or false.
- Identify whether any control signals are corrupted due to power loss. The scope of control signals includes any power shut-off control signals, isolation conditions, or state retention control signals (defined using options such as `-save_edge`, `-restore_edge`, `-save_precondition`, `-restore_precondition`, `-retention_precondition`, or `-active_state_conditions`).
- Define the CPF illegal power control sequences and then check at run time to see whether the sequence happens or not.

Following is the basic format of CPF conditions in an SDL expression:

```
[ power :: ] <cpfObject>
[ power :: ] <cpfObject>@<cpfObject>
[ power :: ] <cpfObject>@<cpfOption>
[ power :: ] <cpfObject>@corrupted.<cpfObject>
[ power :: ] <cpfObject>@corrupted.<cpfOption>
[ power :: ] <cpfObject>@corrupted@<nominalCondition>
```

Here, `power ::` indicates that the condition is related to a CPF file. This is optional. The purpose of using it is to avoid having a signal with the same name in the design database.

`<cpfObject>` is a CPF object defined using the `create_...` CPF commands, that is nominal condition name, power domain name, retention rule name, isolation rule name, power mode name, general mode name, or illegal domain configuration name.

`<cpfOptions>` is an option of the `create_...` CPF command, such as `-shutoff_condition` is an option of the `create_power_domain` command.

Using CPF Conditions in SDL Expressions

The following are the basic CPF commands:

- If the `<powerMode>` defined with the `create_power_mode` command is true

  ```
  power :: <powerMode>
  ```

- If the `<illegalDomainConfiguration>` defined in the `assert_illegal_domain_configurations` command is true

  ```
  power :: <illegalDomainConfiguration>
  ```


© 2021 All Rights Reserved.

- If the <*mode*> defined in the create_mode command is true

  `power :: <mode>`

- Assume <*powerDomain*> is defined with the create_power_domain command, <*nominalCondition*> is defined with the create_nominal_condition command. To check whether <*powerDomain*> is in <*nominalCondition*>

  `power :: <powerDomain>@<nominalCondition>`

- If the <*powerDomain*> is in power-off state

  `power :: <powerDomain>@shutoff_condition`

- If the retention rule save_edge is true

  `power :: <retentionRule>@save_edge`

- If the retention rule restore_edge is true

  `power :: <retentionRule>@restore_edge`

- If the retention rule save_precondition is true

  `power :: <retentionRule>@save_precondition`

- If the retention rule restore_precondtion is true

  `power :: <retentionRule>@restore_precondition`

- If the retention rule retention_precondtion is true

  `power :: <retentionRule>@retention_precondition`

- If the isolation rule isolation_condition is true

  `power :: <isolationRule>@isolation_condition`

- If the power shut-off control signals in the given <*powerDomain*> are in power-loss state

  `power :: <powerDomain>@corrupted.shutoff_condition`

- If the given <*nominalCondition*> corresponding power active_state control signals in given <*powerDomain*> are in power loss

  `power :: <powerDomain>@corrupted@<nominalCondition>`

- If the retention rule save_edge control signals are in power loss

  `power :: <retentionRule>@corrupted.save_edge`

- If the retention rule restore_edge control signals are in power loss

  `power :: <retentionRule>@corrupted.restore_edge`

- If the retention rule save_precondition control signals are in power loss

  `power :: <retentionRule>@corrupted.save_precondition`

- If the retention rule restore_precondtion control signals are in power loss

© 2021 All Rights Reserved.

```
power :: <retentionRule>@corrupted.restore_precondition
```

- If the retention rule `retention_precondtion` control signals are in power loss

```
power :: <retentionRule>@corrupted.retention_precondition
```

- If the isolation rule `isolation_condition` control signals are in power loss

```
power :: <isolationRule>@corrupted.isolation_condition
```

You can use CPF conditions to define CPF illegal sequence, and then do CPF illegal sequence checking during run time. To check the illegal sequence, enable the isolation rule `ISO`, before power off in power domain `PD1`, by adding the following statement to the SDL file:

```
if (PD1@shutoff_condition == 'bP && !(ISO@isolation_condition))
    trigger;
```

Using the xeDebug GUI for Low Power Verification

For power verification in ICE and SA modes, you need to invoke the *Low Power Verification* window by performing the following steps:

1. Invoke the xeDebug GUI using the following command:

```
xeDebug -gui
```

2. Click the *Open Design* button.
3. In ICE mode, after specifying the emulator host configuration, click the *Download* button; whereas, in SA mode, click the *Start Session* button. After the processing completes, the toolbar at the top of the xeDebug GUI window is activated.
4. Click the lightbulb icon ( ) from the toolbar. The *Low Power Verification* window that helps in debugging low-power-aware designs is displayed. Some fields displayed on this window differ based on the mode for which the design was compiled, that is, ICE mode (see Figure 33-6) or SA mode (see Figure 33-7).

© 2021 All Rights Reserved.

Figure 33-6 Low Power Verification Window (ICE Mode)

© 2021 All Rights Reserved.

Figure 33-7 Low Power Verification Window (SA Mode)

The *Low Power Verification* window provides the options for browsing the Palladium CPF implementation, and run-time operations like `enable`, `disable`, `probe`, and viewing and selecting instrumentation logic for probing and SDL. These options are displayed within different group boxes that are discussed in the steps given below.

5. In the *Runtime Control* group box, you can perform the following actions:

   a. The *Enable* radio button adjacent to the *Disable CPF* field is selected by default. To disable the use of CPF in the design at run time, select the *Disable* radio button.

   b. To keep the CPF logic enabled for the design, you can specify your choice to display the power-loss values either as `X` or real emulator values. The *Display X or hardware value* field is set to *False* by default, which indicates that you want to

© 2021 All Rights Reserved.

display the actual emulator values. This is equivalent to using the `lp powerLossXDisplay false` command.

To display the power-loss values as `X`, select the *True* radio button. This is equivalent to using the `lp powerLossXDisplay true` command.

c. The *Form SDL text* list boxes allow you to construct SDL statements for a selected CPF object.

To construct SDL statements, perform the following steps:

Select a CPF object type. The list box adjacent to the object type box is automatically populated with a list of objects based on the selected type. Select a CPF object from the list box.

Select a condition type. The list box adjacent to the condition type box is automatically populated with a list of conditions based on the selected type. Select a condition from the list box.

Click the button to construct an SDL statement for the selected CPF object.

6. In the *Get Current Power State* group box, you can perform the following actions:

© 2021
All Rights Reserved.

a. To get the current value of the various CPF objects, click the *<type>* box to display a list and select the required object type.

b. The list box adjacent to the *<type>* box is automatically populated with a list of objects based on the selected type. The corresponding value of the object is displayed in a non-editable text field.

c. In SA mode, you can use the *Get Simulator Power Info on: Power Domain* and *Object* radio buttons to get information on CPF or HDL objects defined with the Xcelium power commands.

7. In the *Low Power Probes* group box, you can perform the following actions:

a. Click the type of CPF object for which you want to create a probe. Four types of CPF objects that can be probed are—*Power Domain*, *Isolation*, *Retention*, and *Power Mode* are common in ICE and SA modes. This is equivalent to using the `probe` commands.

In SA mode, in addition to the above, you can view buttons to probe power domains, HDL objects in CPF expressions, and the retained values of HDL objects defined with the Xcelium power commands.

© 2021 All Rights Reserved.

b. Click the *Upload* button. This is equivalent to using the `database -upload` command.

8. In the *CPF Event Log Control* group box, you can perform the following actions:

a. Specify the *Log File Name*. You can also browse to a specific directory and select an already existing log file by clicking the button next to the text box.

b. Select the type of log you want to create—*Append* (that is, instant log), *Buffered*, or *LA Mode*. Note only one of these check boxes can be selected at a time.

c. If you specify the type of log as *LA Mode,* select either *Earliest*, *Latest*, or *Trigger* to identify the way in which the CPF events should be captured.

d. Click the *Start* button to begin event logging. You also have buttons to *Stop*, *Flush*, *Info*, or *Show* the event log.

For conceptual details, refer to Logging of CPF Events.

9. In the *View Low Power Definition* group box, you can perform the following actions:

a. Click the *Display CPF File* button to view the contents of the CPF files associated to the low-power-aware design.

The *Common Power Format: File* text window illustrated below is displayed. To view and edit a specific file, either enter the name of the required *CPF File* and click

© 2021 All Rights Reserved.

the button or select it using the *Select CPF file* dialog box invoked by clicking the button next to the *CPF File* text box.

Note: The contents of the CPF file displayed in the *Common Power Format: File* text window are editable.

© 2021 All Rights Reserved.

b. To display the CPF data (from a Palladium Z1 emulator point of view) in a tabular format, click the *-->Show CPF Tableview* button. A table as illustrate in the following figure is displayed:

To change the table layout, select or deselect the check boxes next to the CPF objects listed on the right of the table. Then, click the *Redraw* button to bring the changes into affect. The columns in the table will be displayed according to your CPF object selections.

The CPF object name is displayed as a button. Clicking these button enables you to probe the corresponding object. If a CPF object type displayed in the row above the table is grayed out, it means that the design does not contains CPF objects of the disabled type.

Note: After you click the *-->Show CPF Tableview* button, its label changes to *-->Hide CPF Tableview* button. When you click the *-->Hide CPF Tableview* button, the displayed table is closed.

© 2021 All Rights Reserved.

c. To display the CPF data in a standard hierarchical browser format, click the *-->Show CPF Treeview* button. The following figure illustrates the CPF objects of a design arranged hierarchically in a tree view:

Note: After you click the *-->Show CPF Treeview* button, its label changes to *-->Hide CPF Treeview* button. When you click the *-->Hide CPF Treeview* button, the hierarchical tree display is closed.

Design Considerations for Low-Power Verification

Keep the considerations discussed in the following sections in mind when verifying low-power-aware designs using VXE:

- Creating Power Domains
- Using Verilog Registers as Power-Domain Instances
- Defining Power-Domain States
- Resolving Inconsistencies between the Designs
- Handling Power Shut-Off Designs

© 2021 All Rights Reserved.

- Handling Designs with Multiple Supply Voltages (MSV)
- Handling Dynamic Voltage Frequency Scaling (DVFS)
- Verifying Designs Using Bottom-Up Hierarchical CPF Approach with Macro Abstraction

Creating Power Domains

- A power domain is a collection of instances that share the same power distribution network and can be switched on or off simultaneously. The instances can be either module instances in RTL designs and gate-level netlists, or Verilog registers in gate-level netlists.

  An instance can exist in only one power domain. However, a sub-instance can exist in a power domain different from its parent instance.

  In VXE, a register can be modeled as a combinational logic, flip-flop, or memory instance.

- CPF commands support the use of expressions for defining `True` conditions. In VXE, these expressions can only be specified using terminals, not registers. For example, to use the following command, the `<expression>` parameter must specify the terminals:

  ```
  create_power_domain -default -shutoff_condition <expression>
  ```

  If you had the following module, specify the terminals `a` and `c` in the `<expression>` parameter, and ignore `b` because it is a register:

  ```
  module (a,c)
  input a;
  output c;
  reg b;
  endmodule
  ```

  If you specify registers in the `<expression>` parameter, the `readCPFfile` command will generate an error. But if you need to use a register to control a power domain, a workaround is to create a virtual terminal. Use the following CPF commands in a CPF file to create a virtual terminal mapped to the required register:

  ```
  set_instance u_3 -port_mapping {<virtual_terminal> <register>}
  set_design counter -ports <virtual_terminal>
  end_design
  ```

- Enclose the special character “!” (exclamation mark), “&” (ampersand), or “|” (pipe separator) in quotation marks when creating complex expressions. For example:

  ```
  "!power1 & (power2 | power3)"
  ```

- For each power domain, the power can be either *on* or *off*. At run time, signals in the powered-off domains are marked with the `X` value.

© 2021 All Rights Reserved.

- Unlike CPF 1.1 standard, in VXE, a net or pin must be connected to a power domain for waveform purpose. For feedthrough nets and pins, the value of the signal is the same as that of their driver. For example, if a pin is in a powered-on domain, but its driver is in a powered-off domain, the pin value is `X`. Conversely, if a pin is in a powered-off domain, but its driver is in a powered-on domain, it will have a value of 0 or 1. In VXE, a signal's driver is based on its connected pins, and the software checks the input-output direction.

- To properly restore the state of a power domain, the `restore_edge` condition must occur at least `1 FCLK` cycle after the power is restored to that domain. Otherwise, the power domain will not be restored to the correct saved state from the `save_edge` condition.

  In the following example, module `IPBlock` is instantiated as instance `IPInst` in the design `Top`. Instance `IPInst` is part of a switchable power domain `X`.

  ```
  set_design Top
  create_power_domain -name X -instance IPInst -shutoff_condition switch_en \
  -restore_edge restore_en
  set_instance IPInst
  source IPBlock.cpf
  ```

  In this example, all instances of `IPInst` are converted to state-retention flops, using the restore condition specified for domain `X` at the top-level design.

- All the net segments that comprise one physical net should be in the same power domain unless honor boundary ports or isolation is used. Generally, the power domain of a net is the same as that of its driver. If the net connected to a power domain's driver is isolated, the pin's power domain can change. For example, if instance `A` belongs to power domain `PD1`, instance `B` belongs to power domain `PD2`, output pin `A.a` connects to input pin `B.b`. Without isolation, `A.a` and `B.b` are in power domain `PD1`. With isolation between `A.a` and `B.b`, `B.b` is in power domain `PD2` while `A.a` is in `PD1`.

Using Verilog Registers as Power-Domain Instances

In VXE, you can specify Verilog registers as power-domain instances in CPF files.

Use the `-instances <instance_list>` option of the `create_power_domain` and `create_state_retention_rule` commands to specify the RTL-register variable in the instance list.

In RTL designs, the RTL-register variable should be the variable name. If during synthesis, the compiler retains the register names, the register array in the instance list of the `create_power_domain` and `create_state_retention_rule` commands will be recognized. However, if the compiler changes the register names to respective internal names, the register array in the instance list of these two commands will not be recognized.

© 2021 All Rights Reserved.

For example, consider that you have the following register in the RTL design:

```
reg [1023:0] rf[0:15];
```

Here, it is acceptable to specify the following command syntax in the CPF file:

```
create_power_domain -name PDMem -instance {rf }
create_state_retention_rule -name srMem -instance {rf }
```

However, you cannot use the following command syntax:

```
create_power_domain -name PDMem -instance {rf[0] rf[2] }
create_state_retention_rule -name srMem -instance {rf[2] rf [6] }
```

The format of a register name in the RTL design and the corresponding flip-flop or latch instance names in the netlist can be different. If a CPF file is written for an RTL design, use the RTL object names. If a CPF is written for a gate-level netlist, use the gate-level netlist object names.

Defining Power-Domain States

A power domain has three possible states, namely *power-up*, *power-down*, and *standby*.

A power domain in *power-up* state can have different power voltage supply levels at different periods of time. A power regulator will output the different voltage supplies in different conditions. When a pin-related expression is true, the power domain uses the voltage supply in the corresponding nominal condition.

The *standby* and *power-down* states have been designed to save power consumption, where *standby* does not need state retention, but power-down needs state retention. The *standby* state is entered by using the `create_power_domain -active_state_conditions` command.

Note: The *power-down* state saves more leakage power than the *standby* state. Palladium Z1 treats a power domain in the *standby* state as if it were in the *power-up* state.

In VXE, a power domain cannot be in *power-down* state at one time and in *standby* state at another. The *standby* state is specified with the `-active_state_conditions` option to the `create_power_domain` command. The *power-up* state should also be specified with the `active_state_conditions` option when *standby* is used. As *standby* and *power-down* state should not be mixed, there should be no `-shutoff_conditions` when `-active_state_conditions` are used.

A power domain can be in either of the following *combination of states*:

- *Power-down* or *power-up*

© 2021 All Rights Reserved.

Before power is off, sequential elements in the power domain should be saved; and after the power is back, sequential elements will be restored. During *power-down*, the interface from the powered-down domain to other powered-up domain should be isolated.

The `-power_up_states` and `-power_down_states` options of the `create_power_domain` command enable specification of the values for the sequential elements (such as registers modeled as flip-flops or memory instances) on power-up and power-down. These options accept the values `low`, `high`, `random`, or `inverted`. If these options are not provided with the `create_power_domain` command, by default, the `-power_up_states` option is set to `random` and the `-power_down_states` option is set to `low`.

Note: The upload and download of memories is not affected by their power status. If the `precompileOption -add randomizeMemories` command is specified at compile time, the `-power_down_states` and `-power_up_states` options allow the CPF memories to mimic the behavior of the *real* memories when they are powered down and/or up. For detailed information, refer to Handling of CPF Memory Power Up or Down Behavior.

Handling of CPF Memory Power Up or Down Behavior

To cause the memories to behave in a way that mimics the behavior of *real* memories when they are powered up and/or down, an emulated circuitry is added to the inputs and outputs of the memory that responds to the power-condition signal for the enclosing block. When the signal returns that the block is powered-down or powered-up, the additional circuitry causes the outputs (memory reads) to behave according to the specified value of the `-power_up_states` and `-power_down_states` options. It preserves the ability to upload or download the memories, but the effects of uploading or downloading are affected by the power state of the volatile and non-volatile memories.

Note: A memory is considered to be volatile unless it is identified as a ROM that lacks the circuitry for writing to the memory.

On a power-down event, both volatile and non-volatile memories behave as follows, depending upon the value provided:

`high`	The output of a memory is high when power is off. Writes to the memory are disabled and reads always return all ones.
`low`	The output of a memory is low when power is off. Writes to the memory are disabled and reads always return all zeros.

© 2021 All Rights Reserved.

`random`	The content of a memory is randomized on power-down. Writes to the memory are disabled and reads return randomized data.
`inverted`	The content of a memory is inverted on power-down. Writes to the memory are disabled and reads return inverted data.

On a power-up event, volatile memories behave as follows, depending upon the value provided:

`high`	Same as random
`low`	Same as random
`random`	The content of a volatile memory is randomized on power-up. Then, the addresses might be read and written normally. Addresses that are read before being written, return random data.
`inverted`	The content of a volatile memory is inverted on power-up. Then, the addresses might be read and written normally. Addresses that are read before being written, return the inverse of the content.

Whether poweredup or powereddown, you can access volatile memories through the debugging tool. However, the existing content of a randomized volatile memory appears as random, and an inverted volatile memory appears as inverted. Also, the content loaded into a randomized or an inverted volatile memory during power-down state appears as random or inverted to the circuit.

On a power-up event, the content of a non-volatile memory is restored to its state before power-down, regardless of the value provided. When powered-down, the content of the non-volatile memory is modified according to its power option, and it can be downloaded to the xeDebug tool. A non-volatile memory is the same as ROM, it retains its value during power-down.

If a memory identified as a retain-state element is within a power domain that has up or down states specified, its contents are preserved when state-retention is active. If the power state of the power domain changes, the memory responds according to the specifications of the domain (`high`, `low`, `random`, or `inverted`). When state-retention state is restored, the memory reverts to the way it was when the retain-state began; the content appears to be unchanged by any intervening power up or down events.

© 2021 All Rights Reserved.

Enabling Power State Handling of CPF Memories Manually

The mechanism of handling the CPF memories based on the defined power states is enabled by default. However, if you had disabled the feature earlier, enable it again manually by giving the following command along with other User Data commands either in the XEL script or on the command line at compile time:

```
precompileOption -add randomizeMemories
```

Disabling Default Power State Handling Mechanism of CPF Memories

To disable the default mechanism for handling the CPF memories based on the defined power states, give the following command along with other User Data commands either in the XEL script or on the command line at compile time:

```
precompileOption -add dontRandomizeMemories
```

When such memories are modeled in the Palladium Z1 emulator, the power never really goes off. Instead, the power-down state is modeled according to the behavior specified by the CPF specification and the memories are allowed to retain their state, regardless of the power status of the enclosing block. This is at variance with the real-world behavior.

Resolving Inconsistencies between the Designs

In emulator systems like Palladium Z1, both the testbench and DUT parts of the design are considered. However, Joules, a Cadence power analysis tool, considers only the DUT part of the design. Therefore, to share the same CPF files between the Palladium Z1 and Joules tools, first resolve the inconsistencies between these two definitions of the design.

If the `set_design` CPF command in a CPF file uses a non-top cell name, it means the CPF file contains only the DUT part. So, to run such a design on Palladium Z1, iterate the current top cell to find out the instance, which belongs to the non-top cell name. Then, use that instance for the scope of the current CPF objects.

For example, consider a design top cell named `testbench`, a DUT module named `DUT`, and a DUT instance named `dut`. The CPF file named `dut.cpf` refers to the top-level module `DUT`, and specifies a power domain `pd1` with instances `A` and `B`. The `readCPFfile dut.cpf` command will translate the power domain `pd1` as `pd1` and its instances as `dut.A` and `dut.B`, instead of just `A` and `B`.

This solution chooses one instance that belongs to the CPF top design module because Palladium Z1 does not support multiple top-level modules. If the current design-top cell has multiple CPF top-level module instances, the `readCPFfile` command will fail with an error message.

© 2021 All Rights Reserved.

Handling Power Shut-Off Designs

A design with Power Shut-off (PSO) conditions contains portions that can be switched on or off to save both leakage and dynamic power. PSO is one of the most effective low-power techniques.

There are three types of PSO behaviors: power switching, state retention, and interface isolation. In Palladium emulators, instrumentation logic is used to mimic these three PSO behaviors. This model might increase design capacity, but the emulation speed is not affected. The change depends on the number of sequential elements and isolated pins.

When a power domain is off, all instances, pins, and ports, nets of the power domain are in a power-loss state (also called corrupted state). Before the power-down, sequential element states in the power domain will be saved; after the power is back, the sequential element states will be restored.

The pins of a powered-down domain should be isolated so that other power domains (which can be still on) are not affected. The isolation should be done before the power-down happens. Remove the isolation once the power is back. VXE mimics this PSO behavior.

However, the state retention behavior of a PSO design needs to be handled differently. For details, refer to Handling State Retention PSO Behavior.

As emulation systems are two-logic systems, `0` or `1`, the `-power_down_state` and `-power_up_state` options are used to specify power-loss values and power back initialization values; thus, mimicking the power switching behavior in the design.

In a power-aware design, the PSO intention is given in the CPF file instead of the HDL files. During compilation, the PSO specifications are read from the CPF files. At run time, you can view the power domain states, power modes, occurrence of state retention, and values of the isolation pins in the waveforms. To view the powered-down state, probe the power domain using the `probe -addPowerDomain <domainNameList>` command. The value of a signal in power-loss state is shown as `X` in the waveform.

The simplest PSO use model involves specification of PSO conditions without nominal condition. In this case, the defined PSO behaviors are modeled in the instrumentation logic, but power mode is not displayed in the waveform. The other PSO use model involves specification of PSO conditions with nominal conditions, but without power mode definitions. Both these use models are not recommended when you want to use power mode to analyze the power intent.

Handling State Retention PSO Behavior

A PSO design supports the following three kinds of state-retention cells:

© 2021 All Rights Reserved.

- Clock Low Retention Flip-Flop (CLRFF)
- Clock High Retention Flip-Flop (CHRFF)
- Clock Free Retention Flip-Flop (CFRFF)

By default, VXE only supports the CFRFF kind of state-retention cells that retain the value of the flip-flop during power shut-off only if the flip-flop is in retention mode. Such flip-flops are not dependent on the clock.

To specify a state-retention cell of type other than CFRFF, execute the following command:

```
stateRetentionControl -add {<stateRetentionRuleName> low|high}
```

Specifying the value as `low` in the `stateRetentionControl` VXE command indicates use of CLRFF state-retention cells. Such state-retention cells retain the value on power down only when the retention condition is `true`, and the clock is `LOW` while the flip-flop is going to retention mode.

Specifying the value as `high` in the `stateRetentionControl` VXE command indicates use of CHRFF retention cells. Such state-retention cells retain the value on power down only when the retention condition is `true`, and the clock is `HIGH` while the flip-flop is going to retention mode.

For each `create state retention rule` CPF command, you must execute a `stateRetentionControl` VXE command to specify the corresponding flip-flop type which will be applied to that retention rule. If a value is not set using the `stateRetentionControl` VXE command, then CFRFF is used by default.

Debugging a PSO Power-Aware Netlist

To read PSO specifications for a netlist generated using a tool other than Palladium Z1, generate the power-domain information based on the top-level signal connected to the power supply pins of each instance using the following VXE command:

```
identify power domain {[-name <powerDomainName>] [-modules <moduleNameList>]
                       [-shutoff_condition <expression>]}
```

The power-control net is specified in the PSO condition.

If an instance belongs to a module existing in the specified `<moduleNameList>` and its input pin is connected to the power-control net, that instance is considered as a part of the specified power domain.

If an instance is not within the specified power domain, the instance will be always in power-up state and no instrumentation logic will be added during compilation for PSO operations.

© 2021 All Rights Reserved.

Handling Designs with Multiple Supply Voltages (MSV)

A pure MSV design without *standby* power domains has the following characteristics:

- It uses multiple supply voltages for the core logic, but no other low-power techniques like PSO and DVFS.
- It does not need any instrumentation logic.
- It has only one power mode and each power domain can be associated with only one nominal condition.

To check the MSV level shifter rules, use CLP integrator. VXE does not support level shifter rule checking directly.

A pure MSV design has only one state of design (power mode).

At run time, the downloaded emulator executable image includes the power supply information, such as power domains, nominal condition, instances list, and pin/port list in each power domain.

Handling Dynamic Voltage Frequency Scaling (DVFS)

In VXE, a power domain of DVFS type can be in either *standby* or *power-up* state. However, a power domain cannot have a mix of *powered-down* and *standby* states.

At compile time, the DVFS specifications within the CPF files are checked. Consequently, the PSO instrumentation logic is added to mimic the power switching, state retention, and isolation behaviors. The `-active_state_conditions` option is used to specify in which condition the given power domain will be in which nominal condition.

At run time, the emulator executable image for DVFS designs includes the power mode transition and power domain signal values, that is, *on*, *off*, and *standby*.

Handling Hierarchical CPF Files

As you proceed through the design process, you can add more content to your CPF file. This can be done by adding the additional commands either directly to the original CPF file or to a separate file that is then sourced into the original CPF file.

A CPF file is called a hierarchical CPF file if it includes any of the following CPF commands:

- The `set_instance` command with `-design <power_design_name>` option

© 2021 All Rights Reserved.

Or,

a. The `set_instance <instance>` command

Followed by

b. The `set_design <power_design_name>` command

For information on the usage of flat- and hierarchical-CPF files, refer to the *Process of Creating the CPF Content* chapter in the *Common Power Format User Guide*, available on the *Cadence Online Support* website.

In a hierarchical design, the portion of design that is visible and might be affected by a CPF command is known as its scope. The scope can be applied in either of the following two ways:

- From the top level of the design (referred to as the top-level power domain) to the root (referred to as the block-level power domain)

  OR

- From the root of the design to the top level of the design

Note: The top level generally is a scope-sensitive term that refers to the topmost level of the hierarchy of a scope. It usually does not refer to the topmost design of the chip or entire design hierarchy, unless explicitly stated.

For information about the type of low-power techniques you can use for hierarchical-CPF files, refer to Table 33-2 and Table 33-3.

Verifying Designs Using Top-Down Hierarchical-CPF Approach

In top-down hierarchical-CPF approach, the power intent is clear only in the top-level power domain. The power intent of block-level power domains is derived from the top-level power domain through power domain mapping. Mapping of two power domains results in merging a power domain of a block with another power domain of the top-level design in which the block is instantiated.

For more information on power domain mapping, refer to the *Hierarchical Flow* chapter in the *Common Power Format Language Reference*, available on the *Cadence Online Support* website.

On the Palladium Z1 emulator, after domain mapping in the top-down hierarchical-CPF approach, the power domain settings are resolved in the following manner:

- The domain attributes are impacted as follows:

© 2021 All Rights Reserved.

- If you specify the following options with the `create power domain` CPF command, the state of the top-level power domain takes precedence over the state of the block-level power domain:
  - `-power_up_states`
  - `-power_down_states`
- If you specify the `-boundary_ports` option with the `create_power_domain` CPF command, the ports/pins are merged into the top-level power domain definition.

  For information on processing of top- and block-level power domains based on options specified with the `create_power_domain` command, refer to Handling of Power-Domain Attributes After Domain Mapping.

- The state retention rules are applied as follows:
  - The state retention rules of the block-level power domain apply only to the block-level instances.
  - The state retention rules of the top-level power domain apply to the whole design.
  - If both the top- and block-level state retention rules apply to the same design objects, the block-level state retention rules take precedence.
  - A block-level state retention rule can be optimized if its power domain or instances are in an *always-on* power domain.

    For information on processing of top- and block-level state retention rules based on options specified with the `create_state_retention_rule` command, refer to Handling of State Retention Attributes After Domain Mapping.

- The isolation rules are applied as follows:
  - The isolation rules of the block-level power domain apply only to block-level instances.
  - The isolation rules of the block-level power domain apply only to the corresponding original block-level power domain, but not the merged power domain.
  - The isolation rules of the top-level power domain apply to the whole design. The top-level isolation rules might be optimized on merged power domain boundaries.
  - If both top- and block-level isolation rules apply to the same instance, the block-level isolation rules take precedence.

© 2021 All Rights Reserved.

For information on processing of top- and block-level isolation rules based on options specified with the `create_isolation_rule` command, refer to Handling of Isolation Attributes After Domain Mapping.

- Only top-level power modes are used; the block-level power modes are ignored.

  If a block-level power domain is not mapped to a top-level power domain, it is promoted into the top-level power domain. The top-level power mode might either include this new power domain directly, or treat it as a *power-off* state by default. For details, refer to *Handling Power Modes after Domain Mapping* in the *Hierarchical Flow* chapter of the *Common Power Format Language Reference*, available on the *Cadence Online Support* website.

Handling of Power-Domain Attributes After Domain Mapping

Consider that you have a top-level power domain, *`PdTop`* and a block-level power domain, *`PdBlock`*. You mapped *`PdBlock`* to *`PdTop`* by using the following command:

```
set_instance -domain_mapping {PdBlock PdTop}
```

Then, in *`PdBlock`*, you can specify the different power-domain attributes by using the `create_power_domain` command options specified in the following table:

Use the create_power_domain command option	To...
`-name`	Merge the content of `PdBlock` to `PdTop`.
`-instances`	Merge the instances in `PdBlock` to `PdTop`.
`-boundary_ports`	Merge the boundary ports that were defined using the `set_design -boundary_ports` command in `PdBlock` to `PdTop`.
`-default`	Merge the instances and scope defined in `PdBlock` to `PdTop`.
`-shutoff_condition`	Map the shut-off condition of `PdBlock` to a switchable `PdTop`. **Note:** `PdBlock` and `PdTop` should have the same shut-off condition. If not, the shut-off condition defined in `PdTop` is used and a warning is displayed.

© 2021 All Rights Reserved.

Use the create_power_domain command option	To...
`-external_controlled_shutoff`	Determine the power switch in `PdBlock` by using the shut-off condition defined in `PdTop`. **Note:** The `-external_controlled_shutoff` defined in `PdBlock` is ignored.
`-default_isolation_condition`	Use the default isolation condition defined in `PdTop`. **Note:** ■ The `-default_isolation_condition` defined in `PdBlock` is ignored. ■ If the `create_isolation_rule` command of `PdBlock` does not define an isolation condition, the `-default_isolation_condition` defined in `PdBlock` is used for that isolation rule.
`-default_restore_edge` | `-default_save_edge`	Use the `-default_restore_edge` or `-default_save_edge` definitions from `PdTop`. **Note:** ■ The `-default_restore_edge` or `-default_save_edge` defined in `PdBlock` is ignored. ■ If the `create_state_retention_rule` command in `PdBlock` does not define a save or restore condition, the `-default_restore_edge` or `-default_save_edge` defined in `PdBlock` is used for that state retention rule.
`-power_up_states` | `-power_down_states`	Use the definitions from `PdTop`. **Note:** The power up or down states defined in `PdBlock` are ignored.

© 2021 All Rights Reserved.

Use the create_power_domain command option	To...
`-active_state_conditions`	Use definitions from `PdTop`. **Note:** The `-active_state_conditions` defined in `PdBlock` are ignored. A warning is displayed if all `-active_state_conditions` are not present in `PdBlock`.
`-base_domains`	Use the base domains of `PdTop`. **Note:** ■ `PdTop` can have more base domains than `PdBlock`. ■ Each base domain of `PdBlock` should be mapped to one of the base domains of `PdTop`. A warning will be given if there is no such mapping. ■ If `PdTop` does not have base domains, the base domains of `PdBlock` are ignored.

A block-level power domain that is not mapped to the top-level power domain is named as `<scope>.<PdBlock>`. It is, then, treated as a special top-level power domain – its scope is in top-level, but it cannot be used for domain mapping as a regular top-level power domain.

All the isolation and state retention rules related to this power domain use the name `<scope>.<PdBlock>` to reference it.

Handling of State Retention Attributes After Domain Mapping

The state retention rules defined in the top-level power domain (such as `PdTop`) are processed as per definition. However, the state retention rules defined in the block-level

© 2021 All Rights Reserved.

power domain (such as `PdBlock`) are processed on the basis of the `create_state_retention_rule` command options specified in the following table:

Use the create_state_retention_rule command option	**To...**
`-name`	Define the block-level state retention rule with a name of the format `<scope>.<rule_name>`. Here, the `<scope>` is saved and passed to precompile to indicate the scope of the specific rule.
`-domain`	Define the block-level power domain in the format `<scope>.<power_domain_name>`. **Note:** If the specified block-level power domain is mapped to the top-level power domain, it is not listed in the power mode list; however, it effects the state retention rule.
`-instances` | `-exclude` | `-save_edge` | `-restore_edge` | `-restore_precondition` | `-save_precondition`	Use the block-level power domain settings defined for each of these options.
`-secondary_domain`	Define the secondary power domain in the format `<scope>.<PdBlock>.<base>`. **Note:** ■ Use the name of the secondary domain associated with the top-level power domain if the block-level power domain is mapped to a top-level power domain. ■ Use the secondary power domain of the specified base domain if `-secondary_domain` is not specified. A warning is given if the block-level power domain has more than one base domain. ■ Display a warning if `-secondary_domain` is not specified but the `-instances` option is.

© 2021 All Rights Reserved.

Handling of Isolation Attributes After Domain Mapping

The isolation rules defined in the top-level power domain (such as `PdTop`) are processed as per definition. However, the isolation rules defined in the block-level power domain (such as `PdBlock`) are processed on the basis of the `create isolation rule` command options specified in the following table:

Use create_isolation_rule command option	To...
`-name`	Define the block-level isolation rule with a name of the format *<scope>.<isolation_rule_name>*.
`-isolation_condition | -pins | -exclude`	Use the block-level power domain settings defined for each of these options.
`-from -to -pins -exclude`	Select pins and power domains within the scope of the block.
`-isolation_output`	Use the block-level power domain setting defined for this option. **Note:** Tristate is not supported in block-level power domain.
`-secondary_domain`	Use the name of the secondary power domain originally specified in the block if not mapped to any top-level power domain. **Note:** ■ Use the name of the secondary domain associated with the top-level power domain if the block-level power domain is mapped to a top-level power domain. ■ Use the secondary power domain of the specified base domain if `-secondary_domain` is not specified. A warning is given if the block-level power domain has more than one base domain.

© 2021 All Rights Reserved.

Verifying Designs Using Bottom-Up Hierarchical CPF Approach with Macro Abstraction

Palladium Z1 emulator supports three types of IPs—hard non-custom IPs, hard-custom IPs, and soft IPs. Hard non-custom IPs and soft IPs can allow changes. However, hard-custom IPs do not allow changes, and cannot have logical and physical views.

In both ICE and SA modes, the power behaviors for hard-custom IPs can be described using macro CPF modules written in either Verilog or VHDL. However, in SA mode, if the simulation model did not have the macro CPF power intention, the internal macro cell power intention cannot be emulated correctly.

Modeling Hard-Custom IP Low-Power Intention in a CPF File

In Palladium Z1 emulator, to model a low-power-aware hard-custom IP, the CPF file must contain CPF commands that define the following parts:

- Testbench
- Top-Level Module of DUT
- Macro CPF Module

Testbench

Use the `set_design -testbench` command to specify the top-level module.

The CPF commands to create or define the power domains, state retention rules, isolation rules, power modes, and nominal conditions are specified in the top-level module of the DUT part of the CPF file, not in the testbench. For details, refer to Top-Level Module of DUT.

In the testbench portion of the power-aware hard-custom IP, you can specify only the following CPF commands and options:

```
set_cpf_version <2.0 | 1.1>
set_hierarchy_separator <character>
set_design <instance> -testbench
set_instance <instance>
include <Top_DUV_CPF_filename>
end_design <power_design_name>
```

The `set_instance <instance>` command can be used only to change the scope.

Note: Power intentions specified in the testbench CPF are ignored with a warning message. Power domains and power modes are not allowed in the testbench CPF.

© 2021 All Rights Reserved.

Top-Level Module of DUT

In the top-level module of the DUT part of the CPF file, you must specify CPF commands to create the power domains, state retention rules, and isolation rules; and define the power modes and nominal conditions. In addition, you can specify CPF commands to update the isolation rules.

In the top-level module of the DUT, you can specify only the following commands and options:

```
set_cpf_version <2.0 | 1.1>
set_design <module>
set_hierarchy_separator <character>
create_power_domain -name -default -instances -shutoff_condition
                    -power_up_states -power_down_states
                    -active_state_conditions -default_isolation_condition
                    -default_save_edge_condition
                    -default_restore_edge_condition -base_domains
create_isolation_rule -name -from -to -pins -isolation_output
                      -isolation_condition -secondary_domain -exclude
create_state_retention_rule -name -domain -instances -exclude -restore_edge
                            -save_edge -save_precondition
                            -restore_precondition -secondary_domain
set_instance <instance> -domain_mapping -model
include <CPF_filename>
create_nominal_condition -name -voltage -state
create_power_mode -name -domain_conditions
create_mode_transition -name {-start -to}
get_parameter <parameter_name>
find_design_objects <pattern> [-pattern_type {name | cell | module}] \
                    [-scope][-object][-direction][-leaf_only | -non_leaf_only] \
                    [-hierarchical][-exact | -regexp][-ignore_case]
end_design <power_design_name>
```

Use the `set_design <module>` command to assign CPF description to the specified module.

Note: Palladium Z1 emulator supports specification of an instance name in the first `set_design` command of the top-level module of the DUT if it has not been already specified in the testbench. In addition, all commands supported in the top-level module of the DUT part of the CPF file can be specified in block-level CPF command files.

Use the following command to define a macro model:

```
set_instance <instance> -domain_mapping -model
```

Consequently, the macro CPF is loaded and the power domains are mapped.

If the macro CPF module has been defined in a separate CPF file, specify the name of that file in the top-level module of the DUT by using the `include <CPF_filename>` command.

© 2021 All Rights Reserved.

Macro CPF Module

Use a macro CPF module to define the power domains, power modes, and nominal conditions for the hard-custom IP. It is also used to create state retention and isolation rules.

A macro CPF can contain only the following commands and options:

```
set_macro_model <macro>
create_power_domain -name -default -instances -boundary_ports \
                    -shutoff_condition -power_up_states -power_domain_states \
                    -active_state_conditon -default_isolation_condition \
                    -default_restore_edge_condition \
                    -default_save_edge_condition -base_domains
create_isolation_rule -name -from -to -pins -isolation_output_condition \
                      -exclude -secondary_power_domain
create_state_retention_rule -name -domain -instances -exclude -restore_edge \
                            -save_edge -save_precondition -restore_precondition \
                            -secondary_domain
create_nominal_condition -name -voltage -state
create_power_mode -name -domain_conditions
end_macro_model <macro>
```

Handling Hard-Custom IP Behavioral Models on Palladium Z1 Emulator

Palladium Z1 emulator expects that the behavioral model of the hard-custom IP has power-aware behaviors. Palladium Z1 emulator handles a low-power-aware hard-custom IP as a blackbox. The macro CPF module for such hard-custom IP should describe its implementation. Instrumentation logic for power corruption, state retention, and isolation of registers is not required. However, instrumentation logic for power corruption of the boundary ports can be modeled after domain mapping.

At run time, if the upper-level power domain mapped to the hard-custom IP is off, the IP's boundary ports and the output of the registers are displayed in power-loss state.

Mapping Power Domains of Macro CPF Modules

The macro CPF domain mapping process on Palladium Z1 emulator deviates from the CPF standard because of the following behavioral conflicts:

- When a macro-level switchable domain is mapped to a top-level switchable domain, the top-level shut-off condition is used and there is no structural equivalency checking.
- The `-active_state_conditions` of macro-level mapped power domains are ignored with warning messages. Instead, the top-level `-active_state_conditions` are applied to the macro-level mapped power domains.
- In macro CPF module, registers can be specified with the `create power domain` CPF command. Palladium Z1 emulator can synthesize a register as a memory instance, a flip-

© 2021 All Rights Reserved.

flop or latch, or a wire connecting to an input or output of a combinational logic. Registers that appear in a power domain are recognized only when their names are reserved by the compiler.

- After domain mapping, macro-level power modes are updated to use top-level domain conditions. Palladium Z1 emulator does not conduct a top-level versus macro-level power mode consistency check. Palladium Z1 emulator supports probing of top-level power modes during debug time.

Logging of CPF Events

While a design is running in ICE or SA mode, you might want to see a message when a particular CPF event happens. Palladium Z1 emulator supports logging of messages related to the power mode transitions that happen during a design run and the power domain status. This section discusses the types of logs that enable recording of the messages, and the format of the logged messages.

Note: When the design is re-compiled, the CPF-event log database generated in the previous design run is overwritten by new log data. Therefore, in both online and offline sessions, the CPF-event log of the previous runs becomes inaccessible.

Types of CPF-Event Logs

The following two types of logs can be used to track the occurence of CPF events in a design run:

- Instant Log
- Buffered Log

Instant Log

An instant log enables you to view a message related to a CPF event as soon as it happens. Such type of logging can be done only in STB and SA modes that allow you to pause the design run when required. Whenever a CPF event happens, the design run is paused, the message is printed on the screen, and then the design run is resumed. As an instant log needs to pause the design run, it cannot be used in the LA mode in which the design run cannot be stopped.

© 2021 All Rights Reserved.

Important

You can dynamically change between the LA mode and the STB mode using the `configPM` command. However, when you change the mode, the current CPF event logging will be stopped.

To use an instant log, execute the following command before giving the `run` command to start running the design:

```
lp log -start [<log_file_name> [-append]]
```

If you specify a log file name, the messages will also be saved in a log file, which you can study later. You can save the log file in either the current directory or any other directory. If you specify to save the log file in a directory other than the current one, you should provide the correct path with the log file name. If you specify the `-append` option, the log of new CPF events is appended into the existing log file.

If you execute the `lp log -start` command again, an error message is generated to indicate that a log session is already running and suggest that you should first stop the session.

You can use the `lp log -stop` command to stop the CPF-event log. If you exit from the xeDebug tool without stopping the event log, the VXE software automatically uploads the remaining logs and saves those into the log file.

The log file is saved in binary format. Therefore, you need to execute the `lp log -show` command to decrypt the binary data and get the CPF event log messages. For detailed information, refer to Show CPF-Events Log Messages from a Log File.

The instant log impacts the run-time performance if there are too many messages to log. This is because the pausing and resuming of the design process incurs some overhead. If performance is a critical requirement and you do not need to see the logging message right away, use the buffered logs instead.

Buffered Log

Palladium Z1 uses a memory buffer to capture the CPF events. The size of the memory buffer is limited. You can set the depth of the memory buffer using the following command at compile time:

```
compilerOption -add {cpfLogBufferDepth <n>}
```

Note: The feature of generating buffered logs can be disabled by setting the value of the `cpfLogBufferDepth` parameter to `0`.

© 2021 All Rights Reserved.

In STB and SA mode, whenever the memory buffer is full, the design run is paused to upload the memory buffer and then resume design run. In the LA mode, the emulation cannot be paused. Therefore, there is a different usage model which is described later in Buffered Log in LA Mode.

Use the `lp log -start -buffered <log_file_name>` command to start using a buffered log. Then, start the design run. The logged data is automatically saved to the specified log file while the design is running. After executing a `run` command, the memory buffer might contain data that has not been uploaded. You can execute the `lp log -flush` command to force the data to be uploaded. If you use the `lp log -stop` command to stop the logging, the memory buffer is also flushed.

If you execute the `lp log -start` command again, an error message is generated to indicate that a log session is already running and suggest that you should first stop the session.

When using a buffered log, you will not see any logging message while the design is running. Use the `lp log -show` command to see the logging messages. For detailed information, refer to Show CPF-Events Log Messages from a Log File.

Buffered Log in LA Mode

In the LA mode, the design run cannot be stopped. Therefore, the maximum number of CPF events that can be logged is limited by the depth of the memory buffer. There are following three different strategies to control the logging of the CPF events:

- Earliest

  The CPF event log is stopped once the memory buffer is full. Therefore, the earliest events are logged in the memory buffer.

- Latest

  The CPF event log never stops. When the memory buffer is full, the latest event overwrites the earliest event.

- Trigger

  The CPF event log is stopped once a trigger happens. This is the most preferred strategy because it allows you to get a set of CPF events before each trigger occurs.

Use the following command to start CPF event logging in LA mode:

```
lp log -start -lamode { earliest | latest | trigger } <log_file_name> \
       [-posttrigger <num>]
```


© 2021 All Rights Reserved.

The `-lamode` option indicates that the buffered log is for LA mode. You can execute the `lp log -stop` command to stop the log. Using the `lp log -flush` command uploads the memory buffer and then automatically stops the current log session.

Format of CPF-Events Log Messages

The CPF-events log messages include the following:

- Time when the event happened
- Current power modes
- Current illegal power modes

  Note: An illegal power mode is defined in the CPF file to indicate the power mode state in which the design should never enter.

- Status change for power domains

Design could be in multiple power modes. Following is an example of logging message:

```
INFO CPFLOG: At cycle 1299 design enter power mode M1 and M3. Design enter illegal
power mode M5. Power domain D2 changed from Off to On. Power domain D4 changed from
On to Standby.
```

In SA mode, the time is shown as simulation time instead of cycle, such as `At 1022ns`. The logged messages only contain power domains for which the status changed. The messages are also saved in the `xe.msg` file whenever it is printed on the screen. The message is prefixed with text string `INFO CPFLOG:` so that you can identifiy the messages from `xe.msg` file.

Show CPF-Events Log Messages from a Log File

The log file is saved in binary format. So, to list the messages from the log file, you can execute the `lp log -show <log_file_name>` command. This command shows all messages in the log file.

You can also specify a time range as follows to show only the messages logged during the specified time range:

```
lp log -show [-start <time>] [-end <time>] [<log_file_name>] [-outfile
<out_file_name>]
```

You can execute the following command to get basic information of a log file, such as the time range for logged messages and number of logged messages:

```
lp log -info <log_file_name>
```


© 2021 All Rights Reserved.

If you execute the `lp log -show` command without a log file name, the currently open log file is used, that is, the one specified with the `lp log -start` command. If you do not specify a log file name and there is no open log file, for instance, if you have executed `lp log -stop` command to stop the CPF-event log, an error message is displayed.

© 2021 All Rights Reserved.

Using CPF in SA Mode

This section describes the support for power verification using Common Power Format (CPF) in Simulation Acceleration mode when the designs are compiled using the IXCOM compiler.

CPF Objects Supported in SA Mode

- Power Domains - contain instances that will be powered down dependent on a power shut-off condition.
- Isolation Cells - continue to drive the outputs of power-ed down domains.
- State Retention Cells - can save and restore logic states before and after shut-off events.

CPF File Content in SA Mode

In addition, the CPF file can define the following conditions:

- Nominal Conditions - specify each supply voltage, and optionally the ground and bias voltages.
- Power Modes - define the nominal condition for each power domain in each given power mode.

Support for CPF Features in SA Mode

CPF Feature	Supported in ICE Mode	Supported in SA Mode
PSO, with isolation and/or retention	Yes	Yes
Multi-Voltage Support (MVS) with PSO	Yes	Yes
Macrocell CPF	Yes	Yes
RTL netlist with RTL CPF	Yes	Yes
Hierarchical (top-down or bottom-up) CPF	Yes	No
Gate-level netlist with gate-level CPF	Yes	No
Standby	No	No
Dynamic Voltage Frequency Scaling (DVFS)	No	No

© 2021 All Rights Reserved.

CPF Feature	Supported in ICE Mode	Supported in SA Mode
Mode transitions based on `create_mode_transition` conditions that do not agree with or are in addition to the `create_power_domain` conditions	No	No

Compilation - CPF Constraints - Partitioning

The testbench must not contain any CPF objects that do not exist in the DUT. Else, the backend compiler will report an error stating that the object was not found. The split between the hardware (swap-able) and software portions of the design must follow power domain boundaries - no power domain should contain instances from both portions.

Power control signals might exist in the software portion and be used to control power domains in the hardware. These signals are created in hardware with the appropriate hierarchy and connected to the software signals as primary inputs.

Power control signals might also exist in the hardware in always-on power domains.

Support for CPF at Compile Time by IXCOM Compiler

To enable CPF support in SA mode, use the `-lps_cpf <file>` option with the `ixcom` command and pass the CPF file with it at the time of compiling the design for SA mode using IXCOM.

Note: When using the *xrun* tool to compile the design for SA mode, of all the Xcelium low power switches, only the `-lps_cpf <file>` option is passed to IXCOM. Other options in the command-line are either ignored or directly passed to the Xcelium tools as documented in the Compiling Designs section of Chapter 25, "Compiling and Running Simulation Acceleration Designs with *xrun*."

Specifying the `<cpf_file>` option allows both the Palladium Z1 backend compiler to add CPF instrumentation to the hardware compile and also xmelab to add CPF instrumentation to the software models. An alternate method would be to specify the CPF file separately to the backend compiler through `compileOptions.qel` and to xmelab through `uxc_hdl.var`.

© 2021 All Rights Reserved.

Reading-in of CPF Data

In ICE mode, the XEL compiler command `readCPFfile <cpf_file>` is used to process the CPF file. In SA mode, IXCOM passes the file specified with the `-lps_cpf <cpf_file>` option of the `ixcom` command to the backend compilers to internally call the `readCPFfile` command. While compiling the CPF designs in SA mode, use the `ixcom -xecompileargs -lps_cpf <fileName>` command.

In addition to the `-lps_cpf <cpf_file>` option, you need to specify the precompile options, as specified in ICE mode, by using the `compilerOptions.qel` file.

When you use the `-lps_cpf <cpf_file>` option, you need to also specify the `+use_frcrel_po` option if the primary outputs in the CPF design are driven from a switchable power domain. If the `+use_frcrel_po` option is not specified, the primary outputs might retain `X` values, for example, on waveform, after power back until the values are toggled again.

By default, in both SA and ICE modes, if a design element existing in the CPF file cannot be found in the Palladium database, the compilation continues with a warning message.

However, you can force the compiler to generate an error message and stop the compilation by using the `-lps_cpf+exitOnError` option with the `ixcom -xecompileargs` command in the SA mode, as follows:

```
ixcom -xecompileargs -lps_cpf+exitOnError <fileName>
```

Note: The `ixcom -xecompileargs -lps_cpf+force <fileName>` command is deprecated and not supported in the SA mode.

In the ICE mode, execute the `readCPFfile -exitOnError <cpfFileName>` command to view the error message about the design elements not found in the Palladium database.

Support for CPF at Run Time in SA Mode

The following sections describe the run-time functionality available with the xeDebug tool, and the *xrun* tool when the `xrun` command is invoked with the `-xeDebug` option as following:

```
xrun -hw -ua -lps_<cpf_file> -xeDebug
```

Hot Swap

You can swap in the design to hardware at any time. On the hardware, the CPF-control logic works in the same way as it works in ICE mode. If any of the power domains are powered down at this time, these will remain in powered-down state even after being swapped in to the emulator.

© 2021 All Rights Reserved.

When you swap into the hardware, the CPF instrumentation in the hardware is disabled until the swap is completed.

Swapping out from the hardware and back to the software is not supported with CPF designs.

All run modes except match are supported, including co-simulation, tbrun, autorun, autorun with dynamic target, and nbrun. The match mode is not supported because the CPF implementation on Palladium Z1 is a little different from that on Xcelium.

All CPF-related run-time commands available with ICE mode are available with SA mode.

The following CPF-related run-time commands are available with Xcelium, and available on Palladium Z1 in SA mode:

- `describe -power`

  Returns static CPF information that will not change when the design is swapped into the hardware mode.

- `power <object>`

  Returns the current power state information on HDL objects, power domains, and power modes. This option is not applicable on the design running on the hardware.

Additional CPF functionality is provided by Xcelium as switches to the `probe` command discussed in the *Run-time Commands* chapter of the *VXE Command Reference Manual*.

As in ICE mode, the `value` command returns the value `X` for signals in power-down states.

The `lp -value [-<lp_object_type>] <lp_object_name>` command returns the values set for the CPF-specific objects. The CPF object type can be any of the following:

- ❑ powerDomain
- ❑ retention
- ❑ isolation
- ❑ powerMode
- ❑ mode
- ❑ illegalDomainConfiguration

This is similar to the `xmsim` command, `power <object>` command.

© 2021 All Rights Reserved.

Use the `lp powerLossXDisplay [true | false | 1 | 0]` command to specify whether actual emulator values or X values should be returned. This command also affects the waveform. By default, the signal value displays as `X` during power loss. The `lp powerLossXDisplay 0` or `false` command changes it to not show X values and show the actual emulator values instead. The waveform also displays values instead of `X`.

Assertions

Assertions in the DUT remain active when their power domains are shut off. It is possible that assertion counters increment during power down if the assertion inputs toggle. This is also the default Xcelium behavior. To prevent assertion corruption from PSO while in software mode, Xcelium gets a modified version of the user CPF file. VXE does not support the `create_assertion_control` CPF directive.

Probing

In SA mode, the Xcelium CPF probes mostly work separately from the VXE CPF probes. However, the Xcelium `probe -power` command, which probes all power control signals, will also create hardware probes. For VXE CPF probes, the probing capability in SA mode is the same as in ICE mode. However, you can see the probes in the SimVision Design Browser only after you swap-in the design at the top hierarchy; the probes can be added before swap-in. Note that once the probes are added, these cannot be removed. These probes are neither listed in the output of the `probe -list` command, nor in the *Probe* tab of the xeDebug GUI. These probes are displayed after upload.

Xcelium provides the following CPF-specific options of the `probe` command. These options are supported when the design is swapped in to the software mode:

- `-pwr_mode`

  Probes all the power domains. The domains need not be specified individually. The probes appear in the SimVision Browser as objects at the correct hierarchy.

- `-power`

  Probes all power control signals.

- `-sr_save` [`-depth` <*level*>] [<*path*>]

  Probes the regular and retained values for the specified path.

© 2021 All Rights Reserved.

Probing in Offline Mode with CPF Objects in SA Mode

In offline debugging, the simulator is disconnected. Therefore, the Xcelium options listed above for the `probe` command are not supported. However, the Palladium probe options listed above are supported.

Browsing the Designs with CPF in SA Mode

The Palladium DUT browser does not support CPF. CPF design browsing can be done using the SimVision design browser after adding probes (DUT and testbench probes) with options, the Palladium CPF GUI, or in the SimVision CPF GUI.

Power Commands Supported in SA Mode

To run the design with CPF intent in SA mode, you can use any one of the following two sets of commands:

- Power-related commands supported by Xcelium

  When running the design on the software side, that is when the design run is swapped out of the hardware, you can issue the power-related commands supported by Xcelium. These commands are listed in *CPF Support in Simulation* in the *CPF Support* chapter of the *Low-Power Simulation Guide* available in the *Xcelium documentation set*. On software, you can also issue the set-up commands listed in Table 33-1.

- Power-related commands supported by VXE

  When the design is running on the hardware, you can issue the `lp` <*`CPF_command`*> command to print the value assigned to the specified CPF command in the CPF file. All the CPF commands listed in Table 33-1 in Support for CPF Commands on Palladium Z1 can be given with the `lp` command.

  In addition, to access static CPF data or the testbench objects, you can use some of the Xcelium commands.

Note: To compile a low-power-aware design for SA mode using the IXCOM compiler, you need to specify the `-lps_cpf` <*`cpf_file`*> option with the `ixcom` command.

SimVision Support for CPF in SA Mode

The *SimVision Power Display* sidebar displays the static and dynamic CPF data. When swapped-in, the dynamic CPF data will not show the data from the emulator. So, you should disregard the dynamic data when swapped into the hardware.

© 2021 All Rights Reserved.

Static data is presented as a hierarchy with the following information:

- Power domains
- Each domain's instances, isolation, and retention rules

The dynamic data is presented in the following ways:

- Lightbulb icon changes to on/off/partial on, and so on depending on cursor position on the waveform
- Hovering tooltip states whether an isolation rule is in effect
- Isolation and retention rule icons change depending on whether the rule is in effect

Control

The GUI also provides online control functionality for CPF that includes the following:

- Creating assertions on domain state changes - This will be ignored when the design is swapped in.
- Creating probes - HDL objects that are probed in simulation mode will carry in as probes when swapping in. Creating the HDL object probes when swapped in is supported. CPF objects are not supported.
- Force or release - Not supported when design is swapped in to software.
- Schematic tracer - Displays static information from the snapshot. No active annotation.
- Source browser - Displays the user CPF file.

For details, refer to the Xcelium documentation.

For details on the xeDebug CPF GUI, refer to Using the xeDebug GUI for Low Power Verification.

Differences between Xcelium and Palladium Z1 CPF Low-power Implementation

This section lists the differences between Xcelium and Palladium Z1 low-power implementation.

© 2021 All Rights Reserved.

Low-power options supported on Xcelium and Palladium Z1

The usage of low-power options on Xcelium and Palladium Z1 has the following differences:

- Xcelium supports a number of low-power options. However, only the `-lps_cpf` option is supported on Palladium Z1 in SA mode.
- Xcelium supports testbench (`-testbench`) power-domain shutoff whereas VXE ignores it with a warning, that is, no shutoff and corruption.
- In Xcelium, the retention rule without the save and/or restore condition is applied according to the `default_save_condition` or `default_restore_condition` that was specified with the power domain. In VXE, this is true only if the retention rule is specified with the power domain; not if it is specified with the instance list.

Features supported on Xcelium and Palladium (Swapped-In) Acceleration

The following features are supported differently in Xcelium as compared to VXE when the design is swapped into hardware:

- Break on CPF condition changes - You can specify these using SDL.
- Generation of CPF assertions
- The implementation of save/restore on the Palladium Z1 emulator is different because the save condition on posedge disables the registers and the restore condition on posedge re-enables the registers. Therefore, a write operation after the save operation does not take effect. No warning is issued for this. In Xcelium, the write operations after the save operation do take effect.
- In VXE, there is no support for getting the saved values (if retention rule applies) if the signal is in PSO and the power-up/down state is specified in the CPF file. This is different from Xcelium because in Xcelium, you can use the `value -saved` command to retrieve saved values.
- For retention conditions, in simulation, when the retention condition is applied, the logic continues without any effect. On the Palladium Z1 emulator, on the other hand, when the retention condition is true, the logic does not continue. For example, suppose the `counter instA.mem[5:0]` holds the value `'h6` at time `x`, when its retention edge goes `false`, causing a save. In Xcelium, this does not prevent subsequent `run` commands from incrementing the counter, but on the Palladium Z1 emulator, the counter value will remain fixed.
- Memories in PSO state in emulation show the value which is specified with the `-power_down_states` option; in simulation, these show `XX`.

© 2021 All Rights Reserved.

- According to the CPF LRM, in simulation, if unspecified, the `-power_up_states` remain `X`, but the default is random, and in emulation, these switch to random.
- According to the CPF LRM, in simulation, if unspecified, the `-power_down_states` follow the simulation-default power-down semantics, but because the default is `low`, in the emulation these switch to low.
- In VXE, memories listed as instances in the CPF file in VXE are ignored with a warning message. However, in simulation, these are accepted.
- The `save_precondition` is supported in emulation. The save operation occurs when the `save_condition` and `save_precondition` are both true. In Xcelium, on the other hand, the save operation occurs on the rising edge without precondition, but on the falling edge with precondition. Note that the `-default_save_level` and `-default_restore_level` options are not supported.
- Xcelium supports certain syntax that does not follow the CPF 1.1 standard. For example, instance name can be used with the `set_design` command, which actually requires the model name.
- In order to have similar support as Xcelium, in VXE, the instance name is honored with the `set_design` command if you have either included the `-testbench` option, or you have specified the instance name with the `set_design` command of the top cell, that is, the first `set_design` command.

Using Dynamic Power Analysis with Low-Power-Aware Designs

Dynamic Power Analysis (DPA) can be combined with a low-power-aware design to get the following main benefits:

- Fast running of the designs
- Support for very large capacity designs with real software application environment

The power estimation and power profile are impacted when parts of the logic are in power-off condition. This allows you to get an estimate of the power reduction the design will experience from low-power design.

Note: The `-power_up_states` or `-power_down_states` option of the `create power domain` command impacts the power estimation by creating switching transitions to the new state.

For detailed information, refer to Analyzing Power Consumption with CPF of Chapter 32, “Analyzing Power Consumption of the Designs.”

© 2021 All Rights Reserved.

The following diagram presents the basic architecture for low-power-aware designs:

Figure 33-8 Basic Architecture - Low-Power-Aware Designs

To verify the low-power-aware design with emulation systems:

- The inputs are HDL files, CPF files, compiler commands, debug commands, SDL files;
- The environments are either running with target systems or without target systems, either running with emulation online, or debugging offline without emulator running;
- The outputs of the specified HDL and CPF files are as follows
  - Message files
  - Log files in STB and LA modes
  - SST2 with values marked as X for power loss
  - SST2 with emulator values
  - FSDB with values marked as X for power loss
  - FSDB with emulator values

© 2021 All Rights Reserved.

- TCF with emulator values for pins or nets
- SAIF with emulator values for pins or nets
- VCD with emulator values for pins or nets
- Value for signals
- Value for CPF objects
- TCA with power domains
- Power consumption profile with power domains

Selecting Low-Power-Aware Verification Method Based on Design Type

The following sections discuss low-power-aware verification methods for each type of design:

- Emulating RTL Low-Power-Aware Design
- Emulating Gate-Level Low-Power-Aware Design

Emulating RTL Low-Power-Aware Design

System-level designs can include RTL designs without any low-power intention, and some IP with or without low-power intention.

For RTL and IP part without low-power intention, use CPF files to specify the low-power intentions. For the hard IP part with low-power intention, use macro CPF modules to model the intent in a CPF file.

Table 33-2 covers the type of low-power techniques you can use for different parts of system-level designs.

© 2021 All Rights Reserved.

Table 33-2 Low-Power Techniques for Different Parts of System-Level Designs

Parts of System-Level Design	Possible Low-Power Techniques	Parts of CPF File
RTL without low-power intent	■ PSO ■ DVFS ■ Standby	■ Power intention with RTL names ■ Flatten CPF ■ Hierarchical CPF
Soft IP without low-power intent	■ PSO ■ DVFS ■ Standby	■ Power intention with name in Soft IP simulation model ■ Flat CPF ■ Hierarchical CPF
Hard IP with low-power intent	■ PSO ■ DVFS ■ Standby	■ Macro CPF before DUT CPF and flat CPF ■ Macro CPF with hierarchical CPF

For details related to hierarchical CPF, refer to Handling Hierarchical CPF Files.

Emulating Gate-Level Low-Power-Aware Design

You can directly instantiate low-power cells (switching cells, isolation cells, retention cells, and so on) in the gate-level netlist, or you can describe the power intention in CPF files. Palladium Z1 can be used to emulate gate-level netlist when the power intention is described in CPF files. It does this by adding instrumentation logic to mimic the low-power behaviors. Palladium Z1 cannot be used to emulate gate-level netlists which include low-power cells.

When a gate-level netlist with CPF is used, the CPF description can use netlist object names or original RTL object names.

Table 33-3 covers all the possible cases of emulating a gate-level low-power-aware design.

© 2021 All Rights Reserved.

Table 33-3 Low-Power Techniques for Different Parts of Gate-Level Netlists

Parts of Gate-level Netlist	Possible Low-power Techniques	Parts of CPF file
Gate-level netlist without low-power intent	■ PSO ■ DVFS ■ Standby	■ Power intention with gate-level names from the netlist ■ Flatten CPF ■ Hierarchical CPF
Soft IP without low-power intent	■ PSO ■ DVFS ■ Standby	■ Power intention with name in Soft IP simulation model ■ Flat CPF ■ Hierarchical CPF
Hard IP with low-power intent	■ PSO ■ DVFS ■ Standby	■ Macro CPF before DUT CPF and flat CPF ■ Macro CPF with hierarchical CPF

© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

34

Working with IEEE 1801 Low-Power Designs on Palladium Z1

Unified Power Format (UPF) or IEEE 1801 is a standard format by IEEE for specifying all design and technology-related power constraints for low-power applications. IEEE 1801 provides the infrastructure to specify power constraints at RTL-level of design. A UPF or 1801 file acts as a repository for all power-related information for a design. The low-power design intent can also be specified either in an HDL file or extracted from a liberty file.

VXE supports UPF 1.0 and IEEE 1801 commands. By default, VXE follows IEEE1801-2013 semantics.

IEEE 1801 provides the ability to design power intentions early in design process. Palladium models supply network, power off on corruption, isolation, and retention in the emulation netlist image, which is generated from RTL files and 1801 files.

In ICE mode, the isolation, retention and power switch behavior take effect after several fclks when the control signals are active. The delay is fixed in an emulation model and does not depend on the testcases or designs. VXE does not display a warning at compile time about the delay.

For more details about using IEEE 1801 for a design and the supported 1801 commands, refer to Appendix 35, “Support for IEEE 1801 Commands on Palladium Z1”.

This chapter includes the following sections:

- IEEE 1801 Support Architecture
- Specifying IEEE 1801 Versions and Semantics
- Processing IEEE 1801 Information on Palladium Z1
- Controlling 1801 Instrumentation Logic at Run-Time
- Compiling Low-Power Designs
- Running and Debugging Designs

© 2021 All Rights Reserved.

- Verifying Low Power Aware Designs using the xeDebug GUI
- Viewing Power Information for Probes in Xcelium SimVision GUI
- Simulating with xrun
- Specifying Power Back Initialization Values
- Using Wildcards in IEEE 1801 Commands
- Using 1801 Boolean Expressions
- Modeling Special ARM Retention Cells
- Modeling Cells Using NOR Isolation Strategy
- Performing Dynamic Power Analysis for Designs with 1801 Files
- Controlling Assertions for Powered Down Domains
- Logging IEEE 1801 Events
- Handling IEEE 1801 Illegal Events
- Enabling UPF 1.0 Isolation Semantics
- Locating Encypted Design Instances
- Identifying Partial Power-Aware Modules and Handling Corruptions
- Specifying Default Value Conversion Table (VCT)
- Viewing Low Power Information in Palladium Database
- IEEE 1801 Support in Xcelium and Palladium Z1
- Importing IEEE Standard UPF Packages
- Generating a Coverage Model for 1801 Low-Power Designs
- Viewing Power Loss in Waveforms

IEEE 1801 Support Architecture

The following diagram presents the basic architecture for low-power-aware designs:

© 2021 All Rights Reserved.

Figure 34-1 Basic Architecture of Low-Power-Aware Designs

You can run the low power-aware designs either with or without the target systems. You can also debug offline. To verify the low-power-aware design with emulation systems, the inputs are HDL files, HDL 1801 package wrapper, 1801 files (`.upf`), liberty files (`.lib`), compiler commands, debug commands, and SDL files. VXE provides a utility, called `libTo1801`, to help you extract low power information from liberty files and generate IEEE 1801 commands, which can be read with the other 1801 files during compilation.

The output of the specified HDL and 1801 files, which includes the 1801 files generated by `libTo1801`, are as follows:

- Message files
- Log files in STB and LA modes
- SST2 with emulator values
- FSDB with emulator values
- Value for signals

© 2021 All Rights Reserved.

- Value for 1801 objects

Specifying IEEE 1801 Versions and Semantics

VXE supports syntax of the UPF 1.0, UPF 2.0 (IEEE 1801-2009), and UPF 2.1 (IEEE 1801-2013) versions. By default, VXE follows UPF 2.1 (IEEE 1801-2013) for semantics.

To select the syntax of a specified IEEE 1801 version, use the IEEE standard `upf_version` command. This command does not determine semantics of the IEEE 1801 commands or options. In general, the 2.1 version semantics is followed.

However, the semantics of the `-ack_port` option of the `create_power_switch` command is determined by `upf_version` command.

To select the domain interface semantics of the specified IEEE 1801 version, issue the following command. This command also determines how to add isolation instrumentation logic.

```
precompileOption -add domainInterfaceDef
```

The `-defaultAppliesToOutputs` option of the `read_power_intent` command sets the default value of the `-applies_to` option of the `set_isolation` command based on the specified IEEE 1801 version.

Note: VXE does not support mixed version semantics.

For more information about how the interface of a domain is defined in a 1801 version, refer to *IEEE Standard for Design and Verification of Low-Power Integrated Circuits* released by IEEE.

Defining the Power Supply Network

An IEEE 1801 file describes power supply networks and specifies, at an abstract level, how these supply networks are related to design elements. A power supply network consists of power domains, power supply ports, supply nets, and power switches, and routes power supplies to design elements, whose state during power-aware simulation is determined based on their power supplies. The power supply network must be defined in the power intent file.

To define the power supply network, you must:

- Define the power supply ports

© 2021 All Rights Reserved.

- Define the power supply nets
- Connect the supply nets to the supply ports
- Create supply sets
- Define the power domains and specify the supply sets for the domains
- Define the power switches

You can also:

- Read power information from liberty (`.lib`) files
- Define power state information
- Define information on ports
- Define logic ports and nets
- Connect a logic net to logic ports

Processing IEEE 1801 Information on Palladium Z1

The following table lists the various commands supported in VXE to process the low-power aware designs.

Table 34-1 VXE Commands to Process IEEE information

Task	Command
To check the 1801 file syntax	`check_power_intent`
To add IEEE 1801 information to a design	`read_power_intent` **Note:** The 1801 files cannot reference protected design objects. If protected design objects appear in 1801 files, the `read_power_intent` command cannot recognize them.
To delete all power intent information	`clean_power_intent`

© 2021 All Rights Reserved.

Table 34-1 VXE Commands to Process IEEE information

Task	Command

To display the saved power intent information at both compile and run time	`lp -check <1801_command> [<namePattern>]`
	`lp -check powerIntent | 1801 [<objectType>]`
	`lp -check files`
	`lp -check Is1801`
	
	The `lp` command should be issued after the `read_power_intent` command.

Note: You can also enable the precompile to ignore the power intent and power intent instrumentation logic using the `-add noPowerIntent` option of the `precompileOption` command. This option ignores the power intent and the power intent related instrumentation logic,i.e, the control signals are not added into the netlist.

Controlling 1801 Instrumentation Logic at Run-Time

By default, 1801 instrumentation logic is enabled for designs with 1801 files. You can disable the 1801 instrumentation logic at run time by executing the `lp -runDisable` command. This command also disables all `lp` run time commands and 1801 objects probing. The `lp` run-time commands and `probe -create` command fail to execute successfully and return error messages if executed after the `lp -runDisable` command.

Compiling Low-Power Designs

The precompile adds instrumentation logic based on the committed power intent.

In SA mode, invoke the IXCOM or `xrun -hw` with the additional `-lps_1801 <1801Filename>` option to compile the UPF file. Currently, only one 1801 file is supported; if multiple files are specified, only the last 1801 file is used. All options available in the ICE mode for the `read power intent` command are also available in the SA mode by including the options as arguments to the `-xecompileargs` compilation option as follows:

```
-xecompileargs "-1801 <read_power_intent (space-delimited) options> --"
```

For example,

```
-xecompileargs "-1801 -defaultAppliesToOutputs 1.0 -force --"
```


© 2021 All Rights Reserved.

Overriding Ports During Compilation

If the specified UPF syntax violates the precedence of the `set_port_attributes` command defined in IEEE1801-2013 Clause 5.8 and IEEE1801-2015 Clause 5.7, precompile might fail with an error message similar to the following:

```
ERROR (multclk-1764): Different receiver_supplies are specified for port X.Y.Z in
set_port dut.upf, line 10) and (file dut.upf, line 20)
```

To ignore the error message and direct the precompile to proceed by honoring the last value, use the `precompileOption -add allowOverrideSupply` command.

For example:

```
# in file dut.upf
set_port_attributes -ports {Z} -receiver_supply VDD
set_port_attributes -ports {Z} -receiver_supply VDD_SW
```

In the above example, you can direct the precompile to honor the last specified value — that is, `VDD_SW`, using the `precompileOption -add allowOverrideSupply` command.

Identifying Fake Modules in SA mode

In UPF files, the missing hierarchical instance path to the design's fake and IP modules is not honored by xeCompile and Xcelium. When compiled in SA mode, the downstream tools (xeCompile and Xcelium) display error messages for those fake modules. In such cases, you need to modify the UPF file to eliminate the error messages.

However, you can avoid modifying the UPF code, and reuse the same file for different HDL versions — while compiling your design in SA mode, specify the fake or IP modules using the `-upfFakeModuleStubList` option of the `ixcom` command, as follows:

```
<ixcom> … -upfFakeModuleStubList <upfFakeModulesStubListFile>
```

For more information, syntax, and examples, refer to the description of the `-upfFakeModuleStubList` option in the *Commands and System Tasks for SA Mode* chapter of the *VXE Command Reference Manual*.

Running and Debugging Designs

This section includes the following:

- Probing Low-Power Objects and Viewing Waveforms
- Checking Content of IEEE 1801 Objects
- Getting Value of IEEE 1801 Objects

© 2021 All Rights Reserved.

- Using IEEE 1801 Conditions in SDL Expressions
- Running Designs in SA Mode

Probing Low-Power Objects and Viewing Waveforms

You can add power domains, isolation or retention strategies, supply ports, supply nets, supply sets, power switches, pst, port states, and pst states into a probe set using the following commands:

```
probe -addpowerdomain <namePattern>
probe -addisolation <namePattern>
probe -addretention <namePattern>
probe -addsupplyport <namePattern>
probe -addsupplynet <namePattern>
probe -addsupplyset <namePattern>
probe -addswitch <namePattern>
probe -addpst <namePattern>
probe -addportstate <namePattern>
probe -addpststate <namePattern>
```

For more information, refer to the description of the `probe` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

The `<namePattern>` can include wildcards `*` and `?`. For example, if under an instance `top.dut.m1`, three power domains `PD1`, `PD11`, `PD111` are created, the `top` instance only has instance `dut`, then the `probe -addpowerdomain top.d*.m1.PD?` command will only add `PD1` under the `top.dut.m1` instance into the probe set.

Use a dot (.) as hierarchical separator in the name pattern. For example, if a power domain `PD1` is created under an instance `top.dut`, the `probe -addpowerdomain top.dut.PD1` command will add the `PD1` domain under the `top.dut` instance into probe set.

The waveform of a probed signal is generated in FSDB or SST2. Use the `database -upload` command to add the probed IEEE 1801 objects into SimVision or Novas tools — Verdi and Siloti.

All IEEE 1801 objects in waveform are under the `power_intent_1801_objects.<Object>` hierarchical scope.The content of a probed object depends on its specified options. If driver of a probed signal is in power loss state, the signal on the waveform shows `0` or `1` based on the value it already has or it has derived from the emulator.

For example. all IEEE 1801 power domains in waveform are displayed under the `power_intent_1801_objects.PowerDomain` hierarchical scope, as shown in Figure 34-2.

© 2021 All Rights Reserved.

Figure 34-2 Waveform - IEEE 1801 Power Domain

Note: Low power instrumentation requires a large number of PI probes. The PI probes on instrumentation nets cannot be reduced even if selective FV is used — that is, if you have enabled or disabled certain nets for selective FV using the `instanceFV` and `moduleFV` command, those instances would not be considered for selective FV, and the entire design would be computed for FV.

Checking Content of IEEE 1801 Objects

You can use the `lp` command to check static content or dynamic values of low-power intent objects. The `lp` command is also used to turn on or off the run-time power intent support.

The following commands can be issued on both xeCompile and xeDebug command line:

```
lp -check <1801_command> [<namePattern>]
lp -check powerIntent | 1801 [<objectType>]
lp -check files
lp -check Is1801
```


© 2021 All Rights Reserved.

The following commands can be issued only on xeDebug command line:

```
lp -check powerDomain | domain [<namePattern>]
lp -check isolation [<namePattern>]
lp -check retention [<namePattern>]
```

Besides the UPF package functions in the HDL files, you can also use the following `lp` command to update the power state and voltage of the specified supply port or supply net during run time.

```
lp -assign_supply_state -port <portName>
                        -net <netName>
                        -state {on | off | partialOn} [-voltage <realNumber>]
```

Getting Value of IEEE 1801 Objects

Issue the following `lp` commands on xeDebug command line to get the value of the IEEE 1801 objects:

```
lp -value powerdomain <namePattern>
lp -value retention <namePattern>
lp -value isolation <namePattern>
lp -value supplyport <namePattern>
lp -value supplynet <namePattern>
lp -value supplyset <namePattern>
lp -value pst <namePattern>
```

Using IEEE 1801 Conditions in SDL Expressions

Palladium Z1 emulator enables you to specify IEEE 1801 conditions in SDL expressions for enhanced verification and debugging of the low-power intent in your designs.

The IEEE 1801 conditions can be used to do the following:

- Identify whether the state or attribute of 1801 objects is true or false.
- Define the power control sequences and then check at run time to see whether the sequence happens or not.

The format of IEEE 1801 conditions in an SDL expression is as follows:

```
[power::]<1801Object>
[power::]<1801Object>@<state>
[power::]<1801Object>@<1801Options>@<state>
[power::]<1801Object>@corrupted@<state>
[power::]<1801Object>@corrupted
[power::]<1801Object>@isolation
[power::]<1801Object>@inputDriverCorrupt
[power::]<1801Object>@domainCorrupt
[power::]anyIsolationControlSignalCorrupt
[power::]anyIsolationControlSignalCorruptTrigger
```


© 2021 All Rights Reserved.

```
[power::]anyRetentionControlSignalCorrupt
[power::]anyRetentionControlSignalCorruptTrigger
```

- Here, `power ::` indicates that the condition is related to power intent. This is optional. The purpose of using it is to avoid a signal with the same name in the design database. *<1801Object>* is a 1801 object defined using the `create_...` or `set_...` IEEE 1801 commands. The 1801 object can be a supply port, supply net, supply set, power domain, power switch instance, power retention strategy, or power isolation strategy.
- *<1801Options>* is an option of the `create_...` or `set_...` IEEE 1801 commands used to create or set each 1801 object.
- *<state>* is the state, true or false, of a 1801 object.

You can use IEEE 1801 conditions to define 1801 illegal sequence, and then check 1801 illegal sequence during run time. For example, to check the illegal sequence, such as, isolation strategy `ISO` must be enabled before power domain `PD1` primary supply is `OFF`, add the following statement to the SDL file:

```
If (POWER::PD1 == `bP && !(POWER::ISO@isolation)
Trigger;
```

SDL auto probe will automatically probe 1801 objects when IEEE 1801 conditions appear in the SDL expressions.

The following table lists basic IEEE conditions and the 1801 objects that will be probed when an IEEE 1801 condition appears in the SDL expression:

Note: The prefix `power::` of 1801 condition name is optional. The prefix `power::` is used to differentiate between the similar design objects.

IEEE 1801 Conditions

1801 Condition	Description	Probed 1801 Object
`Power::`*<supplyPort>* `@`*<state>*	True if the *<supplyPort>* is at *<state>*, false if not. The *<state>* is defined with the `add_port_state` command.	`probe -addportstate` *<supplyPortState>*

© 2021 All Rights Reserved.

IEEE 1801 Conditions

1801 Condition	Description	Probed 1801 Object
`Power::<pstState>`	True if the `<pstState>` is true, false if not. The `<pstState>` is defined with the `add_pst_state` command.	`probe -addpststate <pstState>`
`Power::<supplySet>`	True if the `<supplySet>` is in `CORRUPT` state, false if not.	`probe -addsupplyset <supplySet>`
`Power::<stateTransition>`	True if the state transition happens, false if not. The `<stateTransition>` is defined with the `describe_state_transition` command.	None
`Power::<powerDomain>`	True if the `<powerDomain>` primary supply is power off, false if not.	`probe -addpowerdomain <powerDomain>`

© 2021 All Rights Reserved.

IEEE 1801 Conditions

1801 Condition	Description	Probed 1801 Object
■ Power::<*switch*>@on_state@<*state*> ■ Power::<*switch*>@off_state@<*state*> ■ Power::<*switch*>@on_partial_state@<*state*> ■ Power::<*switch*>@error_state@<*state*>	True if the <*state*> of <*switch*> based on <*option*> is true, false if not. The <*option*> can be: ■ on_state ■ off_state ■ on_partial_state ■ error_state	probe -addswitch <*switch*>
Power::<*retention*>	True if <*retention*> supply is CORRUPT, false if not.	probe -addretention <*retention*>
Power::<*retention*>@corrupted	True if power supply of any signals in <*retention*> is corrupted, false if not.	probe -addretention <*retention*>
■ Power::<*retention*>@save_condition ■ Power::<*retention*>@restore_condition ■ Power::<*retention*>@retention_condition	True if the condition of the <*option*> save_condition for <*retention*> is true, false if not. The <*option*> can be save_condition, restore_condition, or retention_condition.	probe -addretention <*retention*>

© 2021 All Rights Reserved.

IEEE 1801 Conditions

1801 Condition	Description	Probed 1801 Object
`Power::<isolation>`	True if `<isolation>` supply is corrupted, false if not.	`probe -addisolation <isolation>`
`Power::<isolation>@isolation`	True if `<isolation>` is at `isolation` state, false if not.	
`Power::<isolation>@corrupted`	True if supply of any control signals in `<isolation>` is corrupted, false if not.	
`Power::<isolation>@inputDriverCorrupt`	True if `<isolation>` is inactive when any supply of the isolated inputs' drivers is off.	
`Power::<isolation>@domainCorrupt`	True if `<isolation>` is inactive when the domain's supply is off.	
`Power::anyIsolationControlSignalCorrupt`	True if any isolation control signal is corrupted.	
`Power::anyIsolationControlSignalCorruptTrigger`	True each time when an isolation control signal is corrupted.	
`Power::anyRetentionControlSignalCorrupt`	True if any retention control signal (appear in one of save/restore/ retention conditions) is corrupted.	

The `lp -check sdl` run-time command displays a list of all possible 1801 conditions for the given design with 1801 files. You can test the state of an IEEE 1801 object using SDL expressions, as follows:

```
<1801 condition> == <UPF-state>
```


© 2021 All Rights Reserved.

```
<1801 condition> == 0
```

or

```
<1801 condition> == 1
```

To view the legal 1801 condition names and their corresponding values, use the `lp -check sdl` [<*1801 condition type*>] command. The values for each sdl event in the `lp -check` sdl command are displayed in the following format:

```
<1801 condition type> { {<1801 condition name> {value list}} …}
<1801 condition ype> { { {<1801 condition name> <1801 condition domain member name>
} {value list} …}
```

The following table lists various low power conditions, values, and the mnemonics. The state mnemonics should be double quoted in SDL expression such as `<1801 condition> == "<state mnenomics>"`. For more information on SDL mnemonics, refer to the IEEE 1801 Conditions section.

Table 34-2 1801 Conditions, Values, and State Mnemonics

1801 Condition Type	1801 Condition	Value or Sate Mnemonics

Global	`power::anyIsolationControlSignalCorrupt`
	`power::anyIsolationControlSignalCorruptTrigger`
	`power::anyRetentionControlSignalCorrupt`
	`power::anyRetentionControlSignalCorruptTrigger`
	`power::anyIllegalEvent`
	`power::runLPDisable`
		1 or 0

supplyNets	`power::<supplyNet>`	OFF FULL_ON, PARTIAL_ON or UNDETERMINED
supplyPorts	`power::<supplyPort>`	OFF FULL_ON, PARTIAL_ON UNDETERMINED <*port state name*> list
supplyPortStates	`power::<supplyPortState>`	1 or 0

© 2021 All Rights Reserved.

Table 34-2 1801 Conditions, Values, and State Mnemonics

supplySet	power::<supplySet>	NORMAL CORRUPT <*power state name*> list
Domains	power::<domain>	NORMAL CORRUPT <*power state name*> list
Isolations	power::<isolation>	NORMAL, CORRUPT ISOLATED, NOT_ISOLATED
	power::<isolation>@isolation power::<isolation>@corrupted power::<isolation>@inputDriverCorrupt power::<isolation>@domainCorrupt	1 0r 0
Retentions	power::<retention>	NORMAL, CORRUPT RETAINED, NOT_RETAINED
	power::<retention>@retention power::<retention>@corrupted power::<retention>@save_condition power::<retention>@restore_condition power::<retention>@retention_condition	1 or 0
Switches	power::<switch>	<*switch state name*> list
	power::<switch>@on_state@<state> power::<switch>@off_state@<state> power::<switch>@error_state@<state>	1 or 0
pst	power::<pst>	<*pst state name*> list
pstStates	power::<pstState>	1 or 0

© 2021 All Rights Reserved.

Table 34-2 1801 Conditions, Values, and State Mnemonics

powerStates	`power::<object>.<powerState>` `power::<object>.<powerState>@corrupted`	1 or 0
stateTransitions	`power::<stateTranisition>`	1 or 0

Example: lp -check sdl

```
XE> lp -check sdl
sdl {
+global {
{anyRetentionControlSignalCorrupt 1 0}
{anyRetentionControlSignalCorruptTrigger 1 0}{anyIllegalEvent 1 0}

{runLPDisable 1 0}}
<supplyNet> {{
power::dut.VDD {OFF FULL_ON PARTIAL_ON UNDETERMINED}}}
<supplyPort> {{
power::dut.VDD {OFF FULL_ON PARTIAL_ON UNDETERMINED }}}
<supplyPortState> {{
power::dut.VDD_sw1.sw1on {1 0}} }
<domain> {{
power::dut.PD_Def {NORMAL CORRUPT }} }
<isolation> {{
{power::dut.iso0 power::dut.PD_bank0.iso0} {NORMAL CORRUPT ISOLATED
NOT_ISOLATED}}}
<isolation>@isolation {{
{power::dut.iso0@isolation power::dut.PD_bank0.iso0@isolation} {1 0}} }
<isolation>@domainCorrupt {{
{power::dut.iso3@domainCorrupt power::dut.PD_bank3.iso3@domainCorrupt} {1 0}}}
<retention> {{
{power::dut.ret_9 power::dut.PD_bank9.ret_9} {NORMAL CORRUPT RETAINED
NOT_RETAINED}}}
<switch> {{
power::dut.sw_0 {sw_ON }} {
power::dut.sw_1 {sw_ON sw_OFF }} {
power::dut.sw_2 {sw_ON sw_OFF }} {
power::dut.sw_3 {sw_ON sw_OFF sw_ERR }} {
power::dut.sw_4 {sw_ON sw_OFF sw_ERR sw_ERR2 }}}
<pst> {{
power::dut.TABLE_1 {off on }} {
power::dut.TABLE_12 {both_off both_on one_on two_on }}}
<pstStates> {{
{power::dut.off power::dut.TABLE_1.off} {1 0}} {
{power::dut.on power::dut.TABLE_1.on} {1 0}} {
{power::dut.both_off power::dut.TABLE_12.both_off} {1 0}} {
{power::dut.both_on power::dut.TABLE_12.both_on} {1 0}} {
{power::dut.one_on power::dut.TABLE_12.one_on} {1 0}} {
{power::dut.two_on power::dut.TABLE_12.two_on} {1 0}}}
}
```


© 2021 All Rights Reserved.

Example: lp -check sdl domains

```
XE> lp -check sdl domains
sdl  {
<domain> {{
power::PD_TOP  {NORMAL CORRUPT }} {
power::PD_C0   {NORMAL CORRUPT }} {
power::PD_C1   {NORMAL CORRUPT }} {
power::PD_C2   {NORMAL CORRUPT }}}
}
```

Example: lp -check sdl isolations

```
XE> lp -check sdl isolations
sdl  {
<isolation> {{
{power::ISO_C0  power::PD_C0.ISO_C0}    {NORMAL CORRUPT ISOLATED NOT_ISOLATED}} {
{power::ISO_C1  power::PD_C1.ISO_C1}    {NORMAL CORRUPT ISOLATED NOT_ISOLATED}} {
{power::ISO_C2  power::PD_C2.ISO_C2}    {NORMAL CORRUPT ISOLATED NOT_ISOLATED}}}
<isolation>@isolation {{
{power::ISO_C0@isolation        power::PD_C0.ISO_C0@isolation}  {1 0}} {
{power::ISO_C1@isolation        power::PD_C1.ISO_C1@isolation}  {1 0}} {
{power::ISO_C2@isolation        power::PD_C2.ISO_C2@isolation}  {1 0}}}
<isolation>@corrupted {{
{power::ISO_C0@corrupted        power::PD_C0.ISO_C0@corrupted}  {1 0}} {
{power::ISO_C1@corrupted        power::PD_C1.ISO_C1@corrupted}  {1 0}} {
{power::ISO_C2@corrupted        power::PD_C2.ISO_C2@corrupted}  {1 0}}}
<isolation>@inputDriverCorrupt {}
<isolation>@domainCorrupt {{
{power::ISO_C0@domainCorrupt    power::PD_C0.ISO_C0@domainCorrupt}      {1 0}} {
{power::ISO_C1@domainCorrupt    power::PD_C1.ISO_C1@domainCorrupt}      {1 0}} {
{power::ISO_C2@domainCorrupt    power::PD_C2.ISO_C2@domainCorrupt}      {1 0}}}
}
```

Running Designs in SA Mode

You can download the design onto the emulator and turn on or turn off the different power domains using the HDL or UPF commands (1801) file. Ensure that a power domain does not exist across hardware or software boundary; otherwise, the backend compilation will fail. A supply net in the testbench might manipulate a power domain in the DUT.

Use the following system tasks that might be helpful in connecting UPF control signals across the TB-DUT boundary.

- `$ixc_ctrl("dut_driver",<dut signal>)` identifies a DUT signal that should be deposited to the TB. This might be needed if you expect some UPF signals to originate from the DUT and drive the TB, for example, an acknowledge port for a power switch in the DUT.
- `$ixc_ctrl("tb_signal",<tb signal>)` identifies a TB signal that should be deposited to the DUT. This might be needed if the UPF file uses a TB signal to control

© 2021 All Rights Reserved.

some UPF logic, for example, a power control management unit in the TB driving a shutoff signal to a power domain in the DUT.

Note: The IXCOM compiler does not process the IEEE 1801 information directly. So, it might not instrument some signal ports between the testbench and DUT that are needed for low power operation.

Hot Swap

In SA mode, the IXCOM compiler does not impose any limitation on *hot swap*. However, there is no guarantee that the power states will be maintained on swap-out from the emulator.

During hot swap from the simulator to the emulator, the states of the power domains, isolations, and retention strategies are transferred from the simulator into the emulator. The supply nets and supply ports are set to the same state and voltage. The values of the retained elements are also transferred.

Caution

During swap out from the emulator, no low power state information is transferred to the simulator.

Netlist Flow

The 1801 support in SA mode works for netlist flow. There is no hierarchy difference if a portion of the design comes from the netlist modules or their behavioral HDL equivalents.

Assertions

Assertions remain powered on during shut-off of the power domains in which they reside. Assertion counters are not included in retention strategies.

Coverage

Currently, functional, block, or toggle coverage are not supported for 1801 designs. Once a power domain is turned off and subsequently on, there is no guarantee that the coverage results will be maintained.

© 2021 All Rights Reserved.

Verifying Low Power Aware Designs using the xeDebug GUI

The xeDebug detects if the design has IEEE 1801 information. When you click on the *Low Power Verification* icon on the xeDebug GUI, if the design has 1801 information, the 1801 specific GUI appears. If the design has CPF information, the CPF specific GUI appears. If low power information is not present in the design, the icon is disabled.

The Low Power Verification GUI for IEEE 1801 has the following tabs:

- Runtime Tab
- Illegal Events Tab
- Power Intent Tab
- Power Supply Tab
- Isolation Tab

Runtime Tab

The Runtime tab is for run-time control of the design and verification. It enables you to perform the following tasks through the GUI:

- Set the supply states and voltages of supply nets and supply ports

  `lp -assign_supply_state -port|-net <name> -state <state> -voltage <voltage>`

- List each isolation strategy for which pins get isolated

  `lp -check isolation [pattern]`

- List each retention strategy for which elements get retained

  `lp -check retention [pattern]`

- Create SDL expressions for IEEE 1801 that can be used either as an SDL trigger expression or as part of an SDL program
- Get the current state of the 1801 objects that include supply nets, supply ports, supply sets, pst, power state, power domains, isolation, and retention rules

  `lp -value <1801objectType>`

- Probe and upload 1801 objects that include supply nets, supply ports, supply sets, power switches, power domains, isolation, and retention rules

© 2021 All Rights Reserved.

Figure 34-3 IEEE 1801 - Runtime

Illegal Events Tab

The Illegal Events tab enables you to view illegal condition checks in an expandable tree-view under types - Isolations, Retentions, Power states, Illegal state transitions, and Power switches. The types are sorted by a specific strategy.

You can enable or disable assertions using the right mouse button.

© 2021 All Rights Reserved.

Figure 34-4 IEEE 1801 - Illegal Events

1801 Illegal Condition Checks	State
⊟ Isolations	
⊟ dut.PD_bank0.iso0	
LPV_ISO_dut.PD_bank0.iso0_DOES_NOT_TOGGLE_DURING_TARGET_NORMAL	enabled
LPV_ISO_dut.PD_bank0.iso0_ENABLED_WHEN_DOMAIN_CORRUPT	enabled
LPV_ISO_dut.PD_bank0.iso0_WHILE_ON_TARGET_DOES_NOT_TOGGLE	enabled
⊞ dut.PD_bank1.iso1	
⊞ dut.PD_bank2.iso2	
⊞ dut.PD_bank3.iso3	
⊟ Retentions	
⊟ dut.PD_bank0.ret_0	
LPV_DMN_dut.PD_bank0.ret_0_ALWAYS_SAVE_BEFORE_RESTORE	enabled
LPV_DMN_dut.PD_bank0.ret_0_NO_RESTORE_WHILE_SHUTOFF	enabled
LPV_DMN_dut.PD_bank0.ret_0_NO_SAVE_WHILE_SHUTOFF	enabled
LPV_DMN_dut.PD_bank0.ret_0_RESTORE_AFTER_SHUTOFF	enabled
LPV_DMN_dut.PD_bank0.ret_0_RETAINED_VALID_UNTIL_RESTORE	enabled
LPV_DMN_dut.PD_bank0.ret_0_SHUTOFF_BEFORE_RESTORE	enabled
LPV_DMN_dut.PD_bank0.ret_0_SHUTOFF_FOLLOWS_SAVE	enabled
⊟ dut.PD_bank1.ret_1	
LPV_DMN_dut.PD_bank1.ret_1_ALWAYS_SAVE_BEFORE_RESTORE	enabled
LPV_DMN_dut.PD_bank1.ret_1_NO_RESTORE_WHILE_SHUTOFF	enabled
LPV_DMN_dut.PD_bank1.ret_1_NO_SAVE_WHILE_SHUTOFF	enabled
LPV_DMN_dut.PD_bank1.ret_1_RESTORE_AFTER_SHUTOFF	enabled
LPV_DMN_dut.PD_bank1.ret_1_RETAINED_VALID_UNTIL_RESTORE	enabled
LPV_DMN_dut.PD_bank1.ret_1_SHUTOFF_BEFORE_RESTORE	enabled
LPV_DMN_dut.PD_bank1.ret_1_SHUTOFF_FOLLOWS_SAVE	enabled
⊞ dut.PD_bank2.ret_2	
⊞ dut.PD_bank3.ret_3	
⊞ dut.PD_bank4.ret_4	
⊞ dut.PD_bank5.ret_5	
⊞ dut.PD_bank6.ret_6	
⊞ dut.PD_bank7.ret_7	
⊞ dut.PD_bank8.ret_8	
⊞ dut.PD_bank9.ret_9	

Power Intent Tab

The Power Intent tab allows you to bring up the original 1801 files and view the files as applied to the design.

The first section allows you to bring up a text window that displays the 1801 file. You can specify a filename in a new window.

The second section displays 1801 objects and attributes in a tabular format. The `lp -check ...` command returns static information of the 1801 objects, which is displayed in this section. The 1801 objects include supply nets, supply ports, supply sets, power switches, power domains, isolation, retention, pst, and state transition. For each object type, all 1801 objects are shown with their properties. Properties that are empty are not shown. You can hide some properties. Object types that do not exist in the design are greyed-out. The entire section can be collapsed.

© 2021 All Rights Reserved.

The third section displays 1801 objects and attributes in a tree format. The content of this section is same as the second section. The entire section can be collapsed.

Figure 34-5 IEEE 1801 - Power Intent

name	scope	fileName	lineno	include_scope	elements	power_up_states	power_down_states	default_power	default_ground	swi
PD_Def	dut	top.upf	4	true		high	low	dut.VDD	dut.VSS	
PD_bank0	dut	top.upf	6		bank_0	high	low	dut.VDD_sw0	dut.VSS	dut
PD_bank1	dut	top.upf	7		bank_1	high	low	dut.VDD_sw0	dut.VSS	dut
PD_bank2	dut	top.upf	8		bank_2	high	low	dut.VDD_sw1	dut.VSS	dut
PD_bank3	dut	top.upf	9		bank_3	high	low	dut.VDD_sw1	dut.VSS	dut
PD_bank4	dut	top.upf	10		bank_4	high	low	dut.VDD_sw2	dut.VSS	dut
PD_bank5	dut	top.upf	11		bank_5	high	low	dut.VDD_sw2	dut.VSS	
PD_bank6	dut	top.upf	12		bank_6	high	low	dut.VDD_sw3	dut.VSS	

You can view the contents of the 1801 files associated to the low-power-aware design.

© 2021 All Rights Reserved.

Figure 34-6 Power Intent File

Power Supply Tab

The Power Supply tab shows a graphical view of the power supply network that includes supply nets, supply ports, supply switches, and power domains. You can choose to view one supply net at a time or all of them together.

© 2021 All Rights Reserved.

Figure 34-7 IEEE 1801 - Power Supply

Isolation Tab

The *Isolation* tab enables you to perform the following operations:

- View or hide the selected power domains and corresponding isolation rules.
- View or hide the isolated pins for the selected isolation rule.
- View or hide the isolated pins' clamped values.
- View the state of isolation rules: active or inactive.

Viewing Power Domains and Isolation Rules

You can view or hide the selected power domains and the corresponding isolation rules.

© 2021 All Rights Reserved.

- Click the required power domain node (which is a child tree) in tree view and then click *Draw Selected Power Domain.* Alternatively, double-click the required node to display the corresponding isolation rules for the selected power domain in the schematics format.
- To hide a power domain's schematics, either click *Remove Selected Power Domain* or double-click the power domain node.

The following figure shows the power domain `PD_bank0` and its isolation rules:

Figure 34-8 Power Domains and Isolation Rules

© 2021 All Rights Reserved.

Viewing Hidden Isolation Rules

If number of isolation rules of one power domain is over a certain threshold, some of them will be hidden. For example, the following figure shows the schematics of `PD_bank2` with a few parts hidden:

Click the corresponding power domain node in the tree view and then click *Show Hidden Isolation Rules* or press the middle button of the mouse to view all hidden isolation rules in a new window.

The following figure shows the complete schematics for `PD_Bank2`.

Figure 34-9 Hidden Isolation Rules

© 2021
All Rights Reserved.

Viewing Hidden Power Domains

Click *Show All Power Domains* to display all power domains and the corresponding isolation rules for a given design. *Show All Power Domains* changes to *Hide All Isolation Rules*. You can also hide the displayed power domains again, if needed.

The following figure shows all power domains in a design:

Figure 34-10 Hidden Power Domains

Viewing Isolated Pins and Clamped Values

You can view the isolated pins for the selected isolation rule.

© 2021 All Rights Reserved.

Double-click an isolation rule node for a specific power domain in the tree view to display its isolated pins in the *Pins* tree view on the right pane.

The following example shows the isolated pins for the `Iso2` isolation rule for the `PD_bank2` power domain.

Figure 34-11 Isolated Pins

Viewing Clamped Values

Click *-->Show Clamped Values* to display the isolated pins' clamped values in the schematics. The button changes to *<--Hide Clamped Values*. You can hide the displayed clamped values again, if needed.

The following example shows the clamped values for all isolation rules for all power domains:

© 2021 All Rights Reserved.

Figure 34-12 Clamped Values

Viewing State of Isolation Rules

Click the *Active* column to check whether isolation rules are active or not. The active rules are show in red color in the schematics.

The following examples show the state of isolation rules in tree view and schematic view formats:

© 2021 All Rights Reserved.

Figure 34-13 Isolation Rules State - Tree View

© 2021 All Rights Reserved.

Figure 34-14 Isolation Rules State - Schematics

Viewing Power Information for Probes in Xcelium SimVision GUI

In the SA mode, you can view the power information using the SimVision tool. It provides a Power Display sidebar that displays the power-domain information and supports backward annotation in Design Browser, Source Browser, Waveform window, Schematic Trace and Power Supply Network windows. To open the Power Display sidebar, click the Power tab in the required SimVision window.

To view this information in the SimVision GUI, you need to execute the `probe -power` command explicitly after swapping the design into hardware, if you did not issue this command before swapping the design. If you open the Power Display sidebar before swapping the design into hardware, you do not need this explicit execution of the `probe -power` command.

By default, the initial view shows the top level of the power-domain hierarchy. Click the plus icon to the left of the top level of the hierarchy to list its power domains. The power domains

© 2021 All Rights Reserved.

are organized by scope in the hierarchy. The light bulb next to each domain name indicates its state (powered up, powered down, or unknown) at the selected waveform time.

Figure 34-15 Power Display Sidebar in SimVision

Expand a power domain to display the items for the instances, isolation ports, isolation strategies, and retention strategies in that domain. You can further expand these items to see their contents. The icons to the left of isolated ports show the isolation state. The possible isolated states are: *in effect with isolated value as high*, in effect with isolated *value as low*, *in effect with isolated value as hold*, *not in effect*, etc.

When you click on isolated ports in the Power Display sidebar of the Waveform window, those ports are displayed in Waveform window. When isolation is in effect, slanted lines background is shown in the isolated port.

The signals in the powered-down power domains are marked with a cross-hatched background in the Waveform window.

© 2021 All Rights Reserved.

Figure 34-16 Isolated Ports and Signals in Powered-Down Power Domains in Power Display Sidebar

If you move the timeline in the Waveform window, you can see the backward annotation in the Design Browser, Schematic Browser, and Power Supply Network windows.

A power switch in the Power Supply Network window is grayed out if the output supply is off. When dpa sparse sampling is enabled in the SA mode, certain points will not have transition values. So, the Power Supply Network window might not have back annotation and the power switches are not grayed out when the time axis in waveform locates in the power-off regions.

Simulating with xrun

If you are running the simulator in single-step invocation mode with xrun, specify all command-line options and source files with the `xrun` command, as follows:

```
% xrun -lps_1801 filename -lps_simctrl_on \
[-lps_options] \
[other_options] \
source_files
```


© 2021 All Rights Reserved.

Specifying Power Back Initialization Values

IEEE 1801 does not specify power-loss values and power back initialization values for sequential elements. Use the `-power_up_states` and `-power_down_states` options of the `create_power_domain` command to affect the values of state devices that do not use retention. These two options are not defined in the IEEE 1801.

For information about the description and valid values of the `-power_up_states` and `-power_down_states` options, refer to the description of the `create_power_domain` command in the *Common Power Format Commands* chapter of the *VXE Command Reference Manual*.

Memory Power Shut-Off Randomization

When `-power_down_states` is specified as `random`, the output of FFs or latches during power-off state randomly returns `0` or `1`. Memory instance returns random values as memory instance content when the domain is in power-off state. If `-power_up_states` is specified as `random` during power-up state and a volatile memory is not restored by the corresponding retention strategies, it returns random values. The retention strategy is defined by the `set_retention` command and applied to the power domain where there are sequential elements. Once the power is back, the non-volatile memory will have the values as before the power shut-off.

Isolation

Palladium inserts isolation instrumentation logic per isolation strategies location option. Generally, isolation is inserted before any optimization is done.

Using Wildcards in IEEE 1801 Commands

Palladium supports use of wildcards in IEEE 1801 commands for referencing design objects.

The *<namePattern>* in the following commands can include "*" and "?" wildcards.

```
lp -check <1801_Command> <namePattern>
lp -check {isolation | retention | domain} <namePattern>
lp -value {domain | retention | isolation | supplyport | supplynet |\
           supplyset | pst} <namePattern>
probe {-addpowerdomain | -addisolation | -addretention | -addsupplyport |\
       -addsupplynet | -addsupplyset | -addswitch | -addportstate |\
       -addpststate} <namePattern>
```


© 2021 All Rights Reserved.

Note: In 1801 files, wildcards cannot be used to refer to the design objects (logic nets, logic ports, power domains, supply sets, and so on) created by the 1801 `create_...` commands.

- The "`.`" and "`/`" delimiter characters are used as meta characters, and treated as separators in the UPF wildcard strings. Ensure that you do not combine "`.`" and "`/`" with wildcards. Consider the following example:

  `U_out1.OUT.L*`

  In the above example, "`.`" is used in combination with the * wildcard. So, use `OUT?L*` instead of `OUT.L*`, as follows:

  `U_out1.OUT?L*`" instead of "`U_out1.OUT.L*`".

- Following are some limitations in using wildcards for generate blocks:

  - In UPF files, square braces in generate blocks should not be used in pattern of the path. Consider the following example for the `create_power_domain` command:

    ```
    create_power_domain PD_8A -elements {mid_INST8/GEN_up[3]/INST_*}
    ```

    In the above statement, `mid_INST8/GEN_up[3]/INST_*` might lead to compilation failure. To work around the issue, replace the square braces with "`*`", to specify the index of generate blocks, as follows:

    ```
    {create_power_domain PD_8A -elements mid_INST8/GEN_up*3*INST_*}
    ```

    Note: The above workaround is not supported by the Xcelium simulator.

  - When using wildcards to match generate block names, ensure that you do not use "`.`" along with "`*`" because Palladium does not support wildcards across cross-hierarchical boundary. Use the <*string*>*<*string*> pattern instead of <*string*>*.<*string*> in generate block names.

    Consider the following patterns:

    ```
    top_inst/genblkmux*.mux/check
    top_inst.genblkmux*.mux.check
    ```

    `check` is a signal of the `mux` instance under the `genblkmux` generate block loop of the `top_inst` instance.

    To avoid errors, change the patterns as follows:

    - Replace `top_inst/genblkmux*.mux/check` with `top_inst/genblkmux*mux/check`
    - Replace `top_inst.genblkmux*.mux.check` with `top_inst.genblockmux*mux.check`

© 2021 All Rights Reserved.

Using 1801 Boolean Expressions

In VXE, a boolean expression can reference HDL names and values. The name can be a vector name. The real number can be used for voltage values. The boolean expression can also reference IEEE 1801 state names and values.

VXE does not support `Time` for use with the interval function.

The following table lists the supported 1801 operators:

1801 Operator	SystemVerilog Equivalent	VHDL Equivalent	Meaning
!	!	not	Logical negation
~	~	not	Bit-wise negation
<	<	<	Less than
<=	<=	<=	Less than or equal
>	>	>	Greater than
>=	>=	>=	Greater than or equal
==	==	=	Equal
!=	!=	/=	Not equal
&	&	and	Bit-wise conjunction
^	^	xor	Bit-wise exclusive disjunction
|	!	or	Bit-wise disjunction
&&	&&	and	Logical conjunction
||	||	or	Logical disjunction

The following is the list of IEEE 1801 commands that use boolean expressions.

- `add_power_state -supply_expr -logic_expr`
- `create_power_switch -on_state -off_state -on_partial_state -error_state`
- `set_retention -save_condition -restore_condition -retention_condition`

© 2021 All Rights Reserved.

Modeling Non-Standard State Retention Registers

VXE enables you to define the following non-standard state retention behaviors: *ARM Retention* and *TSMC Retention*.

The following terms apply for a retention register:

- *Register value* is the data held in the storage element of the register. When power is on, this value gets updated on the rising or falling edge of the clock or gets set or cleared by `set` and `reset` signals, respectively.
- *Retained value* is the data in the retention element of the retention register. The retention element is powered by the retention supply.
- *Output value* is the value on the output of the register.

Prerequisites

You must first model a retention rule using either the 1801 command `set_retention` or `set_retention_control`, or both the commands together. The condition is specified using the following options:

- `-save_signal expr`
- `-restore_signal expr`

For details on the `-save_signal` and `-restore_signal` options, refer to the *IEEE 1801 LRM*.

The following sections explain how to enable the non-standard retention behaviors in VXE:

- ❑ Modeling Special ARM Retention Cells
- ❑ Modeling TSMC Retention Cells

Modeling Special ARM Retention Cells

You can model retention behavior in a low power design based on the special `ARM` retention cell behavior model.

The special `ARM` retention cell has an active high asynchronous reset `R` and an active low asynchronous set `SN` pins, input pin `D`, a single save or restore pin `RETN`, clock pin `CK`, output pin `Q`. During the period between power back time point to retention restore time point, the output `Q` is not always in power up state, instead, when you set `SN` from high to low, `Q` is `1`, when you reset `R` from low to high, the `Q` is `0`.

© 2021 All Rights Reserved.

Enabling ARM Retention

To enable ARM retention, use the `ret_cell_type ARM_RETN_CELL_1` attribute of the UPF `set_design_attribute` command:

```
set_design_attributes -elements {<retention_strategy_list>}
                      -attribute ret_cell_type ARM_RETN_CELL_1
```

The *<retention_strategy_list>* is a list of retention strategies specified in the `set_retention` command. The name can be a hierarchical name related to the current scope. In VXE, the wildcards `*` and `?` can be used.

The `ARM_RETN_CELL_1` is an Xcelium reserved key. Both Palladium and Xcelium verify if a retention strategy is a qualified `ARM_RETN_CELL_1` type. If a retention strategy is not a qualified `ARM_RETN_CELL_1` type, the `ARM_RETN_CELL_1` attribute is not applied to that retention strategy with the following warning:

```
The retention 'DUT_UART_RET' is not a qualified ARM_RETN_CELL_1 type. The save
signal sense is posedge. Ignore the attribute.
```

The rules to verify if a retention strategy is a qualified `ARM_RETN_CELL_1` type are as follows:

- The saved signal sense should not be posedge or negedge
- The restore signal sense should not be posedge or negedge
- The save condition should not exist.
- The restore condition should not exist.
- The retention condition should not exist.
- The save and restore signal should exist.

You can verify if a retention strategy has the `ARM_RETN_CELL_1` attribute by executing the `lp -check set_retention <retentionName>` command, which will show `+set_reset_retn_cell` as true.

For example,

```
XEC> lp -check set_retention
retention {
-name rule_retn -fileName {top.upf top.upf} -lineno {33 39}  -domain domain_pso  -
elements {  u_sr_upf.Q_UPF}  -save_signal RETN high  -restore_signal RETN low
-restore_condition !CK  -transitive true  +set_reset_retn_cell true  -
retention_power_net VDD  -retention_ground_net VSS }
```

Supported Options for ARM Retention Cells

VXE supports the following options for ARM retention cells:

© 2021 All Rights Reserved.

- The design attribute `ret_cell_type` with value `ARM_RETN_CELL_1` is supported with the UPF `set_design_attributes` command.
- The `read_power_intent` command accepts the `ret_cell_type` attribute and passes it to precompile. precompile instruments the retention rule as ARM retention cell if a retention rule has the `ARM_RETN_CELL_1` attribute.
- The `lp -check set_design_attributes` command shows the `ARM_RETN_CELL_1` attribute value.
- The `lp -check set_retention` command shows the `ARM_RETN_CELL_1` type information.

Modeling TSMC Retention Cells

The state retention cells cannot be modeled using IEEE 1801 standard. However, VXE supports a custom state retention model called *TSMC State Retention (SR)* model for Verilog designs.

In the TSMC 10 nm process, there exist state retention cells that restore state when power is restored. During the retention cycle, when the retention signal is active, the output of the TSMC SR model remains locked at the restored value.

In the existing ARM State Retention model, the clock signal toggles during the retention cycle and the output is impacted by the `clock`, `set` or `reset` signals during the retention cycle.

Unlike the ARM State Retention model, the output of the TSMC State Retention model is not effected by the `clock`, `set` or `reset` signals during the retention cycle.

Enabling TSMC SR Model

To enable the TSMC SR model, use the `ret_cell_type tsmc_retn_cell_1` attribute of the UPF `set_design_attribute` command:

```
set_design_attributes -elements {<retention_strategy_list>}
                      -attribute ret_cell_type tsmc_retn_cell_1
```

The `<retention_strategy_list>` is a list of retention strategies specified in the `set_retention` command. The name can be a hierarchical name related to the current scope. In VXE, the wildcards `*` and `?` can be used.

For example,

```
set_design_attributes -elements {PD_PSO.rule_retn} -attribute ret_cell_type
tsmc_retn_cell_1
```


© 2021 All Rights Reserved.

- Power domain and State retention rule: `{PD_PSO.rule_retn}`
- Attribute required to enable TSMC SR model: `-attribute ret_cell_type tsmc_retn_cell_1`

The rules to verify if a retention strategy is a qualified `tsmc_retn_cell_1` type are as follows:

- The saved signal sense should not be posedge or negedge
- The restore signal sense should not be posedge or negedge
- The save condition should not exist.
- The restore condition should not exist.
- The retention condition should not exist.
- The save and restore signal should exist.

You can verify if a retention strategy has the `tsmc_retn_cell_1` attribute by executing the `lp -check set_retention <retentionName>` command.

For example,

```
XEC> lp -check set_retention
```

`+retn_cell_type TSMC_retention_cell1` will be shown in the output if the corresponding retention is applied to the `tsmc_retn_cell_1` type:

```
lp -check set_retention
retention {
-name ret_0
-scope dut -fileName top.upf -lineno 46
-domain dut.PD_bank_0
-elements {
{dut.bank_0.INST_mid1.Q_ARM[0:7] }
{dut.bank_0.INST_mid2.Q_ARM[0:7] }
{dut.bank_0.INST_out.Q_ARM[0:7] }}
-save_signal dut.RETN high
-restore_signal dut.RETN low
-transitive true
+retn_cell_type TSMC_retention_cell1
-retention_power_net dut.VDD
-retention_ground_net dut.VSS
+version 2.1
}
```

The following timing diagram shows the required operation for the TSMC SR model. The state saved when the `RETN` is `low`. The state is restored when power is turned on.

© 2021 All Rights Reserved.

Figure 34-17 Timing Diagram

Note: The output `Q_TSMC`, is restored on power up. `clock`, `set`, and `reset` are ignored. The output, `Q_TSMC`, remains locked and are not impacted by `D` (data), `CD` (reset) or `SDN` (set)) during the retention cycle (`RETN = 0`).

The above diagram illustrates how the state is saved when `RETN` is low, and the value is restored when power is switched on again. The value of `Q` is locked during the retention cycle, and is not affected by the clock, the input signal (`D`), or the changing `set` and `reset` signals.

Supported Options for TSMC Retention Cells

VXE supports the following options for TSMC retention cells:

- The design attribute `ret_cell_type` with value `tsmc_retn_cell_1` is supported with the UPF `set_design_attributes` command.
- The `read_power_intent` command accepts the `ret_cell_type` attribute and passes it to precompile. precompile instruments the retention rule as TSMC retention cell if a retention rule has the `tsmc_retn_cell_1` attribute.
- The `lp -check set_design_attributes` command shows the `tsmc_retn_cell_1` attribute value.
- The `lp -check set_retention` command shows the `tsmc_retn_cell_1` type information.

© 2021 All Rights Reserved.

Modeling Cells Using NOR Isolation Strategy

You can model isolation behavior in a low power design based on the special `NOR` isolation cell behavior model.

In a normal isolation cell, when the isolation supply is off, the output of the isolation cell is unknown. However, in the `NOR` isolation cell, when the isolation supply is off, the output is `0`.

To specify which isolation strategies follow `NOR` isolation cell behavior, you can use the `set_design_attributes` command to indicate the names of those `NOR` related isolation strategies as follows:

```
set_design_attributes -elements {<isolation_strategy_list>}
                      -attribute iso_nor TRUE | FALSE
```

The `<isolation_strategy_list>` is a list of isolation strategies specified in the `set_isolation` command. The name can be a hierarchical name related to the current scope. In VXE, the wildcards `*` and `?` can be used.

Performing Dynamic Power Analysis for Designs with 1801 Files

IEEE 1801 PSO behavior affects DPA power analysis and power estimation reports.

- In NTC (Native Toggle Count) or WTC (Weighted Toggle Count) flow, the instance toggle or high value is set as `0` when the primary supply of the instance is off.
- In Palladium-Joules flow, the power shut-off is processed by Joules. Joules provides accurate power number based on the status of the power domains.
- For macro or black box, the primary supply of the instance is based on its inputs' primary supply. For a normal design instance, its primary supply is determined by its power domain.

Controlling Assertions for Powered Down Domains

In the SA mode, the `create_assertion_control` command enables you to specify alternate assertion behavior when domains are powered down in an IEEE 1801 environment.

The `create_assertion_control` command is not supported in the legacy ICE flow.

When assertions are selected by a `create_assertion_control` directive, the assertions are disabled based on the parent power domain or the assertion control condition. To disable

© 2021 All Rights Reserved.

the assertions, use either the `-assertion_control` or the `-supply_set` option with the `create_assertion_control` command.

While powered down, the assertion is insensitive to any fanin signal activity. On power up, the assertions are re-started in the initial state – that is, the internal state of the assertion at power down is cleared on power up.

If an assertion is powered down on the simulator, the assertion remains in a powered up state – reflected in its activity, waveforms, and values – until the assertion is swapped to the emulator at which point it is powered down.

Similarly, when an assertion that is powered down on the emulator is swapped back to the simulator, it becomes powered up for the duration of the simulator interval.

For a detailed description of the `create_assertion_control` command, refer to the *Xcelium Documentation Set*.

In VXE, the Xcelium behavior of the `create_assertion_control` command is mirrored whenever applicable. This includes the following:

- The `'-type reset'` internal state is reset on power up, that is, it does not display failure expected if state is not reset.
- The `'-type reset'` does not clear the checked, finished, or failed counter values are preserved when powered up.
- Assertion waveform and text values are `'inactive'` when powered down.
- The `-onFailure` limit and failure count towards the limit are preserved through power down and power up events.
- When assertions are powered down, no assertion events are recorded even if all fanin signals are active (that is, OOMRs to powered-up domain).
- Assertion enable or disable state is preserved through the assertion power down or power up state.
- The `create_assertion_control` directives have no effect on protected assertions.
- When a disabled assertion is powered down, assertion remains in the `'off'` state. If the assertion is enabled during power down, it reverts to the `'inactive'` state.
- If an assertion is disabled while powered down, it remains in the `'inactive'` state until powered up and then transition to the `'off'` state.
- Assertion `'inactive'` state and `'-type reset'` handling are preserved through save and restore.

© 2021 All Rights Reserved.

The syntax errors are identified at compile time while executing the `read power intent` command. For example,

```
ERROR (legacy-131313): The specified create_assertion_control '-domain' <val> has
not been defined.
ERROR (legacy-131313): The specified create_assertion_control '-assertions' <val>
does not exist.
```

Logging IEEE 1801 Events

IEEE 1801 enables you to do 1801 low power emulation and co-simulation. While the emulation or co-simulation is running, you can see a message when some 1801 related events happen. For example, a message appears when there is a power domain state change.

In VXE, 10 types of 1801 events are logged. The events are: power domain, supply set, isolation, retention, power state, state transition, port state, pst state, illegal isolation state, and illegal retention state.

This section includes the following:

- Format of Logging Messages
- Showing 1801 Event Messages
- Logging 1801 Event Messages in Streaming Log Mode
- Logging 1801 Event Messages in Buffered Log Mode

Format of Logging Messages

You can specify part of the 1801 event types to be captured in memory buffer by using the following command:

```
compilerOption -add {1801LogType {[powerDomain] [supplySet] [isolation]
                     [retention] [powerState] [stateTransition] [portState]
                     [pstState] [illegalIsoState] [illegalRetState]}}
```

The logging messages include the time when the event happens and the state of related power object.

For example,

```
INFO 1801LOG: At cycle 1299, power domain dut.PD2 changed from NORMAL to CORRUPT.
INFO 1801LOG: At cycle 1299, isolation strategy dut.ISO1 is ENABLED.
```


© 2021 All Rights Reserved.

In SA mode, the time is shown as simulation time instead of cycle, such as `At 1022ns`. The logging message only contains power objects whose state changed. The message is saved in the `xe.msg` file whenever it is printed on screen. The message is prefixed with text string `1801LOG:` to enable you to `grep` the messages from the `xe.msg` file.

Showing 1801 Event Messages

You can control the output of 1801 event messages using the following commands:

- `lp -log -console 0|1`

  If `1` is specified, all power event messages are printed on screen and saved to the `xe.msg` file.

  If `0` is specified, no power event messages are printed on screen or saved to the `xe.msg` file.

- `lp -log -file <filename>`

  All power event messages are written into `<filename>`. If `<filename>` equals to `""` or `{}`, no messages are written to the specified file `<filename>`.

These two commands work independently. If both are set, power event messages are not only printed on screen and saved to `xe.msg` but also written to the specified `<filename>`. If both are not specified, by default power event messages are printed on screen and saved to the `xe.msg` file.

You can view 1801 event message not only in text format but also in SST2 or FSDB waveform format. A waveform file can be generated by using the `lp -log [-shm <filename>] | [-fsdb <filename>]` command before executing the `run` command.

You can also specify which type of 1801 event should be printed or saved to the file by using the following command:

```
lp -log -eventType {[powerDomain] [supplySet] [isolation] [retention] [powerState]
[stateTransition] [portState] [pstState] [illegalIsoState] [illegalRetState]}
```

To limit the scope of 1801 event messages, you can issue the `lp -log -scope <scope_name>` command.

Logging 1801 Event Messages in Streaming Log Mode

With the streaming mode 1801 event messages are logged continuously. You can get the complete data only in STB mode and STA mode in which emulation or co-simulation can be paused. Whenever the hardware memory buffer fills up the emulation or co-simulation is

© 2021 All Rights Reserved.

paused and all event data is uploaded and printed on screen and/or saved to `xe.msg` or the specified file. Then emulation or co-simulation is resumed.

To invoke the streaming mode, issue the `lp -log -start` command. Then, you can issue the `run` command to start emulation or co-simulation. Logging messages are printed on screen and/or saved to `xe.msg` or the specified file automatically while design is running. The streaming mode can be stopped by using the `lp -log -stop` command. The remaining event data is uploaded and printed automatically under this circumstance. If you exit xeDebug, the software also automatically uploads the remaining event data and prints on screen and/or saves the date to `xe.msg` or the specified file. You can use the `lp -log -console 0|1` and `lp -log -file <filename>` commands to specify the output of logging message.

In LA mode and IXCOM dynamic mode, the emulation or co-simulation cannot be paused, Therefore the memory buffer might overflow when the speed of the 1801 events generation is faster than the speed of reading and decoding the 1801 events. Use the `lp -log -overflow ignore|warning|error` command to specify how to handle such overflow. The remaining event data in memory buffer is uploaded once the design stops running.

You can dynamically change between the LA and STB modes in ICE mode or change between the STA and DYN modes in SA mode by using the `configPM` command. No event data is lost when you change the runtime mode. Because the remaining 1801 event data is uploaded automatically after each `run` command.

Logging 1801 Event Messages in Buffered Log Mode

Buffered mode is enabled by default unless you switch to streaming mode by using the `lp -log -start` command. The hardware memory buffer always captures low power events. Because the size of the hardware memory buffer is limited (you can set the depth of the memory buffer with the `compilerOption -add {1801LogBufferDepth <n>}` command at compile time), once the memory buffer is full, the newest data overwrites the oldest data without generating any warnings.

The event data is uploaded from the buffer and printed only when you explicitly request to upload the data through the `lp -log -upload [-num_sample <n>] | [[-start <startTime>] [-end <endTime>]]` command. If you exit xeDebug or switch to streaming mode, the event data is not uploaded automatically.

This mode is useful in LA and DYN mode when streaming mode causes buffer overflow. It is also useful when you do not want to see the 1801 event message until a trigger happens. At that moment you can see the low power events that happened before or around the trigger.

© 2021 All Rights Reserved.

Handling IEEE 1801 Illegal Events

The system enables you to detect illegal low power events at run time.

Xcelium has low power assertions that check for basic correct low power sequences, for example, isolation must occur before retention and shutoff. (These Xcelium assertions are enabled with the xmelab option `-lps_verify`.) VXE provides similar capability with instrumentation that ties into the emulator control.

When illegal events occur, the events might trigger the emulator to stop and/or be reported. You might disable any or all of these illegal event checking.

Instrumentation is created to check for illegal events for each isolation or retention rule, only if it is appropriate and applicable. Besides Xcelium-similar illegal events, the IEEE 1801 illegal events in VXE also include illegal events defined in the IEEE 1801 files, such as illegal power states, state transitions, and error states for power switches.

In VXE, the IEEE 1801 illegal event handling is not implemented as PSL or SVA assertions, therefore, the illegal events are not managed as assertions. However, the illegal event names reference the normal state and are considered to have failed on the event.

Illegal event handling works in both ICE and SA modes with stoppable DUT, that is, it does not work in LA mode. Hot-swap does not transfer any illegal event information from software to hardware.

IEEE 1801 Illegal Condition Checks

A comprehensive list of the names of the IEEE 1801 illegal condition checks in VXE is mentioned in the following sections:

- Isolation
- Retention
- Power State
- State Transition
- Switch State

Note: The isolation name is the hierarchical domained name, for example, tb.dut.pdA.iso1, whereas the corresponding Xcelium name has only the simple name, for example, iso1, or with an explicitly specified domain and underscore separator.

© 2021 All Rights Reserved.

Isolation

The 1801 illegal condition checks for isolation are as follows:

- `LPV_ISO_<isolation>_SIGNAL_NOT_X_Z_U`

  Corresponding Xcelium name: `LPV_ISO_<isolation>_SIGNAL_NOT_X_Z_U`

  The illegal condition check fails when one or more of the control signals for this isolation strategy is corrupt. This check is not created if the control signals are located in an always-on (AON) domain.

- `LPV_ISO_<isolation>_ENABLED_WHEN_INPUT_DRIVER_CORRUPT`

  Corresponding Xcelium name: none

  The illegal condition check fails when the isolation strategy is inactive but one or more inputs to its domain is corrupt. This check is not created if all the inputs to its power domain are driven from an AON domain.

- `LPV_ISO_<isolation>_ENABLED_WHEN_DOMAIN_CORRUPT`

  Corresponding Xcelium name: none

  The illegal condition check fails when the isolation strategy is inactive but its domain is shut off. This check is not created if the power domain has no outputs or is located in an AON domain.

- `LPV_ISO_<isolation>_DOES_NOT_TOGGLE_DURING_TARGET_NORMAL`

  Corresponding Xcelium name:
  `LPV_ISO_<isolation>_DOES_NOT_TOGGLE_DURING_<supply set>_NORMAL`

  The illegal condition check fails when the isolation strategy turns on and off without shutting off its domain power. This check is not created if its power domain is located in an AON domain.

- `LPV_ISO_<isolation>_WHILE_ON_TARGET_DOES_NOT_TOGGLE`

  Corresponding Xcelium name: `LPV_ISO_<isolation>_WHILE_ON_<supply set>_DOES_NOT_TOGGLE`

  The illegal condition check fails when the isolation strategy turns on and stays on through to a second power shut off of its domain. This check is not created if its power domain is located in an AON domain.

© 2021 All Rights Reserved.

Retention

The 1801 illegal condition checks for retention are as follows:

- `LPV_DMN_<retention>_RETENT_COND_NOT_X`

  Corresponding Xcelium name: none

  The illegal condition check fails when any of the signals in the save, restore or retention condition of the retention strategy, or the save or restore signal are corrupt. This check is not created if its save or restore signals and its save, restore or retention condition signals are driven from an AON domain.

- `LPV_DMN_<retention>_SAVE_SIGNAL_NOT_X`

  Corresponding Xcelium name:
  `LPV_DMN_<domain>_<retention>_SAVE_SIGNAL_NOT_X`

  The illegal condition check fails when the save signal of the retention strategy is corrupt or the save signal becomes active while any signal in the save condition is corrupt. This check is not created if its save signal is driven from an AON domain or if the retention type is master/slave alive.

- `LPV_DMN_<retention>_RESTORE_SIGNAL_NOT_X`

  Corresponding Xcelium name:
  `LPV_DMN_<domain>_<retention>_RESTORE_SIGNAL_NOT_X`

  The illegal condition check fails when the restore signal of the retention strategy is corrupt or the restore signal becomes active while any signal in the restore condition is corrupt. This check is not created if its restore signal is driven from an AON domain or if the retention type is master/slave alive.

- `LPV_DMN_<retention>_ALWAYS_SAVE_BEFORE_RESTORE`

  Corresponding Xcelium name:
  `LPV_DMN_<domain>_<retention>_ALWAYS_SAVE_BEFORE_RESTORE`

  The illegal condition check fails when the retention strategy restore event occurs and the restore condition is true but the strategy does not have valid saved values, for example, because the save event does not occur or the saved values are corrupted. This check is not created if the retention type is master/slave alive because save occurs on power off and restore occurs on power back for this retention type.

- `LPV_DMN_<retention>_RESTORE_AFTER_SHUTOFF`

  Corresponding Xcelium name:
  `LPV_DMN_<domain>_<retention>_RESTORE_AFTER_SHUTOFF`

© 2021 All Rights Reserved.

The illegal condition check fails when the retention strategy save event occurs when it already has valid saved values, for example, because there is no intermediate restore event after a prior save. This check is not created if the retention type is master/slave alive because save occurs on power off and restore occurs on power back for this retention type.

- `LPV_DMN_<retention>_NO_SAVE_WHILE_SHUTOFF`

  Corresponding Xcelium name:
  `LPV_DMN_<domain>_<retention>_NO_SAVE_WHILE_SHUTOFF`

  The illegal condition check fails when the retention strategy save event occurs when its domain has power shutoff. This check is not created if the retention type is master/slave alive because save occurs on power off and restore occurs on power back for this retention type.

- `LPV_DMN_<retention>_NO_RESTORE_WHILE_SHUTOFF`

  Corresponding Xcelium name:
  `LPV_DMN_<domain>_<retention>_NO_RESTORE_WHILE_SHUTOFF`

  The illegal condition check fails when the retention strategy restore event occurs when its domains has power shutoff. This check is not created if the retention type is master/slave alive because save occurs on power off and restore occurs on power back for this retention type.

- `LPV_DMN_<retention>_SHUTOFF_FOLLOWS_SAVE`

  Corresponding Xcelium name:
  `LPV_DMN_<domain>_<retention>_SHUTOFF_FOLLOWS_SAVE`

  The illegal condition check fails when the domain enters power shutoff but the retention strategy does not have valid saved values, for example, because there is no prior save event. This check is not created if the retention type is master/slave alive because save occurs on power off and restore occurs on power back for this retention type.

- `LPV_DMN_<retention>_SHUTOFF_BEFORE_RESTORE`

  Corresponding Xcelium name:
  `LPV_DMN_<domain>_<retention>_SHUTOFF_BEFORE_RESTORE`

  The illegal condition check fails when the retention strategy save event occurs twice without an intermediate power cycle. This check is not created if the retention type is master/slave alive because save occurs on power off and restore occurs on power back for this retention type.

- `LPV_DMN_<retention>_RETAINED_VALID_UNTIL_RESTORE`

© 2021 All Rights Reserved.

Corresponding Xcelium name: none

The illegal condition check fails when the retention strategy retained values get corrupted. For example, with a keep-alive type of retention, this check fails if the retention condition is true at power-off, that is, the save condition occurred, but it goes to false before power-back, so the saved values are destroyed. Another example is if the retention supply becomes corrupt.

Power State

The 1801 illegal condition checks for power state are as follows:

- `LPV_STATE_<power state>_ SIGNAL_NOT_X_Z_U`

  Corresponding Xcelium name: none

  The illegal condition check fails when any signal in the logic or supply expression of the power state is corrupt. This check is not created if all the signals in the logic or supply expression are driven from an AON domains.

- `LPV_STATE_<power state>_ILLEGAL_DOES_NOT_OCCUR`

  Corresponding Xcelium name: none

  The illegal condition check fails when the power state that you have defined as illegal occurs.

State Transition

The 1801 illegal condition check for state transition is as follows:

- `LPV_TRANSITION_<state transition>_ILLEGAL_DOES_NOT_OCCUR`

  Corresponding Xcelium name: none

  The illegal condition check fails when the state transition that you have defined as illegal occurs.

Switch State

The 1801 illegal condition check for switch state is as follows:

- `LPV_SWITCH_<power switch>_ERROR_STATE_DOES_NOT_OCCUR`

  Corresponding Xcelium name: none

© 2021 All Rights Reserved.

The illegal condition check fails when the error state of the power switch occurs.

Unsupported Xcelium Assertions

This section lists the Xcelium assertions that are not supported in VXE.

- The following two assertions are similar to each other and served by `LPV_ISO_<isolation>_ENABLED_WHEN_DOMAIN_CORRUPT`.

  `LPV_ISO_<iso_rule_name>_ENABLED_WHEN_<supply_set_name>_NOT_NORMAL`

  `LPV_ISO_<iso_rule_name>_DISABLED_WHEN_<supply_set_name>_NOT_NORMAL`

- The following assertion is not applicable for VXE because VXE implementation only checks save or restore conditions at the event edge.

  `LPV_DMN_<domain>_<retention>_SAVE_AND_RESTORE_NOT_ACTIVE`

- The following assertion is not applicable for VXE because VXE implementation only checks the save condition at the save signal edge.

  `LPV_DMN_<domain>_<retention>_SAVE_COND_NOT_FALSE`

- The following assertion is not applicable for VXE because VXE implementation only checks the restore condition at the restore signal edge.

  `LPV_DMN_<domain>_<retention>_RESTORE_COND_NOT_FALSE`

Messages

You can use a TCL command to get the description for each assertion type. The assertion type is the assertion name without the 1801 object and extra underscore.

Below is an exhaustive list of the messages, with an example of a TCL command for formatting for readability. The message asserts what has happened (which is the failure of the check), that is, while the name is asserting what should be the normal condition, the message is written as the failure condition.

```
XEsim> set i 0; set odd 0; foreach item [lp -illegal * -describe] {set odd [expr
!$odd]; if {$odd} {incr i; puts -nonewline "$i - "}; puts "$item"}
    1 - LPV_ISO_SIGNAL_NOT_X_Z_U
    One or more of the isolation rule's controls are corrupted.
    2 - LPV_ISO_ENABLED_WHEN_INPUT_DRIVER_CORRUPT
    Isolation is not applied but one or more inputs are being driven from a domain
    in the corrupted state.
    3 - LPV_ISO_ENABLED_WHEN_DOMAIN_CORRUPT
    Isolation is not applied but the domain is powered off.
    4 - LPV_ISO_WHILE_ON_TARGET_DOES_NOT_TOGGLE
    Isolation is applied but the supply set has toggled more than once.
```


© 2021 All Rights Reserved.

```
5 - LPV_ISO_DOES_NOT_TOGGLE_DURING_TARGET_NORMAL
The supply set did not power cycle during the time interval when isolation was
applied and then removed.
6 - LPV_DMN_SAVE_SIGNAL_NOT_X
The save signal is corrupt, or the save signal became active while the save
condition is corrupt.
7 - LPV_DMN_RESTORE_SIGNAL_NOT_X
The restore signal is corrupt, or the restore signal became active while the
restore condition is corrupt.
8 - LPV_DMN_RETENT_COND_NOT_X
One or more of the retention condition signals are corrupted.
9 - LPV_DMN_ALWAYS_SAVE_BEFORE_RESTORE
The restore event occurred without valid previously saved values.
10 - LPV_DMN_RESTORE_AFTER_SHUTOFF
A second save event occurred without a restore event after the first save.
11 - LPV_DMN_NO_SAVE_WHILE_SHUTOFF
The save event occurred while the supply set is powered off.
12 - LPV_DMN_NO_RESTORE_WHILE_SHUTOFF
The restore event occurred while the supply set is powered off.
13 - LPV_DMN_SHUTOFF_FOLLOWS_SAVE
Shutoff happened without an earlier save event.
14 - LPV_DMN_SHUTOFF_BEFORE_RESTORE
A second save event occurred without a power cycle after the first save.
15 - LPV_DMN_RETAINED_VALID_UNTIL_RESTORE
The retained values are corrupted because of the retention condition going
false or the retention supply powering off.
16 - LPV_STATE_SIGNAL_NOT_X_Z_U
One or more of the signals in the power state's logic or supply expression is
corrupted.
17 - LPV_STATE_ILLEGAL_DOES_NOT_OCCUR
The defined illegal power state has occurred.
18 - LPV_TRANSITION_ILLEGAL_DOES_NOT_OCCUR
The defined illegal state transition has occurred.
19 - LPV_SWITCH_ERROR_STATE_DOES_NOT_OCCUR
The defined power switch error state has occurred.
```

Enabling UPF 1.0 Isolation Semantics

In the SA mode, VXE supports the `-lps_v10iso` option of Xcelium with the `xrun` command.

This option changes the default behavior to the following:

© 2021 All Rights Reserved.

- A domain interface is used if the only ports considered for isolation are those specified on the domain interface where the LowConn side of the port is in the extent of the domain.
- When neither the `-applies_to` option nor the `-elements` option is specified, `-applies_to` outputs are assumed.

To enable the domain interface concept in the ICE mode, use the `precompileOption -add {domainInterfaceDef 1.0}` command while compiling the low-power designs in the ICE mode. To set the default value of the `-applies_to` option of the `set_isolation` command to UPF `1.0`, execute the following command: `read_power_intent <1801filename> -defaultAppliesToOutputs 1.0`.

Locating Encypted Design Instances

You can encypt the instance names in the low-power designs. You can specify the instances' original signal names in the 1801 files. You can also use the wildcards in the 1801 files for the original signal names. When you issue the `read_power_intent` command at compile time to read the 1801 files, the command automatically locates the original signals from the 1801 files for the corresponding encrypted instances in the design.

Identifying Partial Power-Aware Modules and Handling Corruptions

If an RTL module has VDD or VSS power pins, and a power model to calculate the outputs based on the power pins values, the RTL model is called a power-aware model. If an RTL model has VDD or VSS power pins, and the power pins do not have load in the RTL module definition, the RTL model is called a partial power-aware model.

The power corruption instrumentation logic is not inserted for power-aware and partial power-aware models.

Example: RTL Code for Non-Power Aware Design:

```
module AND(Y,A,B);
input A, B;
output Y;
reg Y;
always @(A or B)
Y <= A && B;
endmodule
```

Example: RTL Code for Fully Power Aware Design:

© 2021 All Rights Reserved.

```
module AND(Y,A,B,VDD,VSS);
input A, B,VDD,VSS;
output Y;
reg Y;
always @(VDD or A or B)
if (VDD == 1)
Y <= A && B;
else
Y <= ‘bx;
endmodule
```

Example: RTL Code for Partial Power Aware Design:

```
module AND(Y,A,B,VDD, VSS);
input A, B,VDD,VSS;
output Y;
reg Y;
always @(A or B)
Y <= A && B;
endmodule
```

Identifying Partially Powered Aware Models and Handling Corruptions in Liberty Flow

If both UPF files and liberty files are provided, Palladium models the UPF simulation state such as power loss corruption based on the UPF files and the liberty files unless a power-aware model or partial power-aware model is identified. A liberty flow for power-aware or partial power-aware model is identified based on the following criteria:

- A module has `pg_pin` with `pg_type primary_power` attributes or `pg_pin` with `pg_type primary_ground` in liberty files.
- The module is in HDL files. The module definition in the HDL files includes `primary_power` or `primary_ground` pins. The module is power-aware, if either `primary_power` or `primary_ground` pin has a loader in the HDL module definition. It is partial power-aware if no load exists for `primary_power` or `primary_ground` in the HDL modules.

Example: Liberty File Cell Definition

```
cell (ram) {
pg_pin(VDD) {
     pg_type: primary_power;
}
pg_pin(VSS) {
     pg_type:primary_ground;
} …
}
```


© 2021 All Rights Reserved.

Example: HDL Module Definition (power-aware model with load)

```
module ram(clk,wr, addr, din, dout, VDD,VSS);
...
always @(negedge clk)
begin
   if (VDD == 1’b0 && VSS == 1’b0) begin
     dout <= 8’b1;
   end
end
…
endmodule
```

Once an HDL module is identified as a power-aware or partial power-aware module, the simstate of the module is disabled and appropriate warning messages are displayed.

The power-aware and partial power-aware modules only disable the simstate such as corruption. They do not disable other PSO behaviors such as isolation or retention, if isolation or retention applies. If `set_related_supply_net` is used for power or ground pin in the power-aware or partial power-aware model, `set_related_supply_net` is also disabled.

Identifying Partially Powered Aware Models and Handling Corruptions in UPF Files

If UPF files are provided, Palladium models the UPF simstate such as power loss corruption based on the UPF files unless a power aware or partial power-aware model is identified.

A UPF power-aware or partial power-aware model is identified based on the following criteria:

- The UPF file has `connect_supply_net` command with `-ports` option.
- The HDL module has ports with `connect_supply_net -ports` and the load. If the ports do not have load in the HDL module definition, the instances of those ports are identified as power-aware instances.

Once an instance in HDL files is identified as power aware or partial power-aware instance, the simstate of the instance is disabled with qtInfo or warning message. If HDL power or ground pin has `set_related_supply_net`, the `set_related_supply_net` is disabled.

Example: UPF connect_supply_net

```
connect_supply_net VDD -ports {ram/VDD} -vct UPF2VHDL_SL
connect_supply_net VSS -ports {ram/VSS} -vct UPF_GNDZERO2VHDL_SL
```

Example: HDL Module Definition for Power-Aware Module (with load)

```
module ram (clk, wr, addr, din, dout, VDD, VSS)
...
```


© 2021 All Rights Reserved.

```
always @(negedge clk)
begin
if (VDD==1'b0 && VSS==1'b0) begin
dout <=8'b1;
end
end
...
```

Example: HDL Top Module Definition

```
module top (clk, reset, addr, data, write, iso,save, pso,q_ver);
……
ram ram (.clk(clk), .wr(write), .addr(addr[2:0], .din(data[63:0], .dout(q_ver));
…
```

Example: UPF create_supply_port

```
create_supply_port VDD -domain domain_pso
create_supply_port VSS -domain domain_pso
```

Supported Design Attributes for Power-Aware Models

The following attributes of the `set_design_attributes -attribute` command are now supported in VXE releases:

- `UPF_dont_touch true`: This attribute is used for simulation only. If the `UPF_dont_touch` attribute is set to `true` for a module, then the module does not have any simulator-imposed semantics, such as corruption, retention, isolation, and level shifter, in UPF and liberty files. The `UPF_dont_touch attribute` has one only one valid value — that is, `true`. This attribute can also be used with the `-model` option of the `set_design_attributes` command.

  For example,

  ```
  set_design_attributes -attribute UPF_dont_touch true -model M1
  ```

- `override_lib_corruption true | false`: This attribute determines if UPF and liberty corruption should be applied. For example, `set_design_attributes -attribute override_lib_corruption false`. The default value for this attribute is `false`.
  - `false`: For a power-aware model, no UPF and liberty corruption is applied. If, however, a model is not power aware, the power down function or related power and ground pin in the liberty file determines the corruption:
    - If `power down` function is present, the output pins of the cells will be corrupted.
    - If, however, the `power down` function is not present, pins of the cells will be corrupted based on the `related_power_pin`, or `related_ground_pin`.

© 2021 All Rights Reserved.

Note: If the `override_lib_corruption` attribute is set to `False`, the `read_power_intent` command will automatically add the simstate behavior to power aware models through the `set_simstate_behavior command`.

You can use the `lp -check set_simstate_behavior` command to view the disabled models. Precompile honors the `set_simstate_behavior disable` command for the disabled power aware models, and the corruption instrumentation logic is not inserted to those power aware models The `lp -check set_design_attributes` command shows the attributes generated by the `libTo1801` utility — that is, the attributes from the liberty files translated into 1801 commands. For details on the `libTo1801`utility, refer to the *Programs and Utilities* chapter of the *VXE Command Reference Manual*.

- `true`: For both power-aware and non-power aware models, UPF corruption is always applied.

  Note: If the `override_lib_corruption` attribute is set to `True`, the `read_power_intent` command ignores the liberty attributes and does not save the `set_port_attributes` which are generated by `libTo1801` — And the `lp -check set_port_attributes` command will not show attributes from the liberty files. The `lp -check set_design_attributes` command also shows `+override_lib_corruption` <*`value`*> attribute.

Specifying Default Value Conversion Table (VCT)

In 1801 `set_design_attributes` command, you can specify a default Value Conversion Table (VCT) that defines how net values are mapped from UPF to an HDL model, or from an HDL model to UPF.

The VCT specified in the `set_design_attributes` command is used by the 1801 `connect_supply_net` command. The `connect_supply_net` command defines the connection of the supply nets to the logic power and ground pins.

This section discusses the supported attributes to specify a default VCT for power and ground nets, and the supported VCTs in VXE. It also explains the VXE behavior to determine a default VCT (if VCT is not specified in the 1801 commands). This section comprises the following topics:

- Supported Design Attributes for VCTs
- Supported VCTs in VXE
- Identifying Power and Ground Supply Nets
- Viewing Default VCTs at Run Time

© 2021 All Rights Reserved.

Supported Design Attributes for VCTs

VXE supports the following attributes in the `set_design_attributes -attributes` command to enable you to specify a default VCT for `connect_supply_net`:

- `default_power_vct`
- `default_ground_vct`

The syntax of the `set_design_attributes` command is as follows:

```
set_design_attributes -attribute <attribute> <value>
```

where, <*attribute*> can be either `default_power_vct` or `default_ground_vct`

The <*value*> can be any of the IEEE 1801 predefined VCTs, or four special VCTs specific to VXE. For information on the supported VCTs, refer to "Supported VCTs in set_design_attributes" on page 1677 in Support for IEEE 1801 Commands on Palladium Z1 section.

For example:

```
set_design_attributes -attribute default_power_vct SV_LOGIC2UPF_2S
set_design_attributes -attribute default_ground_vct SV_LOGIC2UPF_GNDZERO_2S
```

Supported VCTs in VXE

The IEEE1801 LRM defines 12 `UPF2HDL` VCTs. VXE supports all the predefined VCTs in the 1801 commands. For details on the usage, refer to the *IEEE 1801-2015 LRM*.

Additionally, VXE provides the following `SV2UPF` VCTs for the `set_design_attributes` and the `connect_supply_net` command:

- `SV_LOGIC2UPF_MD`
- `SV_LOGIC2UPF_GNDZERO_MD`
- `SV_LOGIC2UPF_2S`
- `SV_LOGIC2UPF_GNDZERO_2S`

For details on the above mentioned `SV2UPF` VCTs, refer to "Supported VCTs in set_design_attributes" on page 1677 in Support for IEEE 1801 Commands on Palladium Z1 section. Also see

Note: If, however, VCT is not specified in the `connect_supply_net` command with the `-vct` option, VXE determines the default ground and power mapping depending on whether

© 2021 All Rights Reserved.

or not a VCT is specified in the `set_design_attributes` command. The following table shows how VXE behaves when VCT is omitted in the `connect_supply_net` command:

Table 34-3 VXE Behavior for Default VCTs

Definition in set_design_attributes	Behavior
`set_design_attributes` specifies either `default_power_vct` or `default_ground_sv2upf_vct` attribute	The `read_power_intent` command automatically adds the `-vct` option in `connect_supply_net`. Precompile honors the specified VCT in the instrumentation logic. **Note:** The default VCT is supported in the ICE mode, and in SA mode when the design is swapped to the emulator.
`set_design_attributes` does not specify `default_power_vct` or `default_ground_vct`	If the design is compiled with the `preCompileOption VSSAlwaysOn` command, the values are mapped as follows: ■ power net: ON is 1; OFF is 0 ■ ground net: ON is 1; OFF is 0
	If the design is compiled with the `preCompileOption VSSswitchable`, the values are mapped as follows: ■ power net: ON is 1; OFF is 0 ■ ground net: ON is 0, OFF is 1

Identifying Power and Ground Supply Nets

Power and ground supply nets are identified in the `read_power_intent` based on the following 1801 commands:

```
create_supply_set <net> -function {power <net>} -function {ground <net>}
set_domain_supply_net <domain> -primary_power_net <ent> -primary_ground_net <net>
set_isolation <iso> -isolation_power_net <net> -isolation_ground_net <net>
set_retention <ret> -retention_power_net <net> -retention_ground_net <net>
```

If `-vct` is omitted in the `connect_supply_net command` in the original 1801 files, the xeCompile command `read_power_intent` adds the `-vct` option in the

© 2021 All Rights Reserved.

connect_supply_net command. By default, the read_power_intent command adds only the hdl2upf type VCT in the connect_supply_net command.

Precompile uses the default hdl2upf VCT added to connect_supply_net to honor the HDL power and ground pins driving the UPF supply nets, and vice-versa.

Example:

The 1801 file has the following commands:

```
set_design_attributes -attribute default_power_vct SV_LOGIC2UPF_2S
create_supply_set VDSet -function {power VDD} -function {ground VSS}
connect_supply_net VDD -ports {instance1/VDD }
```

The read_power_intent command will insert -vct in connect_supply_net as follows:

```
connect_supply_net VDD -ports {instance/VDD} -vct SV_LOGIC2UPF_2S
```

The following message will be displayed:

```
Set default power|ground VCT '<vctName>' for '<supplyNet>' connect_supply_net
(<filename>: line <lineNum>)
```

Note: -pg_type in set_port_attributes and -pg_type in connect_supply_net are not be used by read_power_intent to insert default -vct in connect_supply_net. However, precompile might use the default VCT when connecting the supply net to HDL pg ports.

Viewing Default VCTs at Run Time

At run time, you can use the lp -check command to view the default VCT applied to the set_design_attributes and connect_supply_net command.

Examples:

- The following command returns the default VCT applied to the set_design_attributes command:

  ```
  lp -check set_design_attributes
      +default_power_vct <vct>
      +default_ground_vct <vct>
  ```

- The following command returns the default VCT applied to the connect_supply_net command:

  ```
  lp -check connect_supply_net
      -vct <vct>
  ```


© 2021 All Rights Reserved.

Viewing Low Power Information in Palladium Database

At run time, you can view the low power information, that is, the UPF instrumentation strategies for the design signals, supply signals, and logic signals in the Palladium design database.

- You can view the isolation and retention strategy, power domain, related supply net, and port attribute associated with the given design objects. Execute the following command at run time:

  ```
  lp -netinfo [-state] <design_object1> <design_object2>... <design_objectN>
  ```

- You can view the connection type (implicit, explicit, or automatic), direction, scope, UPF file name, connection side (`HighConn` or `LowConn`), and so on for a given supply net or port. Execute the following command at run time:

  ```
  lp -netinfo [-state] -conn <supplynet/supplyport>
  ```

- IEEE 1801 enables creation and connection of the logic signals from within the UPF file through these 1801 commands: `create_logic_net`, `create_logic_port`, `connect_logic_net`, and `connect_logic_port` commands. To view the connection of these logic signals in either direction, execute the following command:

  ```
  lp -netinfo [-state] -conn <logic signal>
  ```

For details on the above mentioned commands and options, refer to the description of the `lp` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

You can also view power information based on a given power domain by executing `power` command at run time. The `power` command also enables you to view the PST information:

```
power [-show] [[-object <hdl_object> [<hdl_object ...>] |
               [-pdname <domain_name> [<power_domain_name]]
               [-instances | -isolation_ports | -sr_variables | -state | -verbose]
power [-show] -pst <pst> [<pst> …] [-active | -state | -verbose]
```

For details on the `power` command, options, and examples, refer to the description of the `power` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

IEEE 1801 Support in Xcelium and Palladium Z1

This section compares IEEE 1801 support provided by Xcelium and Palladium Z1.

- Similarities between Xcelium and Palladium Z1 IEEE 1801 Low-Power Implementation
- Differences between Xcelium and Palladium Z1 IEEE 1801 Low-Power Implementation

© 2021 All Rights Reserved.

Similarities between Xcelium and Palladium Z1 IEEE 1801 Low-Power Implementation

This section lists the similarities between Xcelium and Palladium Z1 IEEE 1801 low-power implementation.

Both Xcelium and Palladium Z1 support similar IEEE 1801 commands, the same UPF package functions, and take the same power attributes from the liberty files.

- Support for IEEE 1801 Commands and Options

  Both Xcelium and Palladium Z1 build up power supply network and support PSO behaviors, which are specified in IEEE 1801 commands during simulation or emulation run. Power supply network includes root supply ports, supply ports, supply nets and power switches. PSO behaviors include shut-off corruption, retention, and isolation.

  For supported IEEE 1801 commands and options, refer to Appendix 35, “Support for IEEE 1801 Commands on Palladium Z1”.

- Support for UPF Package

  Both Xcelium and Palladium Z1 support the UPF package functions - `supply_on`, `supply_off`, and `supply_partial_on`. The functions are used to switch the root supply power state, therefore, the whole supply network is switched via power switch propagation.

- Support for Liberty Files

  Both Xcelium and Palladium Z1 read liberty files and extract the low power attributes. The information extracted from the liberty files includes cell names and library cell pin attributes. The library cell pin attributes include the `pg_pin` and `pg_type` information, and the `related_power_pin`, `related_ground_pin`, and `power_down_function` of a data pin.

- Support to Disable IEEE 1801 during Run Time

  Both Xcelium and Palladium Z1 can disable IEEE 1801 support during a run-time session.

Differences between Xcelium and Palladium Z1 IEEE 1801 Low-Power Implementation

This section lists the differences between Xcelium and Palladium Z1 IEEE 1801 low-power implementation.

© 2021 All Rights Reserved.

Xcelium and Palladium Z1 have different corruption modeling, isolation details, retention details, compilation commands, debug commands, and debug GUI.

- Corruption Modeling

  Xcelium corrupts via `X` value, whereas, Palladium Z1 models corruption via power down state, which can be `low`,`high`, `inverted`, or `random` value.

  Both Xcelium and Palladium Z1 corrupts off the input ports and sequential elements (latches, FFs, and memory instances). Palladium corrupts sequential elements' value based on the `-power_down_states` and `-power_up_states` options of `create_power_domain`, but for input ports, Palladium always corrupts input ports' receivers' value to `0`.

  Xcelium uses `X` value for corruption. Palladium uses the value in `create_power_domain -power_down_states` as sequential elements' corrupt value.

  IEEE 1801 has four simstates: `CORRUPT`, `CORRUPT_ON_ACTIVITY`, `CORRUPT_STATE_ON_ACTIVITY`, and `CORRUPT_STATE_ON_CHANGE`. Palladium treats all as `CORRUPT`, whereas, Xcelium supports `CORRUPT` and `CORRUPT_ON_ACTIVITY`.

- Power Back Re-Initialization

  Xcelium uses the `set_sim_control` command to specify a list of Verilog initial blocks and disable the default simulation corruption semantics for special integers, reals, or Verilog reg type variables.

  Palladium Z1 does not support this command as it does not support Verilog initial blocks for power back re-initialization.

  Palladium Z1 uses the `low`, `high`, `inverted`, or `random` values specified in the `create_power_domain -power_up_states` command to do power back re-initialization, that is, for un-retained sequential elements in a switchable power domain, when power is back, its value is assigned to specified value. The default value is `random`.

- Isolation Details

  Xcelium does not support the `-location` option of the `set_isolation` command, whereas, Palladium Z1 supports it.

  If the isolated pin is an input pin, Xcelium inserts isolation logic in `LowConn` of the input pin. If the isolated pin is an output pin, Xcelium inserts isolation logic in `HighConn` of the output pin. Palladium Z1 supports the `-location {self | other | parent}` option of the `set_isolation` command. This will cause different values of the isolated pins

© 2021 All Rights Reserved.

between Xcelium and Palladium Z1. However, the load of the isolated pins will have the same values with Xcelium and Palladium Z1.

- Retention Details

  Palladium Z1 applies the retention to real sequential elements, which includes latch, FF and memory instances. If a `reg` in RTL is synthesized into combinational logic, Palladium Z1 will not apply retention to it. Palladium freezes and unfreezes the FFs when save or restore happens.

- Compilation Commands

  Palladium Z1 uses the `read_power_intent` command in xeCompile to read the 1801 files, whereas, Xcelium uses the `-lps_1801` option of the `xrun` command to specify the 1801 files.

  In SA mode, the `xrun` command with the `-lps_1801` option is supported.

- Debug Commands

  Palladium Z1 has `lp -check ...`, `lp -value ...`, `probe ...`, `database -upload`, and `sdl ...` commands to support IEEE 1801 debug.

  Xcelium has `describe {-power|-psn}`, `power ...`, `stop {-pdname|-iso_rule|-sr_rule} ...`, `value -lps ...`, and `driver ...` commands to support IEEE 1801 debug.

- Alert for Power Event

  Xcelium generates assertions to verify the power control signals and power event sequence. Xcelium also supports automatically generated assertions to guarantee the correct power event sequence.

  Palladium Z1 uses SDL mechanism to detect and handle IEEE 1801 power events, which includes power event sequence and 1801 object states.

- Debug GUI

  Xcelium uses SimVision, which helps to debug power supply network, and has IEEE 1801 source files browser and power sidebar in waveform windows.

  Palladium Z1 supports both SimVision (SST2) and Novas waveforms (FSDB). Palladium has its own xeDebug GUI for IEEE 1801.

© 2021 All Rights Reserved.

Table 34-4 PSO Modelling Differences Between Xcelium and Palladium

Feature	Xcelium	Palladium IXCOM Software Session	Palladium IXCOM Hardware Session	Palladium legacy ICE Mode
Corruption Modelling	Corrupts through `X` Value	Corrupts through `X` Value	Models corruption through `low, high, inverted, random`	Models corruption through `low, high, inverted, random`
Power back re-initialization	Supports initial block replay	Supports initial block replay	`low, high, inverted, random`	`low, high, inverted, random`
Isolation	`lowConn` of input `highConn` of output	`lowConn` of input `highConn` of output	`-location {self | parent | other}`	`-location {self | parent | other}`
Retention	Applies retention to `reg`	Applies retention to `reg`	Applies retention to latches, FFs, and memory instances	Applies retention to latches, FFs, and memory instances

Table 34-5 1801 Xcelium and Palladium Command Differences

Feature	Xcelium	Palladium SA Mode	Palladium ICE Mode
1801 reader	`xrun -lps_1801`	`xrun -lps_1801`	`read_power_intent`
Liberty files	`xrun -lps_lib_mfile <lib.f>`	`xrun -lps_lib_mfile <lib.f>`	`libTo1801 <libDirOrfiles> -output <upfFile>`
Checking 1801 content or values	`describe value -lps`	`describe value -lps`	`lp -check | -value`

© 2021 All Rights Reserved.

Table 34-5 1801 Xcelium and Palladium Command Differences

Values	`driver`	`driver` (in SW session) ■ `lp -value`	`lp -value`
LP Coverage	`xrun -lp_verify`	`lp -metrics`	`lp -metrics`
Probing	■ `probe -power` ■ `probe -pwr_mode`	■ `probe -power` ■ `probe -pwr_mode` ■ `probe -add<1801Obj>`	■ `probe -power` ■ `probe -pwr_mode` ■ `probe -add<1801Obj>`
Graphical User Interface	SimVision Low Power GUI	SimVision Low Power GUI and Palladium LP GUI	SimVision waveform and Palladium LP GUI

Importing IEEE Standard UPF Packages

In SA mode, invoke the IXCOM compiler or `vlan` tools with the `-upf` option to import the standard IEEE UPF package. Without the `-upf` option, the compiler expects you to provide a user-defined UPF package.

In both ICE and SA modes, the HDL file must import the package with `"import UPF::*"` in SystemVerilog and `"use ieee.upf.all"` in VHDL.

In SA mode, the IXCOM compiler supports power intent completely through the 1801 file. Use the `-lps_1801 <1801Filename>` option to provide the 1801 file to the compiler. The 1801 file can be used to declare supply nets. The IXCOM compiler passes the 1801 file to Xcelium and HDL-ICE Compiler.

Changing the Power States of Supply Nets and Ports

Each supply port and net maintains two pieces of information: a supply state and a voltage value, which together constitute the power state of the supply net or port. In ICE and SA modes, VXE supports the following standard IEEE UPF package functions for both VHDL and SystemVerilog:

© 2021 All Rights Reserved.

- `supply_on`: Changes the power state of the specified supply port or supply net to `ON` and updates the voltage of the supply.
- `supply_off`: Changes the power state of the specified supply port or supply net to `OFF` and updates the voltage of the supply.
- `supply_partial_on`: Changes the power state of the specified supply port or supply net to `PARTIAL_ON` and updates the voltage of the supply.

An error message is generated if any other UPF package function is used in the HDL files.

Note: In SA mode, declaration of UPF package supply nets in the HDL files is not supported. All supply nets should be declared in the 1801 file.

SystemVerilog UPF Package

In ICE and SA modes, supported UPF package functions for SystemVerilog are as follows:

```
function bit supply_on(string pad_name, real value);
function bit supply_off(string pad_name);
function bit supply_partial_on(string pad_name, real value);
```

For SystemVerilog, these UPF package functions are supported as functions only.

VHDL UPF Package

In ICE mode, supported UPF package functions and procedures for VHDL are as follows:

```
function supply_on(supply_name:IN string; voltage:IN real) return boolean;
function supply_off(supply_name:IN string) return boolean;
function supply_partial_on(supply_name:IN string; voltage:real) return boolean;
procedure supply_on(supply_name:IN string; voltage:IN real);
procedure supply_off(supply_name:IN string);
procedure supply_partial_on(supply_name:IN string; voltage:real);
```

In SA mode, the UPF package functions for VHDL are currently supported as procedures only, not functions. This is contrary to the IEEE 1801 LRM.

```
procedure supply_on(supply_name:IN string; voltage:IN real);
procedure supply_off(supply_name:IN string);
procedure supply_partial_on(supply_name:IN string; voltage:real);
```

Supported System Tasks and Functions

In ICE mode, the supported HDL UPF system functions and system tasks are as follows:

```
$supply_on(string supply_name, real voltage = 1.0);
$supply_off(string supply_name);
$supply_partial_on(string supply_name, real voltage = 1.0);
bit $supply_on(string supply_name, real voltage = 1.0);
```


© 2021 All Rights Reserved.

```
bit $supply_off(string supply_name);
bit $supply_partial_on(string supply_name, real voltage = 1.0);
```

In ICE mode, both `supply_*` and `$supply_*` are supported when IEEE UPF package is imported in the HDL files.

When UPF package is not imported, only `$supply_*` is supported. By default, `$supply_*` is supported without any special HDL-ICE Compiler options. `$supply_*` does not need the `-atb` option with the `hdlImport` or `vavlog` command.

`$supply_on` or `$supply_off` appear in the initial blocks, because the power needs to be managed during initialization. For initial blocks to be recognized, use the `-atb` option, otherwise, the initial blocks are ignored with warnings.

Note: In SA mode, only the HDL UPF system functions are supported, and not system tasks.

Examples

Following is an example of system tasks, where the `-atb` option is not required:

```
always @(posedge clk)
  if (powerOn)
    $supply_on("dut.VDD", 1.5);
  else if (powerOff)
    $supply_off("dut.VDD");
```

The `-atb` option is required for the initial blocks, and not for `$supply_*`. Following is an example of SystemVerilog UPF package, where the `-atb` option is required because of an initial block:

```
import UPF::*;
wire clk;
reg tmp;
initial begin
  @(posedge clk);
  tmp = supply_on("dut.VDD", 1.5);
  tmp = supply_on("dut.VDD2", 5.0);
  repeat (100) @(posedge clk);
  tmp = supply_off("dut.VDD");
  @(posedge clk);
  @(posedge clk);
  tmp = supply_off("dut.VDD2");
  repeat (50) @(posedge clk);
  tmp = supply_on("dut.VDD", 1.5);
end
```

Following is an example of VHDL UPF package, where the `-atb` option is not required:

```
library ieee;
use ieee.upf.all;
architecture power of test is
begin
```


© 2021 All Rights Reserved.

```
  process (enable) begin
    if enable = '1' then
      supply_on("dut.vdd", 1.4);
    end if;
  end process;
  process (disable) begin
    if disbale = '1' then
      supply_off("dut.vdd");
    end if;
  end process;
end;
```

Generating a Coverage Model for 1801 Low-Power Designs

The system enables you to generate and view the coverage data for low power aware 1801 designs. The coverage data can be generated in the following formats:

- Text file format
- Accellera Unified Coverage Interoperability Standard (UCIS) or UNICOV format

In the UNICOV output format, you can merge the coverage data with the simulation data, and then view the data with the Integrated Metrics Center (IMC).

The designs with IEEE1801 files are compiled by default with the hardware coverage counters for low power conditions. These counters are used to collect the coverage data from the emulation runs.

The width of the coverage counters can be configured at compile time with a global setting of 0 to 8 bits, with the default at 8. For an `N` bit wide counter, the maximum value is `2^N-1`; the counter stops upon reaching the maximum value.

The coverage counters are generated for the following IEEE1801 low power objects:

- `supplyNet`
- `supplySet`
- `powerDomain`
- `supplyPortState`
- `pstState`
- `powerState`

© 2021 All Rights Reserved.

- `stateTransition`
- `switchState`
- `isolation`
- `retention`

The coverage counters count the number of state transitions for various low power objects. For example, for power domains, the coverage counters count the number of times the power domain is in the NORMAL state and in the CORRUPT state.

For isolation, the coverage counters count the number of times when the isolation rule is active. In addition, each isolation control signal's activity is individually counted.

For retention, the coverage counters count the number of times when the retention rule is active. In addition, the retention condition, the save while the save condition true, and the restore while the restore condition true, are individually recorded.

Generating the Coverage Report

The coverage report for the low power aware designs is supported in both SA and ICE modes. In the SA mode, the coverage report is available only during the swapped-in to hardware portion of the runtime session. You can also merge the coverage data from multiple runs by adding data from multiple text files.

To generate the coverage report, issue the `lp -metrics` command.

While swapping the design to the emulator, the coverage counters are reset after swap-in and are lost on swap-out.

At compile time, you might decrease the number of bits allocated for the coverage counters from the default of 8. Decreasing the bits saves the hardware capacity. To disable the coverage counters, set the number of bits to 0. The compile time command to set the number of coverage counter bits is `compilerOption -add {1801CoverageWidth [0-8]}`.

Sample Text File With Coverage Information

```
SupplyNet,tb.dut.VDD,FULL_ON,1
SupplyNet,tb.dut.VDD,PARTIAL_ON,0
SupplyNet,tb.dut.VDD,UNDETERMINED,0
SupplyNet,tb.dut.VDD,OFF,0
SupplySet,tb.dut.VDD_VSS,NORMAL,1
SupplySet,tb.dut.VDD_VSS,CORRUPT,0
SupplySet,tb.dut.VDD_VSS_sw0,NORMAL,3
SupplySet,tb.dut.VDD_VSS_sw0,CORRUPT,2
SupplySet,tb.dut.VDD_VSS_sw1,NORMAL,3
PowerDomain,tb.dut.PD_Def,NORMAL,1
```


© 2021 All Rights Reserved.

```
PowerDomain,tb.dut.PD_Def,CORRUPT,0
PowerDomain,tb.dut.PD_bank0,NORMAL,3
PowerDomain,tb.dut.PD_bank0,CORRUPT,2
PowerDomain,tb.dut.PD_bank1,NORMAL,3
PowerDomain,tb.dut.PD_bank1,CORRUPT,2
SupplyPortState,tb.dut.VDD_sw1.x1on,True,3
SupplyPortState,tb.dut.VDD_sw1.x1on,False,2
SupplyPortState,tb.dut.VDD_sw1.x1off,True,2
STState,tb.dut.TABLE_1.off,True,2
PSTState,tb.dut.TABLE_1.off,False,3
PSTState,tb.dut.TABLE_1.on,True,3
SwitchState,tb.dut.sw_0.sw_ON,True,3
SwitchState,tb.dut.sw_0.sw_ON,False,2
SwitchState,tb.dut.sw_1.sw_ON,True,3
SwitchState,tb.dut.sw_1.sw_ON,False,2
Isolation,tb.dut.iso0,Active,0
Isolation,tb.dut.iso0,NotActive,1
Isolation,tb.dut.iso0,IsolationSignal,iso,PosEdge,0
Isolation,tb.dut.iso0,IsolationSignal,iso,NegEdge,1
Isolation,tb.dut.iso0,SupplySet,tb.dut.VDD_VSS,NORMAL,1
Isolation,tb.dut.iso0,SupplySet,tb.dut.VDD_VSS,CORRUPT,0
Retention,tb.dut.ret_0,Active,2
Retention,tb.dut.ret_0,NotActive,3
Retention,tb.dut.ret_0,SaveWithCondTrue,2
Retention,tb.dut.ret_0,RestoreWithCondTrue,2
Retention,tb.dut.ret_0,SupplySet,tb.dut.VDD_VSS,NORMAL,1
Retention,tb.dut.ret_0,SupplySet,tb.dut.VDD_VSS,CORRUPT,0
Retention,tb.dut.ret_9,Active,2
Retention,tb.dut.ret_9,NotActive,3
Retention,tb.dut.ret_9,SaveWithCondTrue,2
Retention,tb.dut.ret_9,SaveWithCondFalse,2
Retention,tb.dut.ret_9,RestoreWithCondTrue,2
Retention,tb.dut.ret_9,RestoreWithCondFalse,2
Retention,tb.dut.ret_9,RetentionCondTrue,2
Retention,tb.dut.ret_9,RetentionCondFalse,3
Retention,tb.dut.ret_9,SupplyNet,tb.dut.VDD,FULL_ON,1
Retention,tb.dut.ret_9,SupplyNet,tb.dut.VDD,PARTIAL_ON,0
Retention,tb.dut.ret_9,SupplyNet,tb.dut.VDD,UNDETERMINED,0
Retention,tb.dut.ret_9,SupplyNet,tb.dut.VDD,OFF,0
```

Viewing the Coverage Report Using IMC

The system enables you to generate the low power objects coverage information in the Unified Coverage Interoperability Standard (UCIS) database. Issue the `lp -metrics -ucis` command to generate the low power objects coverage information in the UCIS database. You can use the Integrated Metrics Center (IMC) to view the UCIS database.

The coverage counters will be probed into the power_intent_1801_objects hierarchy, under the respective UPF objects.

To open the UCIS database in IMC:

1. Launch IMC by issuing the `imc` command.

© 2021 All Rights Reserved.

2. Select `Load Data` from the toolbar and choose the database directory. All objects appear under `Instances > TOP_INST`.

Figure 34-18 IMC Verification Hierarchy

3. Right-click the required object and select the `CoverGroup Analysis` option to view the bins and counts.

© 2021 All Rights Reserved.

Figure 34-19 CoverGroups and Bins

Various other options are provided with the `lp -metrics` command to reset, enable, or disable the coverage counters. For complete information, refer to the description of the `lp -metrics` command in the *Run-Time Commands* of the *VXE Command Reference Manual*. For information about the IMC interface and the covergroup coverage, refer to the *Integrated Metrics Center User Guide*.

At run time, you can also organize the *Type* content in the *IMC Verification Hierarchy* window using the following options of the `lp -metrics` command:

`-ucis -type_pick_first`, `-ucis -type_merge_instances`, and `-ucis -type_sum_instances`.

© 2021 All Rights Reserved.

Tables 34-6 shows three instances — I1, I2, and I3 that belong to the same type “T” with corresponding grades and bins.

Table 34-6 Covergroup Instances

Instance	Type	Grade	Bins
I1	T	50%	1/2
I2	T	0%	0/2
I3	T	0%	0/4

Table 34-7 shows the resultant bins for type “T” on using different type options.

Table 34-7 Using Type Options

Options	Description	Type	Grade	Bins
`-ucis -type_pick_first`	Bins are equivalent to the number of bins in the first created instance.	T	50%	1/2
`-ucis -type_merge_instances`	Bins are calculated as union of bins from all instances.	T	25%	1/4
`-ucis -type_sum_instances`	Bins are calculated as sum of bins from all instances.	T	12.5%	1/8

For details about these options, refer to the description of the `lp -metrics` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

You can also exclude certain bins from the metrics report using the `-exclude` option. Specify the bins to be excluded in a text file and execute the following command:

```
lp -metrics -exclude <binsFile.txt>
```

The bins specified in the text file will not be added to the metrics report. For details about the text file format and examples, refer to the description of the `lp -metrics` command in the *Run-Time Commands* of the *VXE Command Reference Manual*.

Unsupported Low-Power Coverage Options

The options to control the directory structure of the generated low-power coverage database are currently not supported with the run-time `lp -metrics` command:

© 2021 All Rights Reserved.

- `-covtest:` Specifies the name of the low-power coverage database
- `-covscope:` Specifies the scope name for the low-power coverage database
- `-covworkdir:` Specifies the base name for the low-power coverage database directory

Unsupported Xcelium Options

The xmelab `-lps_verify` and `-lps_vplan` options are currently not supported with Palladium.

If these options are used in the SA mode, they affect only the software portion of the run-time session. For example, when the design is running only on the simulator and not in the hardware, the data obtained when the design is run on the emulator will not be accurate.

Coverage Metrics for Supported IEEE 1801 Objects

The following table lists the coverage metrics for the supported IEEE1801 objects:

Table 34-8 Supported IEEE 1801 Objects and Coverage Metrics

IEEE Object	Metric
`supplyNet`	Generates coverage for the following states: OFF, UNDETERMINED, PARTIAL_ON, FULL_ON
`supplySet`	Generates coverage for the following states: NORMAL or CORRUPT
`powerDomain`	Generates coverage for the following states: NORMAL or CORRUPT
`supplyPortState`	Generates coverage when the `supplyPortState` is either True or False.
`pstState` Power State Table (PST)	Generates coverage when the `pstState` is either True or False. Coverage is also generated for the supply nets used in the PST for the following states: OFF, PARTIAL_ON, and FULL_ON
`powerState`	Generates coverage when the `powerState` is either True or False.

© 2021 All Rights Reserved.

Table 34-8 Supported IEEE 1801 Objects and Coverage Metrics

`stateTransition`	Generates coverage when `stateTransition` happens.
`switchState`	Generates coverage when the `switchState` is either True or False.
`isolation`	Generates coverage for the following: ■ Each isolation signal is active or inactive. ■ Isolation power and ground nets: OFF, UNDETERMINED, PARTIAL_ON, FULL_ON ■ Isolation supply set - NORMAL or CORRUPT
`retention`	Generates coverage for the following: ■ Retention active and inactive states. ■ Retention condition is True or False. ■ Save condition true/false with save signal active ■ Restore condition true/false with restore signal active ■ Retention power and ground nets - OFF, UNDETERMINED, PARTIAL_ON, FULL_ON ■ Retention supply set - NORMAL, CORRUPT

Viewing Power Loss in Waveforms

Palladium supports power loss display in waveforms for designs with 1801 files. The display of power loss facilitates debugging functional logic. For example, when a signal value goes down, you can determine that the value change is due to power loss.

There might be several power controls associated to signals. For example, power domains, related supply and ground nets, and power attributes. When one of these power control gets powered down, the signal value gets shut down, and changes to ‘X’, which is represented by red rectangles in the waveforms.

The commands that influence the power are as follows:

- `create_power_domain`

© 2021 All Rights Reserved.

The power domains are specified using the `create_power_domain` command. When signal A is driven by a primitive gate which is under power domain `PD1`, then `PD1` is the power domain for A. Whenever the `PD1`'s primary supply state gets corrupt, the power of signal A gets shut down. Usually in the waveforms, a delay is seen between the power domain primary and power domain simstate, and the signal power down corresponds to primary.

- `set_related_supply_net`

  A signal's control power can be specified by using the `set_related_supply_net` command. If a signal has a power domain and related supply net(s), the signal gets powered down when the power domain gets corrupt or when the related supply net gets `OFF`.

- `set_port_attributes`

  The `set_port_attributes` command inserts additional instrumentation logic between a port's `LowConn` and `HighConn` side, thus making the power driver of the two sides different. For input ports, the `LowConn` side gets affected by the port attributes. for output ports, the `HighConn` side gets affected. The following options impact a port's power control:

  - `-driver_supply` and `-receiver_supply` <*`supply set name`*>

    If both `-driver_supply` and `-receiver_supply` are used in the command line, corruption in the supply set triggers the power loss.

  - `-related_power_port` and `-related_ground_port` <*`supply port name`*>

    If both `-related_power_port` and `-related_ground_port` are used in the command line, corruption in the related power and ground nets triggers the power loss.

    If port attributes, related supply and isolation is applied on a specific port, the isolation cell is inserted at the end for `LowConn` and `HighConn` side. This behavior might impact the original power domain, port attributes and the related supply.

To view the power loss in waveforms, execute the `lp -powerLossXDisplay <true | false>` command at run time. The signal value displays as `X` during power loss and represented by rectangles in the waveform, as shown in the following figure:

© 2021 All Rights Reserved.

Figure 34-20 Power Loss Display in Waveforms

By default, power loss display is disabled. The `lp -powerLossXDisplay <true|false>` command is supported in both SA and ICE mode.

Note: The `lp -powerLossDisplay` command must be executed prior to opening the waveform database to take effect.

Follow the given sequence of commands to view the power loss in waveforms:

1. Execute the `lp -powerLossXDisplay` command at run time:

   ```
   lp -powerLossXDisplay true
   ```

2. Execute the `probe` command:

   ```
   probe -create <net_name1> <net_name2> ...<net_nameN>
   ```

3. Execute the `database` command to create a new waveform database:

   ```
   database -upload
   ```

4. Open the waveform viewer and observe the required probes.

Example:

Signal `A` connects to an output port. `A`'s `HighConn` side port is `B`. The UPF instrumentation for `A` are defined as follows:

```
Create_power_domain PD_Default -include_scope -supply {primary VDDIM_VSS}
set_related_supply_net -power {VDDI_Z} -object_list {A}
set_port_attributes -ports {A} -related_power_port {VDDI_N}
```


© 2021 All Rights Reserved.

In this example, `A`'s power domain is `PD_Default`. `A`'s related supply net is `VDDI_Z`. The power port `VDDI_N` will be applied on `HighConn` port `B`.

The red rectangles in the waveform correspond to an "X", that is, power loss.

Figure 34-21 Rectangles Depicting Power Loss for A

Xcelium and Palladium Differences in Power Loss Display

In Xcelium, the X values propagate. For example, a signal shows value due to power loss, then the other signals it drives also show X values, even though the signals are not powered down.

However, in Palladium run-time waveform database, the X values do not propagate, and only the powered down signals show X values represented by red rectangles.

Tracing HDL and Supply Nets for Designs with UPF Objects

You can trace the HDL and supply nets with a specified depth in both SA and ICE mode and view the tracing results on the console. In Palladium UPF mode, UPF instrumentation logic is inserted during compilation and not in the original RTL or netlist. Tracing UPF supply net or port, tracing a design signal's drivers and showing related low-power attributes facilitates debugging low-power design with the UPF instrumentation logic.

© 2021 All Rights Reserved.

Use the `drivers -depth <n> <net>` command to initiate the tracing at run time. The `drivers` command is useful for both logic design and low-power supply net and port tracing.

You can also generate a Verilog file comprising the tracing results, and view the results in the *Verdi Waveform Viewer*. Verilog file can be generated using the `-v <verilogFilename>` option of the `drivers` command.

- If the traced net is an HDL net, the result shows the following:
  - Net name
  - Primitive gate name
  - Current level of depth
  - I/O ports — that is, the I/O port and HDL net
  - The online session shows the current run-time values.
- If the traced net is a supply net and it connects to a switch, the result shows the following:
  - Net name
  - Switch name
  - Current level of depth
  - I/O ports of primitive gates — that is, the I/O port and supply or HDL net
  - The online session shows the current values — the supply net state and voltage
- If the traced net is a supply net and it connects to an HDL net, the result shows the following:
  - Net name
  - Current level of depth
  - Information about the connectors

The hierarchical nets or connected logic nets are all grouped in the console output or output file.

Example: Tracing UPF Supply Nets (HDL with UPF Objects)

Assuming that the source of `dut.v` and `tb.v` is as follows:

© 2021 All Rights Reserved.

Table 34-9 Source Code (Testbench and DUT)

Source Code: tb.v

```
`define misc pso_def,
pso_driver, pso_load
module tb;
reg i1, i2, o, `misc;
clk_mod c1(clk);
top dut(i1, i2, o, clk,
`misc);
initial begin
i1 = 0;
i2 = 0;
end
always @(negedge clk) begin
i1 = ~i1;
i2 = ~i2;
{`misc} = 0;
end
endmodule
module clk_mod (clk);
input clk;
endmodule
```

Source Code: dut.v

```
`define misc pso_def, pso_driver,
pso_load
module top(in, in2, out, clk, `misc);
input in, in2, clk, `misc;
output out;
reg temp1, temp2, temp3, temp4;
buf(temp1, in2);
not(temp2, temp1);
and(out, temp2, temp4);
ff2_mod DRIVER(in, temp3, clk);
ff2_mod LOAD(temp3, temp4, clk);
endmodule
module ff2_mod(in, out, clk);
input in, clk;
output reg out;
reg mid1;
always @(posedge clk) begin
mid1 <= in;
out <= mid1;
end
endmodule
```

The source code of `top.upf` is as follows:

```
upf_version 2.0
set_design_top tb/dut
#VDD
create_supply_port VDD
create_supply_net VDD
connect_supply_net VDD -ports {VDD}
#VSS
create_supply_port VSS
create_supply_net VSS
connect_supply_net VSS -ports {VSS}

create_supply_port VDD_driver
create_supply_net VDD_driver
connect_supply_net VDD_driver -ports {VDD_driver}

create_supply_port VDD_load
create_supply_net VDD_load
connect_supply_net VDD_load -ports {VDD_load}

create_power_switch sw_driver \
-output_supply_port {out VDD_driver} \
-input_supply_port {in VDD} \
-control_port {cntrl pso_driver} \
-on_state { sw_ON in {! cntrl} }

create_power_switch sw_load \
-output_supply_port {out VDD_load} \
```


© 2021 All Rights Reserved.

```
-input_supply_port {in VDD_driver} \
-control_port {cntrl pso_load} \
-on_state { sw_ON in {! cntrl} }

create_supply_set SS_def    -function {power VDD}    -function {ground VSS}
create_supply_set SS_driver -function {power VDD_driver} -function {ground
VSS}
create_supply_set SS_load   -function {power VDD_load}   -function {ground VSS}

create_power_domain PD_def    -include_scope     -supply {primary SS_def}    -
power_up_states high
create_power_domain PD_driver -elements {DRIVER} -supply {primary SS_driver} -
power_up_states high
create_power_domain PD_load   -elements {LOAD}   -supply {primary SS_load}   -
power_up_states high

set_port_attributes -ports DRIVER/out -driver_supply    SS_driver
set_port_attributes -ports LOAD/in    -receiver_supply  SS_load
```

Command:

```
XE> drivers -depth 3 dut.VDD_load
```

Description:

In this example, we trace the supply net `dut.VDD_load`, which is an output of power switch `sw_load`. The control signal is called `dut.pso_load`, and the input supply comes from the `dut.VDD_driver`. Then, the power switch `dut.pso_load` is traced. However, `dut.pso_load` is connected to `pso_load` which is defined in the testbench. `dut.VDD_driver` is traced with input `dut.VDD` and control signal `dut.pso_driver`.

Sample Output:

```
(Current runtime is: 0 fclk)
Trace net: dut.VDD_load | Primitive gate: sw_load | depth: 1.
   dut.VDD_load = OFF (output, supply).
   dut.pso_load = 0 (input, hdl).
   dut.VDD_driver = OFF (input, supply).
===================< depth = 1 >===================
   dut.pso_load = 0 (input, hdl).
   pso_load = 0 (output, Z, hdl).

Trace net: pso_load | Primitive gate: Q_BUF | depth: 2.
   pso_load = 0 (output, Z, hdl).
   n1 = 0 (input, A, hdl).

Trace net: dut.VDD_driver | Primitive gate: sw_driver | depth: 2.
   dut.VDD_driver = OFF (output, supply).
   dut.pso_driver = 0 (input, hdl).
   dut.VDD = OFF (input, supply).
===================< depth = 2 >===================
   dut.pso_driver = 0 (input, hdl).
   pso_driver = 0 (output, Z, hdl).
```


© 2021 All Rights Reserved.

```
Trace net: pso_driver | Primitive gate: Q_BUF | depth: 3.
   pso_driver = 0 (output, Z, hdl).
   n1 = 0 (input, A, hdl).
```

For more information on the syntax and elaborate examples of the `drivers` command, refer to the description of the `drivers` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Monitoring 1801 Objects with Dynamic RTL

You can use dynamic RTL to create monitor programs with standard HDL at run time. VXE also enables you to monitor the state of the IEEE 1801 objects with dynamic RTL. You can verify the state of an IEEE 1801 object using the following expressions in DRTL designs:

- *<1801 condition>* == POWER.*<UPF_state mnemonics>*

  Note: Only the power domain, supply set, isolation and retention objects are supported for the above expression.

- *<1801 condition>* == 0 or *<1801 condition>* == 1

You can also view the supported 1801 conditions for DRTL designs at run time by executing the following command:

```
lp -check drtl [<1801_condition_type>]
```

For information about the output of the `lp -check drtl` command and an example, refer to the description of the `lp` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

The following table lists the supported 1801 conditions and their values. The prefix "`POWER.`" for the 1801 conditions is mandatory. The prefix is used to differentiate the 1801 objects from the other design objects with the same name.

Table 34-10 Supported 1801 Conditions for DRTL Designs

1801 Condition Type	Supported 1801 Condition	Values (1,0, or state mnemonics)

© 2021 All Rights Reserved.

Global	POWER.anyIsolationControlSignalCorrupt	1, 0
	POWER.anyIsolationControlSignalCorrupt Trigger	
	POWER.anyRetentionControlSignalCorrupt	
	POWER.anyRetentionControlSignalCorrupt Trigger	
	POWER.anyIllegalEvent	
	POWER.runLPDisable	
supplyNets	POWER.<supplyNet>.\@OFF	1, 0
	POWER.<supplyNet>.\@FULL_ON	
	POWER.<supplyNet>.\@PARTIAL_ON	
	POWER.<supplyNet>.\@UNDETERMINED	
supplyPorts	POWER.<supplyPort>.\@OFF	1, 0
	POWER.<supplyPort>.\@FULL_ON	
	POWER.<supplyPort>.\@PARTIAL_ON	
	POWER.<supplyPort>.\@UNDETERMINED	
supplyPortStates	POWER.<port>.<supplyPortState>	1, 0
supplySet	POWER.<supplySet>	NORMAL, CORRUPT
	POWER.<supplySet>.\@<power_state_name>	1, 0
domains	POWER.<domain>	NORMAL, CORRUPT
	POWER.<domain>.\@NORMAL	1, 0
	POWER.<domain>.\@CORRUPT	
	POWER.<domain>.\@<power_state_name>	

© 2021 All Rights Reserved.

Isolations	POWER.<isolation>	NORMAL CORRUPT
	POWER.<isolation>.\@isolation POWER.<isolation>.\@corrupted POWER.<isolation>.\@inputDriverCorrupt POWER.<isolation>.\@domainCorrupt	1, 0
Retentions	POWER.<retention>	NORMAL CORRUPT
	POWER.<retention>.\@retention POWER.<retention>.\@corrupted POWER.<retention>.\@save_condition POWER.<retention>.\@restore_condition POWER.<retention>.\@retention_condition	1, 0
Switches	POWER.<switch>.\@on_state@<state_name> POWER.<switch>.\@off_state@<state_name> POWER.<switch>.\@error_state@<state_name>	1, 0
pst	POWER.<pst>.\@<pst_state_name>	1, 0
pstStates	POWER.<pstState>	1, 0
powerStates	POWER.<object>.<powerState>	1, 0
stateTransitions	POWER.<stateTranisition>	1, 0

© 2021 All Rights Reserved.

Tips for Debugging Low-Power Aware Designs

The following section lists some tips for debugging low-power aware designs:

- To check the domain's static information, use the following command:

  ```
  lp -check create_power_domain *<domainName>
  ```

- To view the power domain waveform, issue the following command:

  ```
  probe -addPowerdomain *
  ```

  You can also probe the specific domain to get the respective domain’s waveform.

- To collect the power event log without probing any waveform, issue the following command:

  ```
  lp -log -start
  ```

- If some or all power supplies are off, use the following command to get all root supply names:

  ```
  lp -check root
  ```

  To probe the root supplies in the waveform, use the following command:

  ```
  probe -addroot
  ```

- If the clamped value is wrong or some pins are not isolated as expected, issue the following command in xeDebug to get static information and the isolated pin list.

  ```
  lp -check isolation *
  ```

  To get the related waveform, use `lp -addisolation *`.

- If the restore value is wrong, the reset does not happen as expected. In such cases, use the following command in xeDebug to get retention static information and actually retained pins or `FF`s.

  ```
  lp -check retention *
  ```

  To get the related waveform, use `lp -addretention *`.

© 2021 All Rights Reserved.

35

Support for IEEE 1801 Commands on Palladium Z1

VXE does not support the entire set of functionality available in the UPF 1.0, UPF 2.0 (IEEE 1801-2009), UPF 2.1 (IEEE 1801-2013), and UPF 2.3 (IEEE 1801-2015) versions. For syntax of the supported IEEE 1801 commands, refer to *IEEE Standard for Design and Verification of Low-Power Integrated Circuits* released by IEEE.

The following table lists the supported and unsupported IEEE 1801 commands:

Table 35-1 Supported and Unsupported IEEE 1801 Commands

Supported IEEE 1801 Command(s)	Unsupported IEEE 1801 Command(s)
`add_domain_elements` [deprecated] 1	`apply_power_model`
`add_port_state` [legacy] 1	`begin_power_model`
`add_power_state`	`create_composite_domain`
`add_pst_state` [legacy] 1	`end_power_model`
`associate_supply_set`	`load_simstate_behavior`
`connect_logic_net`	`load_upf_protected`
`connect_supply_net`	`name_format`
`connect_supply_set`	`set_power_switch`
`create_hdl2upf_vct`	`define_always_on_cell`
`create_logic_net`	`define_diode_clamp`
`set_repeater`	`define_isolation_cell`
`create_logic_port`	`define_level_shifter_cell`
`create_power_domain`	`define_power_switch_cell`

© 2021 All Rights Reserved.

Supported IEEE 1801 Command(s)	Unsupported IEEE 1801 Command(s)
create_power_switch	define_retention_cell
create_pst [legacy] 1	query_upf
create_supply_net	query_associate_supply_set
create_supply_port	query_bind_checker
create_supply_set	query_cell_instance
create_upf2hdl_vct	query_cell_mapped
describe_state_transition	query_composite_domain
find_objects	query_design_attributes
load_upf	query_hdl2upf_vct
map_isolation_cell	query_isolation
set_design_attributes	query_isolation_control
set_design_top	query_name_format
set_domain_supply_net [legacy] 1	query_net_ports
set_equivalent	set_level_shifter
set_isolation	query_partial_on_translation
set_isolation_control	query_pin_related_supply
set_partial_on_translation	query_port_attributes
set_port_attributes	query_port_direction
set_retention	query_port_net
set_retention_control [deprecated] 1	query_port_state
set_scope	query_power_domain
set_simstate_behavior	query_power_domain_element
upf_version	query_power_state
set_related_supply_net 2	query_power_switch
set_retention_elements	query_retention
	query_retention_control
	query_retention_elements

© 2021 All Rights Reserved.

Supported IEEE 1801 Command(s)	Unsupported IEEE 1801 Command(s)
	`query_simstate_behavior`
	`query_state_transition`
	`query_supply_net`
	`query_supply_port`
	`query_supply_set`
	`query_user_interface_cell`

1 The [deprecated] or [legacy] commands are either deprecated or marked as legacy commands in IEEE 1801-2013.

2 This command is a Synopsys proprietary command. It is not included in the IEEE 1801 standard.

Supported IEEE 1801 Commands

The following table lists all IEEE 1801 commands and command options that are supported in VXE.

Command	Option	Valid in Version			Support[1]	Comment
`add_domain_elements`		1.0	2.0	2.1	S	
	`domain_name`	1.0	2.0	-	S	
	-elements	1.0	2.0	2.1	S	
`add_port_state`		1.0	2.0	2.1	S	
	`port_name`	1.0	2.0	2.1	S	
	-state	1.0	2.0	2.1	S	
`add_power_state`			2.0	2.1	S	
	`object_name`		2.0	2.1	S	
	-supply			2.1	S	
	-domain			2.1	S	

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment
	-state		2.0	2.1	S	VXE supports mixed usage of the `-state` option of the `add_power_state` command in the `add_pst_state` command. You can define a power state by using the `add_power_state` command, and use the defined power state in the `add_pst_state` command.
	-supply_expr		2.0	2.1	S	Expression in `-supply_expr` is defined in terms of supply ports, supply nets, and/or supply set handle functions.
	-logic_expr		2.0	2.1	S	Expression in `-logic_expr` is defined in terms of logic nets and/or power states of supply sets and/or power domains. **Note:** VXE does not support the `interval(...)` function in `-logic_expr` of `add_power_state`.
	-simstate		2.0	2.1	S	
	-legal		2.0	2.1	S	
	-illegal		2.0	2.1	S	
	-update		2.0	2.1	S	
`add_pst_state`		1.0	2.0	2.1	S	

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment
	state_name	1.0	2.0	2.1	S	
	-pst	1.0	2.0	2.1	S	
	-state	1.0	2.0	2.1	S	
associate_supply_set			2.0	2.1	S	
	supply_set_name	-	2.0	2.1	S	
	-handle	-	2.0	2.1	S	
connect_logic_net			2.0	2.1	S	
	net_name	-	2.0	2.1	S	
	-ports	-	2.0	2.1	S	
	-reconnect			2.1	NS	
connect_supply_net		1.0	2.0	2.1	S	
	net_name	1.0	2.0	2.1	S	
	-ports	1.0	2.0	2.1	S	
	-pg_type	1.0	2.0	2.1	S	
	-vct	1.0	2.0	2.1	S	
	-pins	1.0	2.0	2.1	NS	
	-cells	1.0	2.0	2.1	NS	
	-domain	1.0	2.0	2.1	S	
	-rail_connection (replaced with pg_type)	1.0	2.0	2.1	NS	
connect_supply_set			2.0	2.1	S	
	supply_set_ref	-	2.0	2.1	S	
	-connect	-	2.0	2.1	S	
	-elements	-	2.0	2.1	S	
	-exclude_elements	-	2.0	2.1	S	
	-transitive	-	2.0	2.1	S	
create_hdl2upf_vct		1.0	2.0	2.1	S	

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment
	vct_name	1.0	2.0	2.1	S	
	-hdl_type	1.0	2.0	2.1	S	
	-table	1.0	2.0	2.1	S	
`create_logic_net`			2.0	2.1	S	
	net_name	-	2.0	2.1	S	
`create_logic_port`			2.0	2.1	S	
	port_name	-	2.0	2.1	S	
	-direction	-	2.0	2.1	S	
`create_power_domain`		1.0	2.0	2.1	S	When power is off, the sequential elements are in corruption states. IEEE 1801 does not specify which values can be used as corruption value. Palladium supports the `-power_up_states` and `-power_down_states` options of the `create_power_domain` command to specify the corruption values as `low`, `high`, `random`, or `inverted` in 1801 flow. These options are not defined in IEEE 1801.
	domain_name	1.0	2.0	2.1	S	
	-simulation_only	-	2.0	2.1	S	
	-atomic	-	-	2.1	NS	
	-elements	1.0	2.0	2.1	S	
	-elements {.}			2.1	S	
	-exclude_elements	-	2.0	2.1	S	

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment
	-supply	-	2.0	2.1	S	
	-define_func_type	-	2.0	2.1	S	
	-available_supplies			2.1	S	
	-update	-	2.0	2.1	S	
	-include_scope	1.0	2.0	2.1	S	
	-scope	1.0	2.0	2.1	S	
	-power_down_states {high | low | random | inverted}	-	-	-	S	Specifies the output values for the sequential logic in the corresponding power domain after the domain is in power-off state. The default value is `low`.

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment
	-power_up_states {high | low | random | inverted}	-	-	-	S	Specifies the output values of the non-state-retention sequential logic in the corresponding power domain after the domain is in power-up state. The default value is `random`. To change power up or power down default values globally, you can use the following command: `set_design_attributes -elements {*} -attribute {CVA_power_up_states | CVA_power_down_states [low | high | random | inverted]}` If `-power_up_states` or `-power_down_states` are given in the `create_power_domain` command, the power up and power down values overwrite the global default values for the power domain. **Note:** `-elements` option is mandatory for both ICE and SA modes.
`create_power_switch`		1.0	2.0	2.1	S	
	`switch_name`	1.0	2.0	2.1	S	
	-output_supply_port	1.0	2.0	2.1	S	
	-input_supply_port	1.0	2.0	2.1	S	
	-control_port	1.0	2.0	2.1	S	

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment
	-on_state	1.0	2.0	2.1	S	
	-off_state	1.0	2.0	2.1	S	
	-supply_set	1.0	2.0	2.1	S	
	-on_partial_state	1.0	2.0	2.1	S	
	-ack_port	1.0	2.0	2.1	S	In SA mode, boolean functions are supported only if `-xmelabargs -lps_v10_ack` options are used during compile.
	-ack_delay	1.0	2.0	2.1	S	The `-ack_delay` option is supported with some limitations. The acknowledge delay is not 100% accurate, it might shift forward in time to be aligned with a valid event in DUT, that is, a posedge on the fastest DUT clock in ICE mode or a scheduled time event according to HDL source code in SA mode. It does not generate a new event along the time axis. In ICE mode, the acknowledge delay is the cycle number. In SA flow, the acknowledge delay is the simulation time.
	-error_state	1.0	2.0	2.1	S	
	-domain	1.0	2.0	2.1	S	
	-instance		2.0	2.1	NS	
	-update		2.0	2.1	S	
`create_pst`		1.0	2.0	2.1	S	

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment
	table_name	1.0	2.0	2.1	S	
	-supplies	1.0	2.0	2.1	S	
create_supply_net		1.0	2.0	2.1	S	
	net_name	1.0	2.0	2.1	S	
	-domain	1.0	2.0	2.1	S	
	-reuse	1.0	2.0	2.1	S	
	-resolve	1.0	2.0	2.1	S	
create_supply_port		1.0	2.0	2.1	S	
	port_name	1.0	2.0	2.1	S	
	-domain	1.0	2.0	2.1	S	
	-direction	1.0	2.0	2.1	S	
create_supply_set			2.0	2.1	S	
	set_name	-	2.0	2.1	S	
	-function	-	2.0	2.1	S	
	-reference_gnd	-	2.0	2.1	S	
	-update	-	2.0	2.1	S	
create_upf2hdl_vct		1.0	2.0	2.1	S	
	vct_name	1.0	2.0	2.1	S	
	-hdl_type	1.0	2.0	2.1	S	
	-table	1.0	2.0	2.1	S	
describe_state_transition			2.0	2.1	S	
	transition_name		2.0	2.1	S	
	-object		2.0	2.1	S	
	-from		2.0	2.1	S	
	-to		2.0	2.1	S	
	-paired		2.0	2.1	S	
	-legal		2.0	2.1	S	

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment
	-illegal		2.0	2.1	S	
`find_objects`			2.0	2.1	S	
	`scope`		2.0	2.1	S	
	-pattern		2.0	2.1	S	Hierarchy is allowed in the pattern string. For example, pattern as `"-pattern *"` and `"-pattern */*"` are supported. Wildcard in the pattern string does not cover delimiter. For example, pattern as `"-pattern */a/b"` is three hierarchy layers under current scope, `"-pattern *a/b"` is two hierarchy layers under current scope. If a pattern is a vector name, it is not expanded into vector bits, instead the vector name is returned. For example, for vector `g[3:0]`, the pattern `g` returns the vector name `"g"`. Pattern `"g\[1\]"` means bus bit `g[1]`. The pattern `"g\[*\]"` mean all bus bits in vector `g`.
	-object_type		2.0	2.1	S	
	-direction		2.0	2.1	S	
	-transitive		2.0	2.1	S	
	-regexp		2.0	2.1	S	
	-exact		2.0	2.1	S	

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment
	-ignore_case		2.0	2.1	S	
	-non_leaf		2.0	2.1	S	
	-leaf_only		2.0	2.1	S	A liberty file can be used to determine if a cell is a leaf cell or not. If a cell is present in a liberty file, it is treated as a leaf cell. The `libTo1801` command is used to extract information from liberty files and generate an IEEE 1801 file. When the `read_power_intent` command reads the generated IEEE 1801 file and the IEEE 1801 file that you provide, you can determine if a cell is a leaf cell or a non-leaf cell. **Note:** Without liberty files, the `-leaf_only` and `-non_leaf` options of the `find_objects` command are ignored with a warning message.
	-object_type <process>		2.0	2.1	NS	
	-object_type <supply_port>			2.3	NS	
`load_upf`		1.0	2.0	2.1	S	
	`upf_file_name`	1.0	2.0	2.1	S	
	-scope	1.0	2.0	2.1	S	
	-version	1.0	2.0	2.1	S	
`map_isolation_cell` (`use_interface_cell`)[deprecated] 1		1.0	2.0	2.1	S	
	isolation_name	1.0	2.0	2.1	S	

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment
	-domain	1.0	2.0	2.1	S	
	-elements	1.0	2.0	2.1	NS	
	-lib_cells	1.0	2.0	2.1	NS	
	-lib_cell_type	1.0	2.0	2.1	NS	
	-lib_model_name	1.0	2.0	2.1	NS	
	-port	1.0	2.0	2.1	NS	
`set_design_attributes`			2.0	2.1	S	The `set_design_attributes` command can be put in a separate 1801 file, and the `read_power_intent` command can read this 1801 file together with the other 1801 files that you provide. This command can also be added directly in an existing 1801 file. The last `set_design_attributes` setting overwrites the previous settings.
	-elements		2.0	2.1	S	
	-models		2.0	2.1	S	
	-exclude_elements		2.0	2.1	NS	

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment

	-attribute	-	2.0	2.1	S	The `-attribute` option only supports some special attributes in VXE.
						The supported attributes are:
						`-attribute {domain_interface_def UPF | 1801_2009 | 1801_2013 | 1.0 | 2.0 | 2.1}`
						`-attribute {default_applies_to UPF | 1801_2009 | 1801_2013 | 1.0 | 2.0 | 2.1}`
						`-attribute {CVA_power_up_states | CVA_power_down_states [low | high | random | inverted]}`
						`-attribute ret_cell_type ARM_RETN_CELL_1`
						`-attribute iso_nor TRUE|FALSE`
						`-attribute UPF_dont_touch TRUE`
						`-attribute override_lib_corruption {TRUE | FALSE}`

	-attribute {iso_nor TRUE}	-	-	-	S	
	-elements {strategy_name}	-	-	-	S	
	-attribute {ret_cell_type arm_rtn_cell_1}	-	-	-	S	
	-elements {retention_strategy_list}	-	-	-	S	
	-is_leaf_cell	-	-	2.1	S	
	-is_macro_cell	-	-	2.1	S	
`set_design_top`		1.0	2.0	2.1	S	

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment

	design_top_module	1.0	2.0	2.1	S	In SA mode, the VHDL *design_top_module* can be specified as follows:
						`<TB>/<DUT>`
						`:<TB>/<DUT>`
						`:<DUT>`

`set_domain_supply_net`		1.0	2.0	2.1	S	
	domain_name	1.0	2.0	2.1	S	
	-primary_power_net	1.0	2.0	2.1	S	
	-primary_ground_net	1.0	2.0	2.1	S	
`set_equivalent`						
	-function_only			2.1	S	
	-nets			2.1	S	
	-sets			2.1	S	
`set_isolation`		1.0	2.0	2.1	S	
	strategy_name	1.0	2.0	2.1	S	
	-domain	1.0	2.0	2.1	S	
	-elements	1.0	2.0	2.1	S	
	-exclude_elements	-	-	2.1	S	
	-use_equivalence			2.1	S	
	-source	-	2.0	2.1	S	
	-sink	-	2.0	2.1	S	
	-diff_supply_only	-	2.0	2.1	S	

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment
	-applies_to	1.0	2.0	2.1	S	The default value of the `-applies_to` option of the `set_isolation` command is determined as follows: ■ For UPF 1.0, the default value is `outputs`. ■ For UPF 2.0 (IEEE 1801-2009), the default value is `outputs`. ■ For UPF 2.1 (IEEE 1801-2013), the default value is none, which implies `both`.
	-applies_to_clamp	-	2.0	2.1	S	
	-applies_to_sink_off_clamp	-	2.0	2.1	S	
	-applies_to_source_off_clamp	-	2.0	2.1	S	
	-no_isolation	1.0	2.0	2.1	S	
	-force_isolation	-	2.0	2.1	S	
	-location	-	2.0	2.1	S	
	-clamp_value	1.0	2.0	2.1	S	
	-isolation_signal	-	2.0	2.1	S	
	-isolation_sense	-	2.0	2.1	S	
	-isolation_supply_set	-	2.0	2.1	S	
	-update	-	2.0	2.1	S	
	-sink_off_clamp	-	2.0	2.1	S	
	-source_off_clamp	-	2.0	2.1	S	
	-isolation_power_net	1.0	2.0	2.1	S	

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment
	-isolation_ground_net	1.0	2.0	2.1	S	
	-transitive	-	2.0	2.1	S	
	-target	-	-	-	S	`-target` is an Xcelium option. This option is not specified in the standard IEEE LRMs.
	-name_prefix		2.0	2.1	NS	
	-name_suffix		2.0	2.1	NS	
	-instance		2.0	2.1	NS	
`set_isolation_control`		1.0	2.0	2.1	S	
	`isolation_name`	1.0	2.0	2.1	S	
	-domain	1.0	2.0	2.1	S	
	-isolation_signal	1.0	2.0	2.1	S	
	-isolation_sense	1.0	2.0	2.1	S	
	-location	1.0	2.0	2.1	S	
`set_partial_on_translation`			2.0	2.1	S	
	OFF	-	2.0	2.1	S	
	FULL_ON	-	2.0	2.1	S	
	-full_on_tools		2.0	2.1	NS	
	-off_tools		2.0	2.1	NS	
`set_port_attributes`			2.0	2.1	S	
	-model	-	2.0	2.1	S	
	-elements	-	2.0	2.1	S	
	-exclude_elements	-	2.0	2.1	S	
	-ports	-	2.0	2.1	S	
	-exclude_ports	-	2.0	2.1	S	
	-applies_to	-	2.0	2.1	S	
	-attribute	-	2.0	2.1	S	

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment
	-clamp_value	-	2.0	2.1	S	
	-sink_off_clamp	-	2.0	2.1	S	
	-source_off_clamp	-	2.0	2.1	S	
	-driver_supply	-	2.0	2.1	S	
	-receiver_supply	-	2.0	2.1	S	
	-pg_type	-	2.0	2.1	S	
	-related_power_port	-	2.0	2.1	S	
	-related_ground_port	-	2.0	2.1	S	
	-domains	-	2.0	2.1	S	
	-exclude_domains	-	2.0	2.1	S	
	-transitive	-	2.0	2.1	S	
	-related_bias_port		2.0	2.1	NS	
	-repeater_supply		2.0	2.1	NS	
	-feedthrough			2.1	NS	
	-unconnected			2.1	NS	
`set_repeater`		-	-	2.1	S	
	-domain	-	-	2.1	S	
	-elements	-	-	2.1	S	
	-exclude_elements	-	-	2.1	S	
	-source	-	-	2.1	S	
	-sink	-	-	2.1	S	
	-use_equivalence	-	-	2.1	S	
	-applies_to	-	-	2.1	S	
	-repeater_supply_set	-	-	2.1	S	
	-update	-	-	2.1	S	
	-name_prefix	-	-	2.1	NS	
	-name_suffix	-	-	2.1	NS	

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment
	-instance	-	-	2.1	NS	
`set_retention`		1.0	2.0	2.1	S	
	`retention_name`	1.0	2.0	2.1	S	
	-domain	1.0	2.0	2.1	S	
	-elements	1.0	2.0	2.1	S	The option `-elements` and can have SystemVerilog multi-dimensional array signal name, or field name. For example, for signal `byte Out[3:0][2:0][1:0]`, the `retention set_retention -elements k/Out[0]` is equal to `set_retention -elements k/Out[0][2:0][1:0][0:7]`
	-exclude_elements	-	2.0	2.1	S	The option `-exclude_elements` can have SystemVerilog multi-dimensional array signal name, or field name.
	-retention_supply_set	-	2.0	2.1	S	
	-no_retention	-	2.0	2.1	S	
	-save_signal	-	2.0	2.1	S	
	-restore_signal	-	2.0	2.1	S	
	-save_condition	-	2.0	2.1	S	
	-restore_condition	-	2.0	2.1	S	
	-retention_condition	-	2.0	2.1	S	
	-use_retention_as_primary	-	2.0	2.1	S	
	-transitive	-	2.0	2.1	S	

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment
	-update	-	2.0	2.1	S	
	-retention_power_net	1.0	2.0	2.1	S	
	-retention_ground_net	1.0	2.0	2.1	S	
	-parameters		2.0	2.1	NS	
	-instance		2.0	2.1	NS	
`set_retention_control`		1.0	2.0	2.1	S	
	`retention_name`	1.0	2.0	2.1	S	
	-domain	1.0	2.0	2.1	S	
	-save_signal	1.0	2.0	2.1	S	
	-restore_signal	1.0	2.0	2.1	S	
	-assert_r_mutex	1.0	2.0	2.1	NS	
	-assert_s_mutext	1.0	2.0	2.1	NS	
	-assert_rs_mutex	1.0	2.0	2.1	NS	
`set_scope`		1.0	2.0	2.1	S	
	`instance`	1.0	2.0	2.1	S	
`set_simstate_behavior`			2.0	2.1	S	
	ENABLE | DISABLE	-	2.0	2.1	S	
	-elements	-	2.0	2.1	S	
	-exclude_elements	-	2.0	2.1	S	
	-lib		2.0	2.1	NS	
	-model		2.0	2.1	S	
`upf_version`		1.0	2.0	2.1	S	The `upf_version` command only affects syntax, it does not determine semantics of the IEEE 1801 commands.
	`string`	1.0	2.0	2.1	S	

© 2021 All Rights Reserved.

Command	Option	Valid in Version			Support[1]	Comment
`set_related_supply_net`		-	-	-	S	
	-object_list	-	-	-	S	
	-power	-	-	-	S	
	-ground	-	-	-	S	
	-reset	-	-	-	NS	
`set_retention_elements`						
	-elements	-	2.0	2.1	S	
	-applies_to	-	2.0	2.1	NS	
	-exclude_elements	-	2.0	2.1	S	
	-retention_purpose	-	2.0	2.1	NS	
	-transitive	-	2.0	2.1	NS	
	-expand (deprecated in IEEE 1801-2003)	-	2.0	2.1	NS	

1 “S” - Supported in VXE

2 “NS” - Not Supported in VXE

Supported VCTs in set_design_attributes

VXE supports the following predefined `hdl2upf` VCT tables in the `set_design_attributes` command for default power and ground supplies:

Table 35-2 Supported VCTs in set_design_attributes

VCT	Value	Support
SV_LOGIC2UPF	■ 1: FULL_ON ■ 0: OFF ■ Z: UNDETERMINED ■ X: UNDETERMINED	S

© 2021 All Rights Reserved.

Table 35-2 Supported VCTs in set_design_attributes

SV_LOGIC2UPF_GNDZERO	■ 1: OFF ■ 0: FULL_ON ■ Z: UNDETERMINED ■ X: UNDETERMINED	S
SV_LOGIC2UPF_MD	■ 1: FULL_ON ■ 0: OFF ■ Z: UNDETERMINED ■ X: OFF	This VCT is specific to VXE.It is not specified in the standard IEEE LRMs.
SV_LOGIC2UPF_GNDZERO_MD	■ 1: OFF ■ 0: FULL_ON ■ Z: OFF ■ X: UNDETERMINED	This VCT is specific to VXE.It is not specified in the standard IEEE LRMs.
SV_LOGIC2UPF_2S	■ 1: FULL_ON ■ 0: OFF ■ Z: OFF ■ X: OFF	This VCT is specific to VXE.It is not specified in the standard IEEE LRMs.
SV_LOGIC2UPF_GNDZERO_2S	■ 1: OFF ■ 0: FULL_ON ■ Z: OFF ■ X: OFF	This VCT is specific to VXE.It is not specified in the standard IEEE LRMs.

© 2021 All Rights Reserved.

Table 35-2 Supported VCTs in set_design_attributes

VHDL_SL2UPF	■ 1: FULL_ON ■ 0: OFF ■ Z: UNDETERMINED ■ X: UNDETERMINED ■ H: FULL_ON ■ L: OFF ■ W: UNDETERMINED ■ –: UNDETERMINED	S
VHDL_SL2UPF_GNDZERO	■ 1: OFF ■ 0: FULL_ON ■ Z: UNDETERMINED ■ X: UNDETERMINED ■ H: OFF ■ L: FULL_ON ■ W: UNDETERMINED ■ –: UNDETERMINED	S

1 “S” - Supported in VXE

2 “NS” - Not Supported in VXE

Supported VCTs in connect_supply_net

In the `connect_suply_net` command, VXE supports all the `hdl2upf` VCTs listed in Supported VCTs in set_design_attributes. Additionally, the `upf2hdl` VCTs listed in the following table are also supported:

Table 35-3 Supported VCTs in connect_supply_net

VCT	Value	Support

© 2021 All Rights Reserved.

Table 35-3 Supported VCTs in connect_supply_net

SV_TIED_HI	■ FULL_ON: 1 ■ OFF: X ■ PARTIAL_ON: X ■ UNDETERMINED: X	S
SV_TIED_LO	■ FULL_ON: 0 ■ OFF: X ■ PARTIAL_ON: X ■ UNDETERMINED: X	S
VHDL_TIED_HI	■ FULL_ON: 1 ■ OFF: X ■ PARTIAL_ON: X ■ UNDETERMINED: X	S
VHDL_TIED_LO	■ FULL_ON: 0 ■ OFF: X ■ PARTIAL_ON: 0 ■ UNDETERMINED: X	S
UPF2SV_LOGIC	■ FULL_ON: 1 ■ OFF: 0 ■ PARTIAL_ON: X ■ UNDETERMINED: X	S
UPF2VHDL_SL	■ FULL_ON: 1 ■ OFF: 0 ■ PARTIAL_ON: X ■ UNDETERMINED: X	S

© 2021 All Rights Reserved.

Table 35-3 Supported VCTs in connect_supply_net

UPF_GNDZERO2SV_LOGIC	■ FULL_ON: 0 ■ OFF: 1 ■ PARTIAL_ON: X ■ UNDETERMINED: X	S
UPF_GND2ZERO2VHDL_SL	■ FULL_ON: 0 ■ OFF: 1 ■ PARTIAL_ON: X ■ UNDETERMINED: X	S

1 “S” - Supported in VXE

2 “NS” - Not Supported in VXE

© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

36

Using the Dynamic Target Mode

A target system (or just a target) is an external board-level electronic system you provide that communicates with the emulated design through special connectors on the Palladium Z1 system's external panel. A static target is a target system whose clocks can be stopped and restarted without corrupting its state. A dynamic target is a target whose clocks cannot be stopped and restarted.

Dynamic Target mode is suited for running a design while using dynamic targets in the SA mode. This chapter covers the following topics:

- Overview of Dynamic Target Mode
- Assertions Support in Dynamic Target Mode
- Limitations of Dynamic Target Mode

Overview of Dynamic Target Mode

The Dynamic Target mode is based on the autorun mode available in the SA mode. When you invoke the autorun mode using the command `xc on -autorun`, the system is configured to run with static targets.

Dynamic Target mode in the SA mode corresponds to the Logic Analyzer (LA) mode in the ICE mode. For example, the rules for reading and writing memories and signal values in the Dynamic Target mode are exactly the same as in LA mode. It is possible to switch between the Dynamic Target mode and the Static Target mode during a debug session. This is useful when dealing with a target system that behaves both as dynamic target and static target. In such a case, some segments of the debug session are run in the Dynamic Target mode as the clock cannot be interrupted, while other segments can take advantage of the better debug capabilities of the Static Target mode. By switching between the Dynamic Target mode and the regular autorun mode (which supports running with a static target), features of IXCOM environment, such as hot swap using Xcelium can be used.

© 2021 All Rights Reserved.

This section provides steps for running and stopping designs in the Dynamic Target mode (both non blocking and blocking), and switching between the Dynamic Target mode and Static Target mode.

Clocking Requirements for Dynamic Target Mode

The following section lists the mandatory clocking requirements while running the designs in the Dynamic Target mode.

Using CAKE Clocks

You can use the CAKE clocks in both SA and ICE modes, with or without targets. However, in SA mode with targets, you must use CAKE clocks. You cannot use `#delay` clocks in SA mode with targets.

You must use the `ixclkgen` utility to generate the CAKE clocks.

For more information about the `ixclkgen` utility, refer to the description of `ixclkgen` command in the *Commands and System Tasks for SA* mode chapter of the *VXE Command Reference Manual*.

If a design using a clock constructed in an always block, is run in the dynamic target mode, the following error message is displayed at run time:

```
ERROR (legacy-52392): To use Dynamic Target mode, you need to define clocks with
the ixclkgen utility.
```

Using #delay Clocks

You can use the `#delay` clocks only in Transaction Based Acceleration (TBA) and in the SA mode without targets.

Running Designs in Dynamic Target Mode

By default, a design run in the Dynamic Target mode is non blocking, which means that you get the prompt to run other commands immediately. You can also run in the blocking mode, which means that the running of other commands (or the availability of command prompt in an interactive session) is suspended until the `run` command terminates.

To run a design in the Dynamic Target mode:

1. Compile the design.

© 2021 All Rights Reserved.

2. Invoke the autorun mode.

   ```
   xc on -autorun
   ```

   The above command configures the system for Static Target mode.

3. Switch to the Dynamic Target mode using the following command:

   ```
   configPM -dynamicTarget
   ```

4. Use the following command to run designs in the Dynamic Target mode:

   ```
   run
   ```

   The above command runs the design in non-blocking mode, by default. To run the design in the blocking mode, use the following command:

   ```
   run -wait
   ```

   The `-nowait` option is supported only for compatibility with ICE mode. If this option is used with the `run` command, it is silently ignored.

   The first `run` command executed after the `configPM -dynamicTarget` command starts both the design clocks and the trace clocks. Subsequent `run` commands only start the trace clocks because design clocks are already running.

   To start the design clocks only, without starting the trace clocks, use the following command:

   ```
   run -emulation
   ```

Stopping a Design

- To stop a run in the non-blocking mode, use the `stop` command without any arguments.
- To stop a run in the blocking mode, use `Ctrl-C`.

In the xeDebug GUI, you can start and stop a run using the buttons in the *Run Control* frame on the *Control* tab, as shown in Figure 36-1 on page 1685.

Figure 36-1 Run Controls on Control Tab

© 2021 All Rights Reserved.

Switching between the Different Modes

The `configPM` command can be called any number of times during a session to switch between the Dynamic and Static Target modes.

- To determine the current mode, run the `configPM` command without any argument:

  ```
  configPM
  ```

- To switch from Dynamic Target mode to the Static Target mode, use the following command:

  ```
  configPM -staticTarget
  ```

  The above command stops any design clocks (if running) and configures the system back to the Static Target mode.

  Note: When switching from Dynamic Target mode to Static Target mode, the waveform data captured in the emulator but not uploaded to the workstation will be lost. To avoid the loss of waveform data, you must upload the waveform using the `database -upload` command before switching to Static Target mode.

  For more information about the `configPM` command, see the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

When running in the Dynamic Target mode, the testbench callbacks are not executed. When you switch to the Static Target mode, the following actions are taken at run time:

- All pending testbench callbacks are triggered.

  Note: When there are multiple instances of testbench callback trigger conditions, only the last one will be processed. It is your responsibility to handle the design requirements that require every trigger condition to be processed.

- The Xcelium software time is synchronized with the hardware time.

- The co-simulation interfaces get activated.

In the xeDebug GUI, the *Setup* tab has two radio buttons corresponding to the *Debug Mode* field within the *Configuration* frame (refer to the highlighted area in Figure 36-2 on page 1687). Select the appropriate radio button to switch between the Dynamic and Static Target modes.

© 2021 All Rights Reserved.

Figure 36-2 Switching Controls on Setup Tab

Assertions Support in Dynamic Target Mode

Although assertions handling in VXE mostly depends on a simulator, the Dynamic Target mode provides some support for handling assertions without using a simulator.

For information on how assertions are supported in the Dynamic Target mode, see Using Assertions in Dynamic Target Mode in Chapter 27, “Compiling and Running Designs with Assertions.”

Limitations of Dynamic Target Mode

Following are the limitations when using the Dynamic Target mode:

- The `force`/`release`/`deposit`/`value` requests and memory `read`/`write` operations are executed without suspending the design clocks; therefore, time coherency is not guaranteed.
- The following run-time features are not available in the Dynamic Target mode:
  - InfiniTrace
  - Saving and restoring design state
- The software simulator does not execute any commands in the dynamic target and non-blocking run modes, even though the command prompt is available.
- A number of assertion features require simulator support and hence are not available or do not function during a design run in the Dynamic Target mode. Following is a list of such features:
  - Failure action blocks
  - Reporting of assertion failures
  - Enabling or disabling of assertions
  - Assertion Control Tasks (ACTs)

© 2021 All Rights Reserved.

- `assertion -kill -onfailure` command
- Assertion summary is not available until the end of design run

© 2021 All Rights Reserved.

37

Importing Netlists as Blackboxes in SA Mode

In the conventional Simulation Acceleration flow, the IXCOM compiler performs design analysis, elaboration, synthesis checks to map portions of the design that can be accelerated in emulator hardware. For synthesizable netlist files, these checks and processes are unnecessary, and the netlist can be directly presented to the hardware-mapping process of the compiler.

Important

The use model described in this appendix is offered only for backward compatability. It is not recommended, and is likely to be obsolete in a future release. It is highly recommended that you use the process described in "Importing External Netlists in the SA Mode" on page 898.

This section describes the process for importing netlists as blackboxes in SA mode using the IXCOM compiler. This use model allows you to bypass the conventional Simulation Acceleration compilation flow.

This use model has the following limitations apply:

- Design must be swapped to the hardware mode at time 0 (at the beginning of the run), and run only in the hardware mode. Swapping back to the software mode is not possible because hot-swap capabilities are not available with this feature.
- Limited testbench access (read, write, force and so on) into the design objects in the imported netlist.

  Note: The User Data and commands for the ICE flow can be used. However, the scripts should be modified to reflect the hierarchical path created by the IXCOM compiler.
- Testbench access to memories in the imported netlist is currently not supported.

© 2021 All Rights Reserved.

Use Model for Importing a Netlist as Blackbox Using IXCOM

This section provides details for importing blackbox netlist in the IXCOM compiler. The following design modules are used to describe the functionality of this use model:

```
// File: tb.v
module Tb;
   reg clk;
   reg [1:0] din;
   wire [1:0] dout;
   initial
   begin
   // see sub-sections below for read, deposit and force access from testbench
   end
   // ...
   Dut d(clk, din, dout);
endmodule // Tb
```

```
// File: dut-netlist.v
module Dut ( clk, din, dout);
   input clk;
   input [1:0] din;
   output [1:0] dout;
   wire en; // internal net
   wire [1:0] fout; // internal net
   // ...
endmodule
```

To import the blackbox netlist into the emulator hardware by using the IXCOM compiler (with no full visibility to design objects in the imported netlist), the use model requires certain steps different from the conventional IXCOM compilation flow.

Note: The RTL modules instantiated within a netlist hierarchy should be specified as exported modules by using `NL_EXPORT`. When the external netlist configuration file does not contain `NL_EXPORT` for a module, the following error message is issued:

```
xmelab: *F,CUITOP: A defined top level module Chk has also been instantiated
```

Following are the steps to import netlists as blackboxes using the IXCOM compiler:

1. Create a shell module for every top-level design unit in the netlist, and enclose the shell modules with the `` `qtrefcel `` and `` `endqtrefcell `` compiler directives.

   ```
   // File::dut-shell.sv
   `qtrefcell
   module Dut(clk, din, dout);
      input clk;
      input [1:0] din;
      output [1:0] dout;
   endmodule // Dut
   `endqtrefcell
   ```

   Note: This shell module allows the IXCOM compiler to compile netlist as blackbox with the interface provided. By default, I/O interface provided in the shell file is the only visibility into netlist module at IXCOM compile time. You must explicitly specify additional visibility into netlist object as explained in Testbench Access to Netlist Objects on

© 2021 All Rights Reserved.

page 1692. The IXCOM compiler ignores all statements in this module except for the I/O interface and `$ixc_ctrl` statements.

2. Provide information about the netlist file to the Palladium Z1 back-end compiler (xeCompile) in the `externalNetlists.qel` file in the working directory.

```
// File: externalNetlists.qel
netlistFile {verilog dut-netlist.sv}
```

Note: The file specified using `netlistFile` must be Verilog. You can use a file with other formats by converting the file to a Verilog netlist file through HDLICE.

3. Compile source files along with the shell module files.

```
ixcom tb.sv dut-shell.sv -ua +dut+Dut
```

For the above compile, IXCOM generates the following kind of design hierarchy:

```
// Tb module (from tb.sv)
module Tb ;
   // ...
   Dut_xcm2 dut (,,);
endmodule

// Dut module from shell file (dut-shell.sv)
module Dut_xcm2 ( clk, din, dout);
   // pragma CVASTRPROP MODULE HDLICE HDL_MODULE_ATTRIBUTE "0 vlog"
   input clk;
   input [1:0] din;
   output [1:0] dout;
   // ...
   // Dut instantiation of the original module from netlist
   Dut i0(clk, {din[1], din[0]}, {dout[1], dout[0]});
endmodule

// Dut from original netlist file (dut-netlist.sv)
module Dut ( clk, din, dout);
   input clk;
   input [1:0] din;
   output [1:0] dout;
   // ...
endmodule
```

For the above design, the hierarchical reference to the netlist object must include an additional `i0` hierarchy introduced by the IXCOM compiler. For example, `Tb.dut.i0.clk` refers to the `clk` signal in the `Dut` netlist module.

Compile a netlist module imported as a top-level design unit as follows:

```
ixcom dut-shell.sv -ua +dut+Dut
```

The shell module name is modified by the IXCOM compiler to differentiate the shell module from the actual netlist module. For the above compile, the IXCOM compiler creates the following design hierarchy:

```
module Dut_ ( clk, din, dout);
   // pragma CVASTRPROP MODULE HDLICE HDL_MODULE_ATTRIBUTE "0 vlog"
   input clk;
```


© 2021 All Rights Reserved.

```
   input [1:0] din;
   output [1:0] dout;
   // ...
   Dut i0 ( clk, { din[1], din[0]}, { dout[1],dout[0]});
endmodule
```

For the above design, the hierarchical reference to the netlist object needs to include both the modified module name in addition to the `i0` hierarchy introduced by the compiler. For example, `Dut_.i0.clk` refers to the `clk` signal in the `Dut` netlist module.

Testbench Access to Netlist Objects

For the top-level port interface objects, IXCOM automatically infers the primary input, output, or inout (bidirectional) property, based purely on the direction of the ports of this shell module. A port with input direction is treated as primary input; a port with output direction is treated as primary output; the port with inout direction is treated as a bidirectional signal, which has drivers from both DUT and testbench side. The marking of inout (when it is really not a bidirectional signal) signals do not affect the functionality, not marking a bidirectional signal as inout would produce incorrect results. However, it might have a performance impact as bidirectional property adds two extra pins at the interface.

Internal objects refer to nets (or signals) and registers in the imported netlist are the ones not specified in the port interface. If there are any references to internal nets that have been optimized away or cannot be found, an error message is generated.

To access (read and write) internal netlist objects from testbench, you need to specify access information in the shell file as follows:

1. Create a shadow object in the shell module defined within the `` `qtrefcell `` and `` `endqtrefcell `` compiler directives.
2. Use the `$ixc_ctrl(...)` statement to bind (or connect) the shadow object to the netlist object.
3. Use the `$export_read` and `$export_deposit` directives in an `initial` statement of the testbench to explicitly give access information.
4. Access the netlist object with hierarchical reference in the testbench (using HDL statement or primitives).

   The `ixc` control statement uses the following syntax:

   ```
   initial
   $ixc_ctrl("<tag>", "[<declaration>;] <primitive instantiation>;");
   ```

   Where:

   - <*tag*> is always set to `hdlICE` for later use by back-end compiler.

© 2021 All Rights Reserved.

- *<declaration>* is SystemVerilog declaration for the shadow object.

  Note: This part can be optional in some control statements.

- *<primitive instantiation>* is used to connect shadow objects to the objects in the netlist. This primitive can be any one of the allowed `ixc` primitive.

Reading Netlist Objects from Testbench

To read an internal netlist object, use the following steps:

1. Create a shadow object in the shell module for the read access to the netlist object.
2. Use the `$ixc_ctrl(...)` statement to bind shadow object to the netlist object.
3. Use `$export_read(<signal path>)` in an `initial` statement of the testbench to explicitly give access information. Where, *<signal_path>* is hierarchical reference to the shadow object.
4. Access (read) the netlist object using hierarchical reference to the shadow object.

Following is an example illustrating the read access to an internal netlist object using the above steps:

```
// File: tb.v
module Tb;
   // ...
   initial
   begin
      $export_read(d.en_QTRC);
   end
   initial
   begin
      if (d.en_QTRC) // access to Dut.en through shadow object
      // ...
   end
   // ...
   Dut d(clk, din, dout);
endmodule // Tb

// File: dut-shell.v
`qtrefcell
module Dut(clk, din, dout);
   input clk;
   input [1:0] din;
   output [1:0] dout;
   reg en_QTRC; // shadow object
   initial
   // bind shadow object to object in the net-list
   $ixc_ctrl("hdlICE",
   "wire en_QTRC; ixc_bind #(1) (en_QTRC, i0.en);");
endmodule // Dut (shell)
`endqtrefcell
```


© 2021 All Rights Reserved.

Write (or Deposit) to Netlist Object from Testbench

To deposit to an internal netlist object:

1. Create a shadow object in the shell module for netlist object that needs to be deposited.
2. Use `$ixc_ctrl(...)` statement to bind shadow object to netlist object.
3. Use `$export_deposit(<signal path>)` in an `initial` statement of the testbench to explicitly give access information. Where, `<signal_path>` is hierarchical reference to the shadow object.
4. Deposit to netlist object through reference to shadow object.

Following is an example illustrating the write (or deposit) access to an internal netlist object using the above steps:

```
// File: tb.v
module Tb;
   // ...
   initial
   begin
      $export_deposit(d.fout_QTRC);
   end
   initial
   begin
      // ...
      d.fout_QTRC = 2'b11; // deposit to internal netlist object
   end
   // ...
   Dut d(clk, din, dout);
endmodule // Tb
```

```
// File: dut-shell.v
`qtrefcell
module Dut(clk, din, dout);
   input clk;
   input [1:0] din;
   output [1:0] dout;
   reg [1:0] fout_QTRC; // shadow object
   initial
   // bind shadow object to netlist object
   $ixc_ctrl("hdlICE",
   "wire [1:0] fout_QTRC; ixc_bind #(2) (fout_QTRC[1:0], i0.fout[1:0]);");
endmodule // Dut (shell)
`endqtrefcell
```

Forcing Netlist Objects from Testbench

To force an internal netlist object:

1. Create a shadow object in the shell module for netlist object that needs to be forced.
2. Use the `$ixc_ctrl(...)` statement to bind shadow object to netlist object.

© 2021 All Rights Reserved.

3. Use `$export_deposit(<signal path>)` in an `initial` statement of the testbench to explicitly give access information. Where, `<signal_path>` is hierarchical reference to the shadow object.

4. Force the netlist object through reference to the shadow object.

Following is an example illustrating the access to force an internal netlist object using the above steps:

```
// File: tb.v
module Tb;
   // ...
   initial
   begin
      $export_deposit(d.fout_QTRC);
   end
   initial
   begin
      // ...
      #1 force d.fout_QTRC = 2'b11; // force to internal netlist object
   end
   // ...
   Dut d(clk, din, dout);
endmodule // Tb
```

```
// File: dut-shell.v
`qtrefcell
module Dut(clk, din, dout);
   input clk;
   input [1:0] din;
   output [1:0] dout;
   reg [1:0] fout_QTRC; // shadow object
   initial
   // bind shadow object to object in the net-list
   $ixc_ctrl("hdlICE",
   "wire [1:0] fout_QTRC; ixc_bind #(2) (fout_QTRC, i0.fout[1:0]);");
endmodule // Dut (shell)
`endqtrefcell
```


© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

38

Design Debugging Processes

This chapter discusses how the design database, derived after compilation, can be run and debugged using the xeDebug and *xrun* tools in the SA and ICE modes using script or command mode.

To use the xeDebug GUI for debugging designs, refer to Appendix 42, “Using the xeDebug GUI” for information.

For detailed information on probing and generating waveforms, refer to Probing Design Signals and Generating Waveforms.

This chapter includes the following sections:

- Basics of Starting a Debug Session
  - Invoking the Debugger Tool(s)
  - Running Commands Remotely through xeclient
  - Command Sequence for ICE Mode
    - Debugging Process for Vector Debug
    - Debugging Process for LA/STB
  - Debugging Process for SA Mode with xeDebug
- Dynamic Swap between Hardware and Software in SA Mode
  - Using Hot Swap Options
  - Using Hot Swap Run Modes
  - Software Simulation Mode
  - Run Mode
  - TBrun Mode
  - C-TestBench Run Mode (CTB Run Mode)

© 2021 All Rights Reserved.

- Autorun Mode
- Match Mode
- Accessing Signal Values
  - Forcing Values
  - Releasing Values
  - Depositing Values
- Files Generated during Debugging
- Debugging Designs with Modular Compilation
- Improving Debug Turnaround Time with Debugging in Regression Mode

Basics of Starting a Debug Session

Overall, before starting to debug the design, you need to set up the debug session and the conditions for launching the Debugger tool. Then, launch the Debugger tool to begin the debugging operations. The set-up part of the debugging operations includes setting probes, specifying the conditions that will cause a trigger, setting vector files, and so on. After the set up, run the design to obtain test results of the run. These test results can be viewed and analyzed using the Debugger browser tools. You can use the browser tools to traverse the design hierarchy and select design objects such as nets, memories, instances or HDL source lines for debugging purposes.

Note: During a debug session, the set-up and run steps can be repeated in any order. For example, you can set the probes and trigger, then run the design, then change probes and trigger, run again, view waveforms, and so on.

Designs can be debugged in an online session (while the emulator connection is available) and offline session (when the connection to the emulator is not available). In both type of sessions, the test results can be viewed in the browsers.

The process of debugging and its steps change slightly, depending on whether you are using LA, STB or VD mode, or whether your design contains memories.

Terminating a User Debug Session from UNIX command prompt

You normally should use the `exit` XEL command to end a debug session. However, if you must terminate a debug session from UNIX command prompt, you can use the `kill` command on the UNIX prompt as follows:

© 2021 All Rights Reserved.

```
% kill <xeDebug_pid>
```

Or,

```
% kill -15 <xeDebug_pid>
```

Or,

```
% kill -TERM <xeDebug_pid>
```

Note: You can identify the <*xeDebug_pid*> from the `./tmp/session.log` file available in the debug session directory.

Invoking the Debugger Tool(s)

For running and debugging the designs in SA mode, use either *xrun* or xeDebug tool. In ICE mode, use only the xeDebug tool.

For information about invoking either of these tools, refer to Appendix 3, "Invoking Palladium Z1 Compilers and Debugging Tools."

Running Commands Remotely through xeclient

You can execute the XE commands remotely through the XE server for various debugging tools. Use the `xeclient` script to connect to the XE server from a terminal to run the commands on a tool running on another terminal. The script can be used for all XE tools: xeCompile, xeDebug, and vvmDebug. You can connect to the XE server in both GUI and non-GUI modes. To connect to the required tool, you should know the port of the XE server.

You can run the `xeclient` script either on the XE server or any other host, as required. The script can be run from any directory, as follows:

```
xeclient [<hostname>:]<port> <command>
```

You can also specify "`-`" as the port number.

- In the non-GUI mode, source the `xeserver.tcl` script available at <*VXE_INSTALL_HOME*>`/share/vxe/etc/qel/xeserver.tcl`. The script creates a new server and prints the port on the screen. It also creates the `.xe_server_port` file in the design directory.

  For example, you can use the `xeserver.tcl` file in xeDebug as follows:

  ```
  XE> source <VXE_INSTALL_HOME>/share/vxe/etc/qel/xeserver.tcl
  XE server port: 56522
  ```

  Then, use the following command on the UNIX command-line prompt to run the `xeclient` script:

© 2021 All Rights Reserved.

```
% xeclient 56522 "debug ."
```

- In the GUI mode, the `.xe_server_port` file is created by default in the design directory. You can either use the port number specified in the `.xe_server_port` file or specify "`-`" as the port number while executing the `xeclient` script. When you specify the port number as "`-`", the script will use the default port number from the `.xe_server_port` file.

The following exit codes are returned to help you determine the issues while running the `xeclient` script:

- `0`: OK
- `1`: Syntax error
- `2`: Unreadable port file
- `3`: xeDebug connection error
- `4` - Error in the Tcl command

You can also define a server port according to your requirements instead of using the server port generated by the `xeserver.tcl` script. Use the `XE_SERVER_PORT` environment variable to define a server port. This variable is honored by both the `xeclient` command and the `xeserver.tcl` script. The `XE_SERVER_PORT` environment variable is supported in both GUI and non-GUI mode.

Command Sequence for ICE Mode

To run a design in ICE mode, you need to perform the following steps:

1. Open a debug session and specify the directory path to the design database:

   ```
   debug <dbPath>
   ```

   For example:

   ```
   debug .
   ```

   Important

   Different directories can be used for the Compiled database and for running the Debugger tool so that the Debugger files created during emulation, are separated from the design database files.

   Only one debug session can be run from a single directory. However, multiple debug sessions can be run from different debug directories using the same design directory.

2. Specify the emulation host and download the design:

© 2021 All Rights Reserved.

```
host .
download
```

Note: The emulation host can be alternatively specified using the `xeDebug -host` command.

3. Specify the debugging mode as VD, STB, or LA (for ICE designs only):

   ```
   configPM -la|-vd|-stb
   ```

   For example:

   ```
   configPM -la
   ```

4. Set up the target connection parameters, and clock parameters, if required.
5. Optionally, if the design contains memories, initialize the memories before starting any testing.

The steps mentioned above are mandatory. After these steps, you can execute various set of commands for debugging operations, as documented in the *VXE Command Reference Manual*.

Debugging Process for Vector Debug

1. Open the design. Specify the emulation host and download the design to the emulator.
2. Specify the Vector file (the file can be in any one of the accepted vector formats).

   Add probes, if desired.

   Specify break condition through SDL, or number of cycles to run, if desired.

   Run the design, and observe data on the waveform display. The `run` command also reports back whether or not the actual results match expected results.
3. The Waveform browser automatically displays the comparison between expected and actual results, highlighting any mismatches.

Determining the Vector Debug Compare Methodology

You can use a comparison methodology that involves both a stimulus memory and output results memory in the hardware. This methodology provides fast data throughput as compared to the legacy VD compare.

A dedicated results memory is used for comparison in place of uploading the results to a Vector Binary Format (VBF) file — the waveforms will not be uploaded by default.

© 2021 All Rights Reserved.

The VD compare method does not capture the input results. Therefore, those values cannot be compared with the expected results. In this case, the comparison information only reflects the total of outputs and inout pins, and not the total of all pins as previously reported in the legacy compare method:

```
INFO (legacy-####): Compared <n> signals out of <n> signals in stimulus file.
```

You can perform the following operations in the Vector Debug mode according to your requirements:

- Enabling Stimulus and Waveform Upload
- Enabling Legacy Vector Debug Compare
- Breaking Runs on Miscompares
- Skipping Waveform Upload and Results Comparison
- Performing Detailed Vector Debug Comparison
- Viewing Timestamps in Waveforms and Message Logs
- Verifying Pins in the Stimulus and Designs

Enabling Stimulus and Waveform Upload

By default, stimulus and waveform upload is disabled and the following message is displayed:

```
INFO (legacy-52519): Skipping input waveform creation to improve performance.
```

For both legacy and new VD compare, to enable the stimulus and result waveforms generation, execute the `xeset vdCreateWaves 1` command prior to executing the `vector` command. Alternatively, select the *Autowave* option in the *Control* tab of the xeDebug GUI to enable waveform generation.

If the `xeset vdCreateWaves 1` command is executed after the `vector` command, result waveforms will be uploaded for all subsequent runs after the `xeset` command. However, no stimulus or expected waveforms will be produced.

Note: If FSDB format is used, waveforms will not be displayed if either `xeset vdCreateWaves 1` is executed or *Auto Upload* option is enabled in the *Control* tab after the `vector` command. A message similar to the one shown below is displayed:

```
Warning: Unable to display graphical waveform comparison: Input waveform database
DTS.cbc.in.fsdb does not exist. To ensure input waveform database is created,
execute 'xeset vdCreateWaves 1' or specify 'AutoWave' before the 'vector' command.
ERROR (legacy-45299): Display Result Waveform failed.
```


© 2021 All Rights Reserved.

Enabling Legacy Vector Debug Compare

You can enable the legacy VD compare method using the `xeset vdLegacyCompare 1` command.

Ensure that you execute the `xeset vdLegacyCompare 1` command before executing the `vector` command. Otherwise, an error message will be displayed:

```
ERROR (legacy-52595): Vector Debug comparison mode may not be changed after vector
command.
```

For details, refer to the description of the `xeset vdLegacyCompare` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Example:

```
download
xeset vdLegacyCompare 1
vector file.cbc
run
...
```

In the above example, the input signals in the stimulus file will also be compared.

Breaking Runs on Miscompares

You can determine if a run should continue on miscompares or whether the individual miscompared signals should be identified. To stop the run when a miscompare is detected, use the `xeset vdBreakOnMiscompare 1` command.

To break the current run when a miscompare is detected, execute the `run -BreakOnMiscompare` command.

Example:

```
vector file.cbc
run -BreakOnMiscompare
...
Or
```

```
vector file.cbc
xeset vdBreakOnMiscompare 1
run
...
```

Run will be interrupted shortly after the mismatch instead of completing all the vectors because the vectors are processed from a buffer and the break happens after the vectors in the current buffer are exhausted.

The `xeset vdBreakOnMiscompare 1` and `run -BreakOnMiscompare` commands cannot be used if the legacy VD comparison method is enabled with the `xeset vdLegacyCompare 1` command. If these commands are used together, the following

© 2021 All Rights Reserved.

warning is displayed:

```
WARNING (legacy-13131): BreakOnMiscompare functionality is not possible using
legacy comparison handling and '-breakOnMiscompare' will be ignored. The legacy
comparison will be performed.
```

For more information about these commands, refer to the description of the `xeset vdBreakOnMiscompare` and `run -BreakOnMiscompare` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Skipping Waveform Upload and Results Comparison

Waveform upload and results comparison can be skipped and only the stimulus can be loaded to improve the performance.

To load only the stimulus and skip waveform upload and results comparison, use the `xeset vdSkipCompare 1` command. Use this command if you prefer alternate methods of validating the results, such as assertions. To skip the waveform upload and comparison of the actual results with expected results for a specific run, use the `run -skipCompare` command.

Performing Detailed Vector Debug Comparison

You can get a detailed comparison of the actual results and the expected results for all runs using the `xeset vdDetailCompare 1` command. When this command is used, detailed comparison of specific pin mismatches are displayed.

To get a short comparison of the actual results with expected results for a specific run, use the `run -noDetailCompare` command,

If either `run -noDetailCompare` or `xeset vdDetailCompare 0` command is used, and if any mismatch is found between the actual and expected results, the detailed comparison is not done and the following message is displayed for mismatch:

```
INFO (legacy-52574): Vector comparison failed.
xxx cycles had mismatches in the last vector comparison. Enable detailed
comparisons to see specific pin mismatches.
```

Example:

```
vector <file.cbc>
run -noDetailCompare
...
or
vector <file.cbc>
xeset vdDetailCompare 1
run
...
```


© 2021 All Rights Reserved.

Viewing Timestamps in Waveforms and Message Logs

The system enables you to get the absolute timestamps (ns/ps) in message logs (`*.cbc.failurelog`). Waveforms can also be configured to display `ns` as unit instead of `fclk`.

To get the timestamps in message logs and waveforms, execute the `xeset vdTimeStamp 1` command at run time. By default, the `vdTimeStamp` run-time parameter is set to 0, that is, false, So, the `fclk` cycles are displayed in waveforms and message logs.

Once you set `vdTimeStamp` to 1 (that is, `true)`, the value cannot be changed back to 0, and the following warning will be displayed:

```
WARNING(legacy-52740): The variable vdTimeStamp can only be changed from false to
true. Keep vdTimeStamp as true.
```

Figure 38-1 Waveform with Timestamps

Examples: To generate time stamps in SST2 or FSDB waveforms in online VD mode and in failure log file:

Sample file: `test1scan.cbc`

```
debug .
host .
download
configPM -vd
xeset vdTimeStamp 1
```


© 2021 All Rights Reserved.

```
vector test1scan.cbc
run
resultWaveform -display


debug .
host .
download
configPM -vd
xeset vdTimeStamp 1
vector test1scan.cbc.in - {}
resultWaveform -display
```

Example: To generate time stamps in SST2 or FSDB waveforms in offline VD mode:

Sample file: `test1scan.cbc.phy`

```
debug .
host -offline test1scan.cbc.phy
xeset vdTimeStamp 1
probe *
database -upload
```

For more information, refer to the description of the `xeset vdTimeStamp` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Precedence Conflicts of Vector Debug Commands

The system displays appropriate warnings if precedence conflicts arise while executing the `xeset` and run command options in the Vector Debug mode.

- The `xeset vdSkipCompare`, `run -skipCompare`, `xeset vdBreakonMiscompare`, and `run -BreakOnMiscompare` commands will be ignored with a warning, if used after `xeset vdLegacyCompare` 1.

```
xeset vdLegacyCompare 1
download
vector file.cbc
xeset vdSkipCompare 1
xeset vdBreakonMiscompare 1
run
...
```

or

```
xeset vdLegacyCompare 1
download
vector file.cbc
run -BreakOnMiscompare -skipCompare
...
WARNING (legacy-52551): 'legacyCompare' will take precedence over conflicting
settings: 'skipCompare & breakOnMiscompare' will be ignored.
```


© 2021 All Rights Reserved.

- The following commands can be used after executing the xeset vdLegacyCompare 1 command: xeset vdCreateWaves 1 and xeset vdDetailCompare 1. For example,

```
xeset vdLegacyCompare 1
download
xeset vdCreatewave 1
vector file.cbc
xeset vdDetailCompare 1
run
...
```

- If you set both xeset vdSkipCompare 1 and xeset vdBreakonMiscompare 1, the xeset vdSkipCompare command takes precedence over the xeset vdBreakonMiscompare command.

Precedence Conflicts of Run Command Options

The order of precedence of the run command options for the Vector Debug mode is as follows:

1. run -skipCompare
2. run -BreakOnMiscompare
3. run -noDetailCompare

While processing a run command, appropriate messages of conflicts are printed for precedence of the run command options.

For example, if both run -skipCompare and run -BreakOnMiscompare commands are specified, the following message will be printed:

```
WARNING (legacy-1111): skipCompare will take precedence over conflicting settings:
-BreakOnMiscompare will be ignored.
```

Note: If both xeset command and vector run command option are in use for an operation, the run command operation silently overrides the xeset command operation.

Examples:

```
...
xeset vdSkipCompare 0
run -skipCompare
...
...
xeset vdBreakOnMiscompare 0
run -breakOnMiscompare
...
...
xeset vdDetailcompare 1
```


© 2021 All Rights Reserved.

```
run -noDetailCompare
...
```

Verifying Pins in the Stimulus and Designs

Pins described in the stimulus CBC/VBF file can be verified for consistency with declaration in the design file that is described in the `dbFiles/<design>.io` file. By default, this check is disabled.

To enable the check to verify the consistency, set the `XE_VD_ENABLE_VBF_DIRECTION_CHECK` environment variable, as follows:

```
export XE_VD_ENABLE_VBF_DIRECTION_CHECK=1
```

or

```
setenv XE_VD_ENABLE_VBF_DIRECTION_CHECK 1
```

If `XE_VD_ENABLE_VBF_DIRECTION_CHECK` variable is set, you get appropriate warning messages if any discrepancy is found between the pin usage described in the CBC/VBF file and the usage in the design, such as:

```
WARNING (legacy-####): Pin ‘P1' listed in dbFiles/io.Top outputs section is used
as input in stim.cbc.in.
WARNING (legacy-52565): Pin 'col_clear' listed in ./dbFiles/TOP.io outputs section
is used as BIDIR in DTS.cbc.in
```

Debugging Process for LA/STB

1. Open the design. Specify the emulation host and download the design to the emulator.
2. Add probes.
3. Using SDL, define a trigger for stopping the clocks in STB mode or for stopping data collection in LA mode.
4. Initialize contents of all memory instances to 0 or 1, or load contents of specific memory instances from files.
5. Force some nets to specific constant values.
6. Run the design.
7. Inspect values of specific nets.
8. Inspect contents of memory instances, or edit their contents.
9. Upload probed data for viewing in the waveform browser.
10. Inspect the design using the waveform tools and/or by reading memories.

© 2021 All Rights Reserved.

Multiple User Sessions

In all modes, the xeDebug tool allows you to set up multiple user sessions. Each session is associated with a directory where the results are stored. These sessions are stored in automatically-created subdirectories under the debug directory (which is the directory from which you started the debugger). These subdirectories store all the files associated with that particular user session. Thus, you can have multiple output files, but each will be stored in the specific directory associated with the session.

In multi-user sessions, opening a user session switches the `xeDebug.log` to the session directory. If the log file was already opened by another user, then the log is created with a temporary name and will become `xeDebug.log` after opening a session in a subdirectory. This log file contains an exact copy of your terminal or console window. The file is incremental over multiple debug sessions.

Figure 38-2 Sample Multiple Session Hierarchy

You can specify any text as the session name. The session directory is named `<session_name>.session` file. All output files for these user sessions are stored in the appropriate session directory. Sessions must be released when moving from the online debugging session to the offline debugging session (or vice versa).

Note: From this point onwards, the online debugging session will be referred as online session and the offline debugging session will be referred as offline session

© 2021 All Rights Reserved.

Debugging Process for SA Mode with xeDebug

To run a design in SA mode with xeDebug, perform the following steps:

1. When you invoke the xeDebug tool, you must explicitly start a debug session using the following command:

   ```
   debug <dbPath>
   ```

   For example:

   ```
   debug .
   ```

2. Specify the emulation host to which the design should be downloaded:

   ```
   host <host>
   ```

   The specified emulation host must have an InfiniBand channel to enable its interaction with Xcelium. In SA mode, the `download` and `configPM` commands are executed implicitly by the VXE software during swap-in (when switching from pure simulation to SA mode). Therefore, you need not execute these commands separately as required in ICE mode.

3. When you have finished running the design on the simulator and want to run it on the emulator, you can *hot swap* by using the `xc on` command to specify the run mode (run, tbrun, autorun). Then, use the `run -swap` command to complete the *hot swap*.

   For example:

   ```
   xc on -run
   run -swap
   ```

4. If running the design under *autorun* mode, use the `configPM` command to switch between the Static Target and Dynamic Target debugging modes. Use the following command after *hot swap* for this action:

   ```
   configPM -staticTarget|-dynamicTarget
   ```

5. Set up the target connection parameters, and clock parameters, if required.

6. Add probes.

7. Using SDL, define triggers for stopping the clocks.

8. If the design contains memories, initialize contents of all memory instances to 0 or 1, or load contents of specific memory instances from files.

9. Force some nets to specific constant values.

10. Run the design. You can dynamically swap the design from the hardware to software and back again while running a design. For details, refer to Dynamic Swap between Hardware and Software in SA Mode.

© 2021 All Rights Reserved.

11. Inspect values of specific nets.
12. Inspect contents of memory instances, or edit their contents.
13. While running the design on the hardware, upload probed data for viewing in the waveform browser.
14. Inspect the design using the waveform tools and/or by reading memories.
15. To run a design in SA mode using *xrun*, the steps are similar to using xeDebug.

Tip

You can specify a timeout duration for the simulator to wait before exit if the simulator turns unresponsive. Use the `XE_XMSIM_EXIT_TIME` environment variable at run time to specify the wait time in seconds, as follows:

```
setenv XE_XMSIM_EXIT_TIME 20
```

For more information, refer to the description of `XE_XMSIM_EXIT_TIME` environment variable in Environment Variables Required in Specific Cases section.

Debugging Process for SA Mode with *xrun*

To run a design in SA mode with *xrun*, perform the following steps:

1. When you invoke *xrun* without the `-xedebug` option, it is not necessary to specify a debug session or a host. The tool implicitly executes the following commands:

   ```
   debug .
   host .
   ```

   In addition, the tool implicitly executes the `configPM` and `download` commands.

2. The rest of the steps are just like running a design in SA mode with xeDebug. Refer to Debugging Process for SA Mode with xeDebug for the remaining steps.

   For example:

   ```
   xc on -run
   run -swap
   probe -create dut -depth all
   sdl -enable
   sdl -verify
   force dut.sig1 1
   value dut.sig2
   run
   database -upload
   ```

Also, refer to Chapter 25, “Compiling and Running Simulation Acceleration Designs with *xrun*” for details specific to running SA designs with *xrun*.

© 2021 All Rights Reserved.

Dynamic Swap between Hardware and Software in SA Mode

The presence of the Xcelium simulator enables support for enhanced Hot Swap capabilities in the use models where the design clocks can be stopped. During Hot Swap, the simulator takes full control and the emulator is released. Later, the design can be swapped back to the emulator (as available) and the emulator takes control.

Important

As a deviation from the traditional SA mode, hot swap is not supported when an ICE target is present. This means that you have to start running the design on the hardware from the beginning of the simulation time.

Using Hot Swap Options

The `xc on -xt0` and `xc on -xt1` commands specify how the signals with simulation values of `X`, `W`, `-`, or `U` (VHDL `STD_LOGIC` or `STD_ULOGIC`) or `X` (Verilog nets or registers) are to be handled in either of the following situations:

- Anytime during simulation acceleration modes, if an input to the DUT goes to any of these values, or if an attempt is made to write these values to a DUT signal **or**
- At the time the design is swapped from software mode to simulation acceleration modes, if a VHDL signal or Verilog register signal happens to be at one of these values.

The `xc on -xt0` command sets the hardware to translate their values to 0 if any of these values exist in the preceding situations. DUT inputs and DUT signals written to with these values will be assigned 0 during simulation acceleration modes. VHDL signals or Verilog registers with any of these values will get 0 values instead when swapped to simulation acceleration modes.

If a signal has simulation value of `Z` in either of the situations defined above for `xt0` and `xt1`, the `xc on -zt0` command sets the hardware to translate the signal values to `0` and the `xc on -zt1` command translates the values to `1`.

There is a limitation while probing signals when using hot-swap. At the time of hot-swap, xeDebug must stop probing or remove the probes located in the DUT and create new probes in hardware. However, you can save probes as a Tcl script and restore probes right after swap or any time later.

© 2021 All Rights Reserved.

Figure 38-3 Hot Swap Options on Setup Tab of xeDebug GUI

Using Hot Swap Run Modes

xeDebug tool starts in software mode, and then you can switch to the four available hardware modes. To switch between the available hardware modes, you must first swap back to the software mode by using the `xc stop` or `xc off` commands, and then switch to the new hardware mode.

For running a design in SA mode, the following run modes are available:

- Software Simulation Mode
- Run Mode
- TBrun Mode
- NBrun Mode
- C-TestBench Run Mode (CTB Run Mode)
- Autorun Mode
- Match Mode

Before switching to the TBrun, NBrun, or Autorun mode, ensure a clock generator is created in the DUT module. For details, refer to the Modeling Clocks in SA Mode section.

XE Prompts on xeDebug Tool

The xeDebug tool has four `XE` prompts. The `XEsim>` and `XErun>` are the two main prompts on xeDebug. The `XEin>` and `XEout>` are the two transitional prompts on xeDebug.

The `XE` prompts on xeDebug indicate the following:

- The `XEsim>` prompt indicates that the design is in the software mode.
- The `XErun>` prompt indicates that the design is in the hardware mode.

© 2021 All Rights Reserved.

- The `XEin>` prompt indicates that you have requested to swap the design into the hardware mode, but the xeDebug tool is still in the software mode. The design is swapped to the hardware when you execute the next `run` command. The attempt to swap the design to hardware might fail because of the `X` or `Z` values, or unfinished `init` block.
- The `XEout>` prompt indicates that you have requested to swap the design into the software mode, but the xeDebug tool is still in the hardware mode. The design is swapped to the software when you execute the next `run` command. The attempt to swap the design to software might fail because of the `X` or `Z` values, or unfinished `init` block.

For example,

```
XE> debug .
XE> host .
XEsim> xc xt0 zt0 run
XEin> run -swap
XErun> xc off
XEout> run 2
XEsim>
```

Software Simulation Mode

In this mode, the whole design runs in software; nothing is accelerated with the emulator. This is the default mode, in which the xeDebug tool starts.

If your design contains `initial` blocks in DUT modules, you should run in software simulation until all events generated by those blocks are simulated. If you attempt to swap to simulation acceleration modes while events are still pending, unless you use the `xc on -initok` option, you get an error by default.

Run Mode

The parts of the design that meet certain requirements are accelerated with the emulator; the other parts stay in the software.

To run in this mode use the following command:

```
xc on -run
```

This mode is the general purpose acceleration execution mode. The complete successfully compiled design can be accelerated in this execution mode. Both testbench and DUT events are processed and executed in this mode. Although no event is ignored, the accelerated parallel event simulation algorithm can have different scheduling order than the software and

© 2021 All Rights Reserved.

thus have different results. For most designs that are race free and compatible with two-state simulation, the `xc on -run` mode should produce the same result as software simulation.

Alternatively, execute the following command before specifying the `xc on -run` command:

```
xc checkxon
```

The `xc checkxon` and `xc checkxoff` commands can be specified interchangeably to disable or enable X checking. However, by default X checking is disabled.

TBrun Mode

In the TBrun mode, the hardware and software schedulers always block each other. These are synchronized by the nearest delays in each domain. While advancing the hardware time, the software time is blocked until the hardware stops. Likewise, while advancing software time, the hardware is blocked until the nearest software delay elapses. At any point in time, either the hardware or the software can advance time; both cannot advance together. This lock-step mechanism ensures that all the software-to-hardware and hardware-to-software events are transferred immediately as these happen based on the same time wheel. This is called simulation congruency.

The TBrun mode only works for designs with the following restrictions:

- Design compiled with clock generator in DUT (`+iscDelay`) can use the `xc on -tbrun` mode.
- The output events generated from DUT are processed and propagated into the testbench only at the following testbench synchronization events:
  - **a.** `tbcall` (`tbcall_region`) or import DPI call events.
  - **b.** Time delay events scheduled by testbench.
  - **c.** Special output events explicitly defined using `$export_event` compiler directives.

  The `xc on -tbrun` mode can run close to peak simulator speed if the testbench synchronization events happen at low frequency and requires low processing overhead.

In `xc on -tbrun` mode, only the next scheduled time event in the testbench module (including SystemC- and PLI-based time events) is considered. Then, fast emulation speed like `xc on -autorun` mode is used to run the hardware from the current time to this next scheduled event.

You can have PI and PO signals in your design, but they are not used to control the hardware execution in `xc on -tbrun` mode. PI and PO signals are *enabled-on-sync*, which means they are enabled whenever *hardware and software synchronize*, for example:

© 2021 All Rights Reserved.

- at `axis_tbcall`
- at a DPI-call
- at scheduled `#delay` events before `xc on -tbrun` or from an `axis_tbcall` or DPI-call task during `xc on -tbrun`.

If you schedule testbench events in very short intervals, then `xc on -tbrun` mode is executed like the `xc on -run` mode automatically. If you schedule testbench events with a long delay, then `xc on -tbrun` provides the same performance as `xc on -autorun`.

In `xc on -tbrun` mode, you can have both `#delay` and mode swap control from the CLI just as in `xc on -run` mode.

To run a design in `xc on -tbrun` mode:

1. Compile the entire design as the DUT, leaving no module out.
2. Use `#delay` statement to model clock generation and use ixcom switch `+iscDelay` to compile the clock generator.
3. Use the `axis_tbcall` primitive/procedure for file I/O operations (`$readmemb`, `$readmemh`, `$display`) or system control functions (`xc stop` and `xc off`).

To stop `xc on -tbrun` mode and return to software simulation mode:

```
xmsim> run 500000
xmsim> xc on -xt0 -tbrun
```

You not only get back to the CLI at time 500000, but you also keep the design in the hardware. You have full control of the system from the CLI just like `xc on -run` mode.

Note:

- Only the hardware output events are transferred to the testbench at testbench-scheduled time events. You must wait for time, not output events, to *wake up* testbench activity. To use a DUT output event to wake up the testbench, you must use `axis_tbcall` or a DPI call for the task.
- Do not use the `xc on -tbrun` command with an embedded software module in the DUT because it requires P/I-P/O events to work. Therefore, you should only use the `xc on -run` command with such modules.

Example

For the following code:

```
initial
  do some PI activities with short delay;
```


© 2021 All Rights Reserved.

```
#1000000;
do some other PI activities with short delay;
```

If you are using the `xc on -tbrun` command, when there are PI activities, the emulator will run the same way as when the `xc on -run` command is issued. During the long-delay period of #1000000, the emulator will run the same way as when the `xc on -autorun` command is issued.

NBrun Mode

In the NBrun mode, both software and hardware domains advance the time concurrently instead of alternately. Therefore, this run mode is called the non-blocking run mode or NBrun mode. This mode is in complete contrast to the TBrun mode.

After the DUT is swapped into the emulator, the Xcelium process spawns a hardware control thread, which advances the hardware time immediately until the first TBCall happens. The software testbench also advances the time in parallel after the DUT state is swapped. In the NBrun mode, the timestamps in the software testbench and hardware DUT are not synchronized and can vary randomly from one session to another. It will not function correctly if the communication between the testbench and DUT is based on relative times. Therefore, the NBrun mode is not simulation congruent.

The software testbench can also send DUT events asynchronously. These asynchronous events force the hardware control thread to stop the hardware DUT time immediately to accept these new DUT events.

Once the hardware time is stopped for TBCalls or asynchronous software events, the hardware waits until the hardware control thread detects that the hardware time has stopped. When this is detected, it will do the following:

1. Advance the software time by a zero-delay delta cycle to update DUT outputs, execute the TBCall service functions, and send DUT inputs.
2. Then, advance the hardware time and software time again in parallel.

You should be careful when writing a multi-thread testbench. A multi-thread testbench might not function as expected in the NBrun mode if a thread is waiting for something to be returned from DUT while another thread has entered a critical section that is supposed to happen only after the first thread has finished.

Note: In NBrun mode, switching between the hardware and software is not allowed.

© 2021 All Rights Reserved.

NBrun versus TBrun

- The NBrun mode is not necessarily faster than the TBrun mode, however, it can reduce the number of synchronizations for the designs in which the software clock is running frequently and faster in real time than the hardware clock. For example, you might have a small software clock generating a sequence of data to be sent to the hardware after filling up a FIFO. There is no need to synchronize hardware at each small software clock edge.
- The NBrun mode is useful when the software has small delay events most of which are not dependent on the DUT. In the TBrun mode, the hardware stops at every software delay event for synchronization. In the NBrun mode, the hardware stops only when the hardware needs service or when the software has to send some data to the hardware.
- The NBrun mode allows emulation and simulation times to proceed concurrently, which means that the simulation times of DUT and testbench are independent and do not synchronize with each other.
- In both the TBrun and NBrun modes, the TBCall service cannot be executed in parallel with the hardware. For example, in the SCE-MI pipe applications, the pipe-message transfer is done in zero-delay delta simulation cycles while holding the hardware time. The hardware time cannot proceed while the software is filling or emptying the hardware pipe.

Running a Design in NBrun Mode

To run a design in this mode, use the following command:

```
xc on -nbrun <delay>
```

You can specify `<delay>` as a number with or without time units, such as `100ns` or just `100`. When you specify only a number, the minimum precision of the design is used.

Cadence recommends that you start from a minimum `<delay>` to get the design functionality correct. Then, gradually increase it to a maximum value to get the best performance in the NBrun mode. To determine the `<delay>`, consider the following guidelines:

- At the minimum, `<delay>` should be the smallest of the following:
  - The smallest delay in the testbench
  - The fastest clock delay in DUT because the DUT state will not change within the fastest clock delay
- At the maximum, `<delay>` should not be larger than the time between two consecutive testbench-to-DUT events.

© 2021 All Rights Reserved.

You can also specify `0` as the delay. The NBrun scheduler automatically schedules a delay to poll if the hardware controlled clock stops. This delay will be the smallest of the following:

- Next testbench timed event such as `#delay` in SystemVerilog
- Difference between the current testbench time and the DUT time returned from the previous hardware poll

At the end of an NBrun session, a message like the following is added to the run logs:

```
--- HW execs finished at tbTime=1 US dutTime=16056 NS #tbDelays=1000
```

- `dutTime:` Hardware time when NBrun finishes execution
- `tbTime:` Simulator time when NBrun finishes execution
- `#tbDelays`: Number of emulator polls caused by the testbench time delay events

`tbTime` and `#tbDelays` help you determine the average `#delay` in the testbench. If the average testbench `#delay` is shorter than expected, then review the `#delay` statements in the testbench, and increase them for better performance.

Using xeDebug to Run a Design

You can use the *NBrun* button from the *Setup* tab of the xeDebug GUI to run a design in the NBrun mode.

To specify `<delay>`, use the *nbrun delay* text field, which becomes visible on clicking the *--> Show Hot Swap Options* button from the *Configuration* group box (refer to Figure 38-3).

You can use the *Save* and *Restart* buttons to save and restart a session in the NBrun mode. When you restart a session, the time at the swap-in needs to be synchronized; thus, the time after the restart is the time in software. You can reset a session in the NBrun mode, which implicitly changes the run mode to software; thus, the time is `0` at reset. Similar to the TBrun mode, in the NBrun mode, you can use other features, such as SDL, dual-clock domain, DYNP mode, and FV mode. However, the hardware waveforms and the SDL time appear according to the time in hardware. For more information, refer to Using Setup Tab in SA Mode of Chapter 42, “Using the xeDebug GUI.”

Viewing Timestamps in NBrun Mode

In the NBrun mode, the timestamps in the software and hardware domains are not synchronized. Therefore, the timestamps displayed in different channels might be different.

The following channels display the software timestamps:

© 2021 All Rights Reserved.

- The time in the run-time log files, such as `xrun.log` or `xeDebug.log`
- The time in the command prompt window
- Waveforms for the software signals

The following channels display the hardware timestamps:

- *Current Time* in the *System Status* group box in the *Control* tab of the xeDebug GUI
- Waveforms for the hardware signals including controlled and uncontrolled clocks

Viewing Waveform Database Generated in NBrun Mode

In the NBrun mode, the software and hardware signals cannot be viewed in the same waveform viewer window. You should open separate waveform viewer windows to see the hardware signals and software signals. For example, if `trace.shm` is your database, you need to perform the following steps:

1. In the SimVision *Design Browser* window, open the `trace.shm` database.
2. Select the *Windows* menu of the *Design Browser*.
3. From displayed sub-menu, click *New* > *Waveform* to open a new waveform viewer window. Repeat this step to open the second waveform viewer window as well.
4. In both the waveform viewer windows, click the *Browse the design hierarchy* tab to display the *Design Browser* pane on the left.
5. In the first waveform viewer window, select *trace* instead of *All Available Data* from the *Browse* drop-down list. From the displayed list of software testbench signals, select the required signals for viewing.
6. In the second waveform viewer window, select *trace_hw* instead of *All Available Data* from the *Browse* drop-down list. From the displayed list of hardware DUT signals, select the required signals for viewing.

Note: When you trace the waveform on hardware in an NBrun session, the waveform shows the hardware timestamps. The hardware might be running faster or slower than the software. If the hardware time is later than the software time, and you swap the design out to software, and swap back to hardware, the waveform that was captured in the first NBrun session will be deleted. This is because the starting time in hardware at the time of the second swap-in needs to be synchronized with the time in software. But it is possible that the time in software is lesser than the ending time of the first NBrun session in hardware. Thus, an error similiar to the following is generated:

```
ERROR (legacy-44643): upload: start cycle 585124 is larger than end cycle 18000.
```


© 2021 All Rights Reserved.

Refer to Figure 38-4 that illustrates two separate waveform viewer windows.

Figure 38-4 Viewing Software and Hardware Signals in NBrun Mode

© 2021
All Rights Reserved.

Using Infinitrace in NBrun with GSFIFO and SFIFO Calls

NBrun supports Infinitrace with GSFIFO/SFIFO memory write operations. The usage model for NBrun with infiniTrace is same as that of TBrun with infiniTrace. However, in NBrun mode, the fclk is paused during memory writes, which might have performance impact.

Example: Using Infinitrace with GSFIFO/SFIFO Memory Writes

```
xc xt0 zt0
xc on -nbrun 0
run -swap
infiniTrace -prepare -overwrite prep1
....
run 100
memory -dump %h top.d1.s1.mem_exp -file mem1.hex
infiniTrace -close
infiniTrace -observe prep1
infiniTrace -goto begin
infiniTrace -goto end
memory -dump %h top.d1.s1.mem_exp -file mem5.hex
diff mem1.hex mem5.hex > mem1.hex.DIFF
infiniTrace -close
```

Note: Infinitrace is not supported with multi-channel GFIFO calls.

C-TestBench Run Mode (CTB Run Mode)

The IXCOM compiler supports testbench task/function calls from RTL and RTL task/function calls from testbench. Such testbench routines can be implemented in C or HDL, such as SystemVerilog or SystemC. To execute HDL testbench tasks in the hardware run mode, the control needs to switch between the hardware scheduler (that is running in C) and the xmsim space. Such context-switching between C and xmsim might cause costly DPI context-switching overheads.

CTB run mode supports C-only testbench tasks/functions to and from hardware. In this mode, the C hardware scheduler to xmsim context-switching is completely avoided until the end of the run. All transactions between C-only testbench tasks/functions and hardware are done in the C space and the control does not switch back to HDL.

This run mode is categorized into the CTB TBrun and CTB NBrun modes. These modes are similar to the TBrun Mode and NBrun Mode.

CTB TBrun Mode

The CTB TBrun mode is the same as TBrun mode except that it only supports C tasks/functions to/from hardware. Like the regular TBrun mode, concurrent GFIFO functions are executed in parallel while hardware clock is running. Also like the regular TBrun mode, if a

© 2021 All Rights Reserved.

new event is scheduled by the concurrent GFIFO calls, that event is delayed and executed only after the hardware clock is stopped (TBsync).

Following is the command to enable CTB TBrun mode:

```
xc on -ctb -tbrun
```

CTB NBrun Mode

The CTB NBrun mode is the same as the CTB TBrun mode except that the events that are scheduled by concurrent GFIFO calls, are executed immediately while the hardware clock is still running. Following is the command to enable CTB Nbrun mode:

```
xc on -ctb -nbrun 0
```

Advancing Clocks in CTB Run Mode (xrun -R only)

In the CTB run mode, while running simulation, you can advance clocks using `xrun -R`. Use the xmsim command `xrun` <*t*> to specify a relative or absolute target timestamp to which you want the clocks to advance.

```
xcelium> xrun [-rel | -abs] <t>
```

If you specify the xmsim native `run` <*t*> command, clocks advance forever until interrupt or `$finish`. The `xrun` command is also available in the Autorun mode with the same syntax.

If, you are using xeDebug, use the `run` <*t*> command to advance clocks to a specified time.

Suspending import C Task for a Delay

In an `import C` task, you can call this C function to suspend the calling task for a specified delay in nanoseconds. The scheduler suspends the task and resumes the task after the specified delay.

```
marg_wait_delay_ns(double d);
```

Limitations of the CTB Run Mode

- The CTB run mode does not switch to xmsim during the entire run. So, the actions that rely on the Verilog simulator are not executed. Simulation threads that are suspended are blocked forever. PI/PO propagation is not supported between hardware and SystemVerilog testbench. You can use `marg` functions from C to access hardware.
- `#delay` in Verilog is ignored. When the CTB run mode is entered, suspended `#delays` in Verilog is blocked forever. Future `#delays` are not executed.

© 2021 All Rights Reserved.

You can use `marg_wait_delay_ns()` to model `#delay` in an `import C` task.

- HDL testbench is not supported. An HDL task/function that is mapped to software is ignored in CTB run mode. This includes `sv_task` and `tb_import` mapping for HDL tasks or functions, such as UVM. HDL task/function that is mapped to hardware is supported.
- SystemC is not supported.
- Swapout is not supported.
- Active (suspended) `import C` tasks are blocked. When the CTB run mode is entered, `import C` tasks that are still suspended (active) are blocked forever. It is recommended that the CTB run mode be entered at time `0` to reduce the chance of active import tasks.
- Active initial `import C` tasks are blocked. Active `import C` task call from an initial block can be suspended even at time `0`. As an alternative, use the `-mapInitClk` option of the `ixcom` command to map initial blocks to hardware.
- Save-restart is not supported. Only the `-process_save` option of xmsim is supported.

Autorun Mode

The Autorun mode uses the most restricted execution model with the following features:

- Designs compiled with clock generator in DUT (`+iscDelay`) can only use `xc on -autorun` mode.
- Only the `tbcall` (`tbcall_region`) and import DPI calls from DUT are processed in the `xc on -autorun` mode.
- Only non-time consuming `tbcall` and import DPI calls are allowed in the `xc on -autorun` mode.
- All pending testbench time delay events scheduled before entering the `autorun` mode are executed after exiting the `autorun` mode.
- In most cases, the `tbrun` mode has the similar performance as the `autorun` mode but has fewer restrictions than the `autorun` mode, and is therefore recommended for use.
- Only in special cases when you want to ignore all testbench time delay events, such as testbench local generated clocks, `autorun` mode has its special applications for better performance.
- The testbench input events are ignored to get faster speed.

© 2021 All Rights Reserved.

- An error is generated if the design contains any input ports referencing to DUT. Specify `xc on -piok -autorun` command to ignore the input port error. This ensures that autorun ignores the DUT input (PI) changes completely.

When using xeDebug tool, specify `run` command to advance hardware time in autorun mode. Specifying `run` command without a time argument runs the hardware indefinitely. `run 100ns` command advances hardware time by `100ns` and then stops.

When using *xrun* tool in autorun mode, specify `xrun` command to advance hardware time. Specifying `xrun` command without a time argument runs the hardware indefinitely. `xrun 100ns` command advances hardware time by `100ns` and then stops.

After swapping back to simulator from the autorun mode, only the DUT states in Xcelium are uploaded by the emulator. The testbench states remain the same as at the time of swap-in. To get best performance in autorun mode use a design which has no testbench and only uses tbcall with no delays in the tbcall tasks.

For example, if you have a testbench clock rising at every `10ns` and the autorun starts at `100ns` and ends at `414ns`; since a rising edge event is scheduled to execute at `+10ns` when autorun starts the pending rising edge event takes place at `424ns` which is `10ns` after autorun stops at `414ns`.

Running a Design in Autorun Mode

To run a design in autorun mode, perform the following steps:

1. Compile the entire design as the DUT, leaving no module out.
2. Use `#delay` statement to model clock generation and use ixcom switch `+iscDelay` to compile the clock generator.
3. Use the `axis_tbcall` primitive/procedure for file I/O operations (`$readmemb`, `$readmemh`, `$display`) or system control functions (`xc stop` and `xc off`).

Stopping a Design Running in Autorun Mode

Stop a design running in autorun mode by using the following commands:

1. Use `Ctrl-C` from the CLI to interrupt the operation.
2. Add `$stop` and `$ixc_ctrl` ("`tb_import`", "`$stop`") in a DUT module. The simulator stops when hardware executes the `$stop` statement.

Note: To ignore all PI events during autorun, use the `xc on -piok -autorun` command.

© 2021 All Rights Reserved.

Match Mode

When an unexpected behavior is noticed while running in the emulator, run in match mode to debug the issue.

In the Match mode both simulation and emulation run concurrently and the DUT values from the emulator model are checked at each time step to make sure that they match with the simulator DUT model.

A typical reason for mismatch can be a race condition in your RTL code, and the solution can be to rewrite the source code to avoid this race condition. If there is a mismatch while running in match mode, an error message is generated and printed out and the run is stopped.

For example:

```
sample error message:Error: PO top.mem_address(0) mismatch hw = 1, sw = 0 at 230
NS Memory Usage - 20.2M program + 296.0M data = 316.2M total CPU Usage - 0.1s system
+ 0.3s user = 0.4s total (0.1% cpu) Simulation stopped via $stop(2) at time 230 NS
+ 7
```


© 2021 All Rights Reserved.

Accessing Signal Values

You can force, release, deposit values to the design objects.

- Forcing and Releasing Nets:

  You can force and release values of the nets in the design. The value can be entered using Verilog notation. You can also save current list of forced nets and restore forced nets from a file. There is no need to specify whether the net is located in the testbench or DUT.

- Depositing Values to Design Objects:

  When running in SA mode, the values can be deposited to testbench objects with the following options:

  - With respect to absolute/relative time
  - Repeat/cancel time period
  - Release force prior to deposit
  - With inertial, transport or no delay

  Values can be deposited to DUT objects with optionally releasing force prior to depositing values. You can deposit values to the nets and memories in the design.

This section provides information on forcing, releasing, and depositing values to design objects. For details of the commands mentioned in this section, refer to the *System Tasks and Procedures section* in the *Compile-time Commands in SA Mode chapter* in the *VXE Command Reference Manual*. The steps and concepts for these are described in System Task Mapping in IXCOM flow on page 762.

Forcing Values

You can set a given value for specific design objects, and force them to retain it until explicitly released. For information on releasing the forced values from objects, refer to Releasing Values.

The design objects to which values can be forced include signals, symbols, and buses. A signal can be forced only if it is specified as a `keepNet` at the compile time.

The method for forcing values varies based on the run-time tool used, that is, xeDebug or *xrun*. The following sub-sections discuss this in detail.

© 2021 All Rights Reserved.

Note: If you force a partial vector mapped to the hardware in the Xcelium simulator, the whole vector is forced. For example, if the current value of a vector is `'b1101` and the `bit[0]` is forced to `0`, then the vector is forced with `'b1100`.

Forcing Values Using the xeDebug Tool

In both ICE and IXCOM flows, execute the following XEL command to force a value:

```
force <object_name> <value>
```

When a signal represents a single bit, *`<value>`* is either `0` or `1`. When forcing a value to a vector that represents several bits, the *`<value>`* can be specified as a binary, octal, decimal, or hexadecimal radix, prefixed by the appropriate letter following Verilog syntax. For example, `'b011`, `'o1237`, `'d089`, `'hffcd`. A number without a radix prefix is assumed to be decimal, for example `89` is the same as `'d89`.

Refer to "Force Tab" on page 1954 for details about using the xeDebug GUI for forcing signals.

You can also force multiple signals simultaneously instead of forcing signals one by one. Specify various signals and values in a text file and execute the following command:

```
force -file -prep <textfile>
```

The above command generates a binary (`.dat`) file containing an array of nets and data. Specify the generated binary file in the following command to force multiple signals at the same time:

```
force -file <textfile.dat>
```

For information on the text file format and restrictions, refer to the description of the `force` command in the *VXE Command Reference Manual*.

Forcing a keepnet is safe while forcing a non-keepnet might have unexpected side effects. For example, a net could be optimized so that other unrelated logic might depend on this net. When this net is forced, other unrelated logic might also be affected. Therefore, forcing non-keepnets is not recommended. The software might issue a warning when a non-keepnet is forced. Use following `xeset` command to decide whether a warning should be issued when forcing a non-keepnet.

```
xeset allowExtraForcing [silent | warn]
```

For more information, refer to the description of the `xeset allowExtraForcing` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

© 2021 All Rights Reserved.

Forcing Values Using *xrun*

When using *xrun* in SA mode, executing the `force` XEL command does not access the hardware correctly. Therefore, follow either of the steps given below before you proceed:

- Modify the HDL design to include the `$export_frcrel` system task on the target signals and then re-compile the design.

  OR

- At run time, to dynamically export the target signals to force the values, execute the following command:

  ```
  xc export_frcrel <signal>
  ```

  Note: The `xc export_frcrel` command might not work for certain DUT signals that have been optimized by the compiler.

These steps create a special force logic for the specified signals in the hardware and cause these signals to be exported from the emulator, which is required if you want to force and release the signal in SA mode. After executing either of the above-mentioned steps, use the `force <signal> <value>` command to force values to the DUT signals. Otherwise, the `force` command will work only for the testbench objects.

Releasing Values

A signal retains a forced value until it is explicitly released. The method for releasing a previous force varies based on the run-time tool used, that is, xeDebug or *xrun*. The following sub-sections discuss this in detail.

Releasing Values Using the xeDebug Tool

To release any previous force (executed using the `force` command) on the specified signal or symbol, execute the following command:

```
release {<signal> ... | <symbol> ...}
```

Consequently, the value of the specified signal is recalculated at the next cycle. However, the sourceless nets declared as `keepNets` will retain their values (state).

Note: If the `release` command is operated on a bus in LA mode while the clocks are running, time coherency between the values of the signals in the bus is not guaranteed.

When using the xeDebug GUI, the *Release* button on the *Force* tab enables you to release individual signals or nets (See Figure 42-54 on page 1955).

© 2021 All Rights Reserved.

Releasing Values Using *xrun*

Similar to forcing values, follow either of the two steps discussed in Forcing Values Using *xrun* to correctly access the hardware for releasing values.

Note: The `xc export_frcrel` command might not work for certain DUT signals that have been optimized by the compiler.

Depositing Values

You can set the state of a signal, symbol, or a bus to the specified value. For a net type object (not a memory) that exists in the emulator, the deposit operation can be done reliably only if the object had been specified as a `keepNet` during compile. The deposit operation will force the value of the object to the specified value, and then release it. The number of cycles through which the value remains forced on the object depends on the operation mode.

The method for depositing values varies based on the run-time tool used, that is, xeDebug or *xrun*. The following sub-sections discuss this in detail. Refer to Deposit Tab on page 1957 for using xeDebug GUI for depositing values.

Note: The `value` and `deposit` commands do not access memories if MDA is not bit-blasted. Use the `memory -value` and `deposit -value` commands instead. The `memory -value` and `deposit -value` commands are supported in both SA and ICE modes.

Depositing Values Using the xeDebug Tool

In both ICE and SA modes, execute the following XEL command to deposit a value:

```
deposit <object_name> <value>
```

When a signal represents a single bit, `<value>` is either `0` or `1`. When forcing a value to a vector that represents several bits, the `<value>` can be specified as a binary, octal, decimal, or hexadecimal radix, prefixed by the appropriate letter following Verilog syntax. For example, `'b011`, `'o1237`, `'d089`, `'hffcd`. A number without a radix prefix is assumed to be decimal, for example `89` is the same as `'d89`.

In SA mode, to release an active force and allow to deposit a specified value using the `deposit` command, execute the following command:

```
deposit <object_name> <value> -release
```

In ICE mode (STB, LA, and VD modes), the `-release` option is superfluous because a previous force is always released before applying the deposit.

© 2021 All Rights Reserved.

For testbench objects in SA mode, following additional options of the `deposit` command can be used:

```
deposit <object_name> [=] <value>
        [-after <time_spec> [value -after <time_spec> ...]]
        [{-relative | -absolute}]
        [-generic]
        [-inertial]
        [-release]
        [-repeat period [-cancel period]]
        [-cancel 0]
        [-transport]
```

For syntax details, refer to the description of the `deposit` command in the *Xcelium Simulator Tcl Command Reference* in the Xcelium documentation set.

You can also set the values of multiple signals simultaneously. Specify various signals and values in a text file and execute the following command:

```
force -file -prep <textfile>
```

The above command generates a binary (`.dat`) file containing an array of nets and data. Specify the generated binary file in the following command to set the values of multiple signals at the same time:

```
deposit -file <textfile.dat>
```

For information on the text file format and restrictions, refer to the description of the `force` command in the *VXE Command Reference Manual*.

Setting values for keepnets does not affect the other nets. If you need to set the same value to the non-keepnets as well, execute the following `xeset` command at run time before the `deposit` command:

Depositing values to keepnets is safe while depositing values to non-keepnets might have unexpected side effects. For example, a net could be optimized so that other unrelated logic might depend on this net. When deposit operation is performed on this net, other unrelated logic might also be affected. Therefore, depositing values to non-keepnets is not recommended. The software might issue a warning when deposit operation is performed on a non-keepnet. Use the following `xeset` command to decide whether a warning should be issued when depositing a non-keepnet.

```
xeset allowExtraForcing [silent | warn]
```

For more information, refer to the description of the `xeset allowExtraForcing` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

© 2021 All Rights Reserved.

Depositing Values using *xrun*

When using *xrun* in SA mode, execute the following command:

```
xc deposit <signal> <value>
```

This command deposits a value to a DUT signal. It does not work for a testbench signal. The DUT signal does not have to be pre-compiled with `$export_deposit`. If the DUT signal is not pre-compiled with `$export_deposit` and it is optimized by the compiler, the `xc deposit` command will report an error.

Initializing Registers Randomly in SA Mode

The following xc commands enable users to save the list of registers with `X` or `Z` values, and then, randomly initialize these registers in SA mode:

```
xc xreg -save
xc xreg -rand <seed> -scope <file_scope>
xc xreg [-all0 | -all1] -scope <file_scope>
xc xreg -verbose
```

Using these commands to randomly initialize registers can be useful for regression tests.

The usage of these commands is described in the following sections.

Saving X Registers

The recommended usage model for random initialization of registers is to first save a list of registers with `X` or `Z` values by using the following command in software mode:

```
xc xreg -save
```

This command can be used only in the software mode. If you issue this command in the hardware mode, an error is generated. By default, the designs run in the software mode at time `0`. There will be very few `X` registers after swap-out. Saving `X` registers after swap-out is not useful.

If the `xc xreg -save` command is issued at time `0`, the list of registers is saved at the end of time `0`, because many variables in the `initial` blocks are initialized at time `0`. Note that these registers cannot be randomized.

If the `xc xreg -save` command is issued at a non-zero time, the list of registers is saved immediately.

With each subsequent `xc xreg -save` command, the previous list of registers with `X` or `Z` values is overwritten.

© 2021 All Rights Reserved.

The list of registers is saved in the CPU memory. It cannot be carried over to the next run-time session.

You can execute the `xc verbose` command before `xc xreg -save` to view the registers that are saved by the `-save` option.

Specifying Scope

If you specify `-rand`, `-all0`, or `-all1`, you must also specify the module instance scope of registers to be changed by using the following switch:

```
xc xreg -scope <file_scope>
```

You can also specify multiple instances in a file and provide the file to the `-scope` option.

You can use the following command to set all registers in <*file_scope*> to `0` or `1`:

```
xc xreg [-all0 | -all1]
```

For example, `-scope top` specifies that registers under the `top` hierarchy should be randomized. You can also specify `-scope scopeFile` where `scopeFile` is a text file containing lines in the following format:

```
include top.dut
exclude top.dut.clock_gen
exclude top.dut.reset_module
```

With the above file, all the registers in the `top.dut` hierarchy except the ones under `top.dut.clock_gen` and `top.dut.reset_module` are randomized.

Randomizing Register Values

After saving the list of registers with `X` or `Z` values by using the `xc xreg -save` command, you can randomize that list of registers by using the following command:

```
xc xreg -rand <seed>
```

You must also specify `-scope <file_scope>` when you do `xc -rand <seed>`. The random <*seed*> value can be any integer number. If the same seed is used from one run-time session to the next one, the randomization sequence remains the same. For example, if in the first *xrun* session, the randomization sequence was `5,19,3,7`, in the next *xrun* session, you will still get `5,19,3,7`.

You must issue this command only in the hardware mode. If you issue this command in the software mode an error message is generated.

The `xc xreg -rand` command does not require a prior save. All registers are randomized if the `xc xreg -save` command was not used before swap-in. However, randomizing all

© 2021 All Rights Reserved.

registers might increase the probability of changing a critical part of the design, such as reset circuitry getting changed. So, you must exclude the critical parts of the design in the file specified with the `-scope` option.

Following is an example to illustrate the usage of the register-randomization commands:

```
xc verbose        // Enable display of registers that will be saved by xc xreg
                    -save command.
xc xreg -save     // Save the list of registers with X or Z values
xc on -tbrun      // Switch to the hardware tbrun mode
run 10us          // Run the design for 10 us
xc xreg -rand 1 -scope tb.dut  // Randomly set the saved list of registers
                                  under the tb.dut hierarchy
run 10us          // Run the design for another 10 us
exit              // Exit the run-time session
```

Debugging Register Randomization

To see the registers that are saved by the `xc xreg -save` command, issue the `xc verbose` command before the `xc xreg` command. The following examples illustrate how you can debug randomly initialized registers:

Example 1:

```
xmsim> xc verbose
xmsim> xc xreg -save
          xreg dut.w1 sw = x
          xreg dut.w2 sw = x
xmsim> xc on -tbrun
xmsim> run 10us
xmsim> xc xreg -rand 1 -scope dut -verbose
          xreg dut.w1 hw = 1
          xreg dut.w1 hw = 1
```

Using the Palladium Xref Browser of SimVision GUI in SA Mode

When you compile the designs for SA mode, the IXCOM compiler generates an instrumented snapshot for the Xcelium simulator. At the time of debugging, this snapshot is used, and it appears to be quite different from the original RTL design sources in the Source browser windows of the SimVision GUI. So, you might experience difficulties while watching active annotation during simulation or setting breakpoints.

To resolve these issues, you can use a side-by-side browsing tool, called the *Palladium Xref Browser*, which consists of essentially two source browsers that cross-link the instrumented snapshot back to the corresponding lines in the original design files.

© 2021 All Rights Reserved.

To enable access to this browser, you need to include the `+xref_browser` option of the `ixcom` command at the time of compiling the design. When you include this option, the IXCOM compiler ensures that the mapping information from the original source browser to the instrumented sources is automatically generated. When you invoke the *Palladium Xref Browser*, it searches for the mapping file and returns an error if the file is not found.

This browser is available whether you invoke xeDebug independently by using the `xeDebug -gui` UNIX command, or by using the `xrun -xeDebug -gui` command. It is provided in the SimVision GUI under the *New* menu option. It supports both online and offline sessions. This browser is available **only in SA mode** from the SimVision GUI, and allows only object-level and file-level cross referencing.

You can open only one instance of the *Palladium Xref Browser* at a time. When you invoke the *Palladium Xref Browser*, two source browsers open side by side, and display the current simulation point. These source browsers are:

- The source browser in which your original sources are displayed. It is called the *User Source Browser*.
- The instrumented source browser in which the instrumented sources are displayed. This is the normal source browser window, which is titled *Source Browser #*. The `#` suffix in the source browser window title indicates the number of source browser windows that are currently open, for example *Source Browser 1*.

Figure 38-5 illustrates the two source browser windows of the *Palladium XRef Browser*. The instrumented source browser is displayed on the left and the *User Source Browser* on the right. In the following figure, the corresponding assertion is highlighted in both browsers.

© 2021 All Rights Reserved.

Figure 38-5 Palladium Xref Browser

Using the Source Browsers of the Palladium Xref Browser

Both windows in the *Palladium Xref Browser* are standard SimVision source browser windows and, therefore, have the standard menus: *File*, *Edit*, *View*, *Select* (disabled in the *User Source Browser*), *Format*, *Simulation*, *Windows*, and *Help*. The user source browser has neither the *Palladium action* toolbar, nor the *Scope selector*.

The *Palladium Xref Browser* automatically starts a processing loop to detect the changes in the instrumented source files, as these occur, either as a result of simulation advancing, or because of user interaction, such as changing the scope. If the instrumented source browser is blank, the *User Source Browser* is blank. You can initialize the instrumented browser by selecting an object in the design browser.

If the *User Source Browser* is closed, the *Palladium Xref Browser* feature terminates, but the instrumented source browser remains open. If the instrumented source browser is closed, the *Palladium Xref Browser* feature terminates, but the *User Source Browser* remains open. To re-enable the *Palladium Xref Browser*, re-invoke it from the SimVision pull-down menu; this will re-open its closed browsers.

© 2021 All Rights Reserved.

You can manually open a different file using the menus, or change the scope by using the scope dropdown box. Changes in the instrumented source browser scope or file affect the *User Source Browser*. But if no mapping information available, there is no change in the *User Source Browser*.

In the *Palladium Xref Browser*, the objects, for example signals, can be highlighted in either of the browsers. Simulatenously, the corresponding objects, are highlighted in the other browser. However, you can perform object-manipulation operations, like probing, forcing, or value annotation only in the instrumented source browser. Selecting objects in the other SimVision browsers, even another Design Browser has no effect on the *Palladium Xref Browser*.

Offline Debugging

This feature is available when you start an offline session of xeDebug, that is, without the simulator. During offline debugging, you do not have access to simulator-related features, such as breakpoints and pointer advancing. However, active annotation from the waveform in the instrumented source browser is available.

Assertions

Assertion failures also display the relevant assertions in the original source code. You can invoke the SimVision Assertion browser to display the failed assertion, and select the assertion to be sent to the SimVision source browser. Then, you can invoke the *Palladium Xref Browser* to display the original sources.

Limitations

The *Palladium Xref Browser* does not support the following functionality:

- Mapping original user design objects to the equivalent instrumented design objects to be used for probing and other standard functions is not supported. In the source browser, you can highlight strings in the text window, but you need to rely on mapping data, including column information, to find the corresponding instrumented object. Automated probing of user design objects is not supported.
- The IXCOM compiler does not provide line-to-line mapping information to allow correspondence points within the file other than objects. So, breakpoints in user sources and live position updates are not possible.
- Certain standard SimVision operations work orthogonally to this feature. So, the reset, restart, and save operations do not disable the *Palladium Xref Browser*.

© 2021 All Rights Reserved.

Files Generated during Debugging

Here are the log and message files that the xeDebug tool generates during the debug process.

xeDebug.log

When a debugging session is started, xeDebug makes a main log named `xeDebug.log` in the design directory. This log file contains an exact copy of your terminal or console window. This file is incremental, that is, multiple debug sessions get sequentially recorded within the `xeDebug.log` file. You can control the main log using the `logfile -set` <*logfilename*> command in xeDebug or vvmDebug.

Controlling the main log is typically helpful when you need to split the log according to various testcases or when a design is run for many days. In such cases, the `xeDebug.log` file can be several GB. Larger log files are slow to open and process.

For information about the various options of the `logfile` command, refer to the description of the `logfile` command in the *VXE Command Reference Manual*.

xe.msg

A file named `xe.msg` is also generated which supports recursive logging, that is, nested command calls are also logged in the file. The `xe.msg` file contains a complete log of all previous compile and run-time sessions. This file includes the output displayed on the screen (same as `xeDebug.log`), as well as additional messages that otherwise are not recorded in the console log file.

The `xe.msg` file is a superset of `xeDebug.log`. It is also incremental like the `xeDebug.log` file. Therefore, many commands are logged twice; first as an echo to the screen and second from inside the commands. Echoes are shown with a prompt in upper case, like on the screen, and logging from inside commands are shown with the prompt in lower case.

xeDebug.key

The xeDebug tool generates another log file named `xeDebug.key` that contains all the commands that you have typed or GUI has generated. It is a record of the debug session, and if sourced, produces exactly the same results as the original session with a few exceptions.

- The first exception is for time-sensitive features. For example, consider executing the `run -nowait` command followed by delay of one second, and then execute the `stop`

© 2021 All Rights Reserved.

command. The result in this case depends on the performance of the emulator, host, and host load.

- The second exception is `Ctrl+c` that you have entered. If you start a blocking run and then press `Ctrl+c`, it cannot be recorded correctly.

There could be other reasons for the replayed session to produce different results. For example, if during the debug session you modify contents of files used by the debugger, such as memory initialization files or SDL files, then the replayed session will only use the latest version of the files, which might not be the same as the contents used in the original debug session.

Debugging Designs with Modular Compilation

Modular compilation distributes compilation to multiple workstations. You need to partition your design in smaller units, referred to as *buckets*. The buckets are the module names defined by you in the *bucketModules.lst* file. For detailed information on how to use modular compilation in ICE and SA modes, refer to Modular Compilation in ICE Mode and Executing Non-Synthesizable Code in the Emulator sections respectively.

After modular compilation, you have the option to debug your design according to the defined buckets. This section explains the debugging methodology and the MC-specific changes needed in the input for xeDebug and behavior of various run-time commands. This section comprises the following topics:

- Specifying Buckets to Debug
- Controlling Side File Generation for Bit-Blasted Memories
- Debug Commands Support for Modular-Compiled Designs

Specifying Buckets to Debug

You can specify the buckets at run time by the following alternative ways:

- Specifying Buckets with debug Command
- Specifying Buckets with XE_DEBUG_ARGS Environment Variable
- Specifying Buckets with xeDebug Command

To debug your design without opening any bucket, use the `debug -bucket none` command.

© 2021 All Rights Reserved.

Specifying Buckets with debug Command

At the beginning of a debug session, you can either specify a bucket for debugging or debug without any bucket. You cannot change the specified bucket in the same debug session. This behavior applies to both online and offline debugging.

Note: No bucket is open by default. You can open maximum one bucket at run time in either online or offline mode.

Use the `-bucket` option of the `debug` command to specify a bucket to debug. For example:

```
debug . [-bucket <bucket_module_instance> | <bucket_number>] \
        [-session <session_name>] ...
```

You can either specify the module instance name or the bucket number to debug. The `session` option is optional. Bucket `0` corresponds to the top module.

For example:

```
XE> debug . -bucket {test.\genblk_0_.u_add }
```

Note: Bucket `0` refers to the design top and also contains the instances not explicitly assigned to other buckets. Buckets 1, 2, and so on correspond to the instances of modules listed in the `bucketModules.lst` file.

You can view the bucket names in the design directory using the `debug -listbuckets` command.

For example:

```
XE> debug -listbuckets
{} {test.\genblk_1_.u_add } {test.\genblk_0_.u_add } test.u_sub_1_ test.u_sub_0_
test.u_mul_1_ test.u_mul_0_
```

Specifying Buckets with XE_DEBUG_ARGS Environment Variable

VXE provides the `XE_DEBUG_ARGS` environment variable that replaces all arguments specified with the `debug` command. When a design compiled using modular compilation is run using the `xrun` tool (that is, with the `xrun -R -xeDebug` command), the `debug` command is executed implicitly. In such cases, the arguments specified with the `XE_DEBUG_ARGS` variable are used as buckets.

Example:

```
setenv XE_DEBUG_ARGS=". -bucket 1"
```


© 2021 All Rights Reserved.

Specifying Buckets with xeDebug Command

Use the `-input` argument of the `xeDebug` command to specify a bucket, as follows:

```
xeDebug -input "@debug . -bucket <module_instance_name> | <bucket_number>"
[other_xeDebug_arguments]
```

For example:

```
xeDebug -input "@debug . -bucket 0 ; host . ; xc run xt0 zt0; run 10ns; -exit"
```

Controlling Side File Generation for Bit-Blasted Memories

Information about critical nets and all the memories is saved at compile time. In addition to critical nets, all user-defined keepnets are also saved. These keepnets are referred to as *preserved nets*. The name-translation support for the preserved nets is limited. By default, the information about bit-blasted memories is saved in a separate side file generated in the `/dbfiles` directory. Generation of this side file might be a time-consuming process. You can disable the side file generation using the `compilerOption -add -EnableExtraFileForRegressionMode OFF` command in your compile script. In this case, bit-blasted memories will not supported at run time.

By default, full memory information is not saved for better performance. To include full memory information in the side file, use the `compilerOption -add dumpFFMemoryFullInfo ON` command.

The debug session provides the debug functionality for all objects from the current bucket and all the preserved nets. All memories are available for the whole design.

The *Design Browser* in xeDebug shows only the skeleton hierarchy and information for the current bucket. The browser functions such as tracing and search are supported for the current bucket.

Debug Commands Support for Modular-Compiled Designs

The following table lists the supported information and the restricted behavior of various run-time commands when the `debug . -bucket` or `debug . -bucket none` command is used:

© 2021 All Rights Reserved.

Table 38-1 Debug Commands Supported for Modular-Compiled Designs

Command	debug -bucket	debug -bucket none
assertion	Supported through xmsim only. Changing state of assertion by name is possible only for the current bucket or assertions saved as preserved nets. The `assertion` command accepts only the full valid RTL names. The `assertion` command is supported for the whole design. However, controlling assertions by name is supported only for the current bucket or for the assertions saved in preserved nets.	Supported through xmsim only
bm	Supported	Supported
clockConfig	Supported only in ICE mode. Not supported in SA mode.	Supported only in ICE mode. Not supported in SA mode.
clockConfig -stopMode	Supported	Supported
configPM	Supported	Supported
convertRadix	Supported	Supported

© 2021 All Rights Reserved.

Command	debug -bucket	debug -bucket none
database -open	The generated *.phy file supports the entire design and SHM or FSDB file is generated for the current bucket.	Supported
database -open <*trace_name*> -cont -phy	The generated *.phy file supports the whole design.	The generated *.phy file supports the whole design.
database -open <*trace_name*> -cont	The generated *.phy file supports the whole design and SHM/FSDB file is generated for the current bucket.	The generated *.phy file supports the whole design. The SHM/FSDB file is not generated.
database upload	Supports only the current bucket	Not Supported
database -captureMode	Supported	Supported
database -prepareOffline	The generated *.phy file supports the whole design.	The generated *.phy file supports the whole design. However, the phy file generation with the database -prepareOffline command is not supported in the DYNP mode.
database -tracewindow -start <*time*> -end <*time*>	Supported	The database -tracewindow command is ignored because the SHM/FSDB file is not generated.
database -partialfv -add <*instance_name*> [-exclude]	Supports only the current bucket	Not Supported
database -partialfv -clear	Supports only the current bucket	Not Supported

© 2021 All Rights Reserved.

Command	debug -bucket	debug -bucket none
`database -partialfv -list`	Supports only the current bucket	Not Supported
`database -partialfv -rm` <*inst1_name*> <*inst2_name*>... <*instN_name*>	Supports only the current bucket	Not Supported
`database -stream -close`	Supports only the current bucket	Supported only for preserved nets
`database -stream [-open]` <*streaming_db_name*>	Supports only the current bucket	Supported only for preserved nets
`database -stream -disable | -enable`	Supports only the current bucket	Supported only for preserved nets
`database -stream -exitWait [1 | 0]`	Supports only the current bucket	Supported only for preserved nets
`database -stream -overflow [ignore | warning | error]`	Supports only the current bucket	Supported only for preserved nets
`database -stream -show`	Supports only the current bucket	Supported only for preserved nets
`database -stream -startStream`	Supports only the current bucket	Supported only for preserved nets
`database -stream -stopStream`	Supports only the current bucket	Supported only for preserved nets
`database -stream -upload`	Supports only the current bucket	Supported only for preserved nets
`database -stream -waitWhileBusy`	Supports only the current bucket	Supported only for preserved nets
dbd	Supported for the current bucket only.	Not Supported
debug	Supports either one or no buckets at all.	Supported

© 2021 All Rights Reserved.

Command	debug -bucket	debug -bucket none
deposit	Supported only for the current bucket and keepnets. All the names from the current bucket are completely supported. The preserved names are supported in debug functionality, however, only full valid names are completely supported. The style (RTL or gate) of short names in the current scope must match the style of the current scope.	Supported only for keepnets.
deposit -allff	Not Supported	Not Supported
download	Supported	Supported
dpa	Not Supported	Not Supported
drtl	Supported only for the current bucket and keepnets.	Supported only for keepnets.

© 2021 All Rights Reserved.

Command	debug -bucket	debug -bucket none
force	Supported only for the current bucket and keepnets. All the names from the current bucket are completely supported. The preserved names are supported in debug functionality, however, only full valid names are completely supported. The style (RTL or gate) of short names in the current scope must match the style of the current scope.	Supported only for keepnets.
getCableStatus	Supported	Supported
getCycleNum	Supported	Supported
getHostStatus	Supported	Supported
getPMStatus	Supported	Supported
getTime	Supported	Supported
hdlConfig	Not Supported	Not Supported
host	Supported	Supported
host -dynp	Supported; probes are limited to the current bucket and keepnets	Not Supported

© 2021 All Rights Reserved.

Command	debug -bucket	debug -bucket none
infinitrace	Supported	Supported
	Forceable nets must be keepnets.	Forceable nets must be keepnets.
	It is not mandatory that the Observe session must have the same bucket open as the Prepare session.	It is not mandatory that the Observe session must have the same bucket open as the Prepare session.
lp	Not Supported	Not Supported
memory	Supported	Supported
	The memories in the whole design are completely supported. Only full valid RTL or gate names are accepted.	The memories in the whole design are completely supported. Only full valid RTL or gate names are accepted.
mergeSAIF	Supported	Supported
power	Not Supported	Not Supported
probe (in FV)	Supported; Nets within the buckets can be probed. Keepnets are not supported.	Not Supported
probe (in partial FV)	Not Supported	Not Supported
probe (in DYNP)	Supported; Nets within the buckets can be probed. Keepnets are not supported.	Not Supported

© 2021 All Rights Reserved.

Command	debug -bucket	debug -bucket none
`probe -stream`	Supported; All nets within the bucket and the preserved nets in addition to user-defined keepnets, which may or may not be in the current bucket.	Only user-defined keepnets can be used as SDL input.
`release`	Supported only for the current bucket and keepnets. All the names from the current bucket are completely supported. The preserved names are supported in debug functionality, however, only full valid names are completely supported. The style (RTL or gate) of short names in the current scope must match the style of the current scope.	Supported only for keepnets.
`reset`	Supported	Supported
`restart`	Supported	Supported
`resume`	Supported	Supported
`run`	Supported	Supported
`stop`	Supported	Supported
`save`	Supported	Supported

© 2021 All Rights Reserved.

Command	debug -bucket	debug -bucket none
scope	Supported only for the current bucket. Scopes outside the current bucket cannot be always be validated, especially in the ICE mode.	Not Supported
sdl	Supported; All nets within the bucket are supported by SDL, in addition to the user-defined keepnets which may or may not be in the current bucket.	Supported with keepnets
suspend	Supported	Supported
symbol	Supports only the current bucket and keepnets.	Not Supported
targetAssign	Supported	Supported
targetLocation	Supported	Supported
tca	Not Supported	Not Supported
userData	Supported	Supported
value	Supported only for the current bucket and keepnets.	Supported only for keepnets.
waitWhileBusy	Supported	Supported
xeset	Supported	Supported
xegui	Supported	Supported

For detailed information on modular compilation in ICE mode and the limitations, refer to the Modular Compilation in ICE Mode section. For information on how to use modular compilation in SA mode, refer to the Executing Non-Synthesizable Code in the Emulator section.

© 2021 All Rights Reserved.

Improving Debug Turnaround Time with Debugging in Regression Mode

xeDebug might sometimes consume long initialization time. Opening the Palladium database impacts time and memory. In certain cases, only a simple validation of the design is required, and only if the validation fails, a full-scale debugging is used. To save the initialization time in such cases, you can skip loading the Palladium database and instead load a smaller version of the database while opening a debug session in xeDebug. This mode is referred to as the *regression* mode.

The regression mode can be enabled using the following ways:

- Using the `-regression` option of the `debug` command:

  ```
  debug . -regression
  ```

  or

- Using the `-regression` argument of the `xeDebug` command:

  ```
  xeDebug -regression
  ```

With the `-regression` option, a smaller version of the database is loaded — the overall run-time environment preparation time is reduced and the `host` command execution is faster.

Note: This functionality to debug using a smaller version of the database is supported for any design and in any mode and does not require any special compilation such as modular compilation or compilation using Parallel Partition Compiler (PPC).

The system ensures that all the critical nets needed for basic run are preserved. The collection of nets saved for run time are called preserved nets. The preserved nets include the following:

- Internal instrumentation nets
- User-defined keepnets — that is, the nets added by the `keepNet -add` user-data command. The preserved nets do not include the keepnets created by instructions from any phase of compilation.

The number of preserved nets are design-dependent and do not affect the run-time performance, however, the start-up time and the memory usage might increase if the preserved nets are large in number.

By default, the information about bit-blasted memories is saved in a separate side file generated at `/dbfiles`. If information about bit-blasted memories is not required, you can turn off the side file generation using the `compilerOption -add -EnableExtraFileForRegressionMode OFF` command in your compile script. By

© 2021 All Rights Reserved.

default, full memory information is not saved for better performance. If, however, you need full memory information in the side file, then use the `compilerOption -add dumpFFMemoryFullInfo ON` command.

`value`, `force`, `release`, `deposit` commands, and SDL are supported for preserved nets. The `memory` command is also supported. All the run-time commands that do not depend on the Palladium database work as expected.

Limitations of Debugging in Regression Mode

The commands that depend on the Palladium database work only for the nets from the preserved collection. The functionality of `debug` command with preserved nets might be limited because only the important properties are saved.

Following limitations are applicable to the run-time commands when the debugging session is invoked in the regression mode:

- Regular `probe` command is not supported. The streaming probes, however, work for preserved nets.
- Wild cards are not supported.
- RTL name format can be used for single bits and not full constructs or sub-constructs. No name translation or correction is done and only the exact match to the gate or RTL names is saved in the preserved nets file.
- To add more preserved nets, you must execute the `keepNet` command and redo the backend compilation.
- Debugging in regression mode is not supported for designs compiled using the Vector Debug (VD) mode.
- Debugging in regression mode is not supported for designs with Acceleratable Testbench (ATB) Constructs.
- Phy file generation with the `database -prepareOffline` command is not supported in the regression mode if Dynamic Probes (DYNP) mode is used.

The following table lists the supported information and the restricted behavior of various run-time commands in the regression mode:

© 2021 All Rights Reserved.

Table 38-2 Support and Restrictions for Run-Time Commands in Regression Mode

Command	Support in Regression Mode
assertion	Supported through xmsim only
bm	Supported
clockConfig	Supported only in ICE mode. Not supported in SA mode.
clockConfig -stopMode	Supported
configPM	Supported
convertRadix	Supported
database -open	Supported
database -open <*trace_name*> -cont -phy	The generated *.phy file supports the whole design.
database -open <*trace_name*> -cont	The generated *.phy file supports the whole design. The SHM/FSDB file is not generated.
database upload	Not Supported
database -captureMode	Supported
database -prepareOffline	Supported **Note:** In regression mode, the database -prepareOffline command is not supported with the DYNP mode.
database -tracewindow -start <*time*> -end <*time*>	The database -tracewindow command is ignored because the SHM/FSDB file is not generated.
database -partialfv -add <*instance_name*> [-exclude]	Not Supported
database -partialfv -clear	Not Supported
database -partialfv -list	Not Supported

© 2021 All Rights Reserved.

Command	Support in Regression Mode
database -partialfv -rm <*inst1_name*> <*inst2_name*>... <*instN_name*>	Not Supported
database -stream -close	Supported only for preserved nets
database -stream [-open] <*streaming_db_name*>	Supported only for preserved nets
database -stream -disable | -enable	Supported only for preserved nets
database -stream -exitWait [1 | 0]	Supported only for preserved nets
database -stream -overflow [ignore | warning | error]	Supported only for preserved nets
database -stream -show	Supported only for preserved nets
database -stream -startStream	Supported only for preserved nets
database -stream -stopStream	Supported only for preserved nets
database -stream -upload	Supported only for preserved nets
database -stream -waitWhileBusy	Supported only for preserved nets
db	Not Supported
debug	Supported
deposit	Supported only for keepnets.
deposit -allff	Not Supported
download	Supported
dpa	Not Supported
drtl	Supported only for keepnets.
force	Supported only for keepnets.
getCableStatus	Supported

© 2021 All Rights Reserved.

Command	Support in Regression Mode
getCycleNum	Supported
getHostStatus	Supported
getPMStatus	Supported
getTime	Supported
hdlConfig	Not Supported
host	Supported
host -dynp	Not Supported
infinitrace	Supported Forceable nets must be keepnets. It is not mandatory that the Observe session must have the same bucket open as the Prepare session.
lp	Not Supported
memory	Supported
mergeSAIF	Supported
power	Not Supported
probe (in FV)	Not Supported
probe (in partial FV)	Not Supported
probe (in DYNP)	Not Supported
probe -stream	Only user-defined keepnets might be used as SDL input.
release	Supported only for keepnets.
reset	Supported
resume	Supported
restart	Supported
run	Supported
stop	Supported

© 2021 All Rights Reserved.

Command	Support in Regression Mode
save	Supported
scope	Not Supported
sdl	Supported
suspend	Supported
symbol	Not Supported
targetAssign	Supported
targetLocation	Supported
tca	Not Supported
userData	Supported
value	Supported only for keepnets.
waitWhileBusy	Supported
xeset	Supported
xegui	Supported

© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

39

Run-Time Commands Support

This section provides support information about the XEL run-time commands in the ICE and SA modes.

For syntactical details of the listed commands, refer to the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Table 39-1 XEL Run-Time Commands Support Matrix

Run-Time Commands	ICE Mode	SA Mode

ATB commands:
`hdlConfig`
`hdlRunLogfile`
	Supported	ATB feature is not supported.

`clockConfig`	Supported	This command is typically not supported in SA mode because of the differences in the modeling of the clocks. However, some options of the `clockConfig` command are partially supported if there are uncontrolled clocks.
`configPM`	Use switches: `-stb` and `-la`	Different switch names need to be used: ■ `-staticTarget` instead of `-stb` ■ `-dynamicTarget` instead of `-la`
`lp`	Supported	Supported when using the xeDebug tool and not supported in *xrun*.

© 2021 All Rights Reserved.

deposit	Supported	The `-allff` option is not supported. Subtle difference in behavior by default, when compared to ICE mode. The behavior in ICE mode can be configured to be the same as SA mode by setting the `depositSimMode` run-time parameter.
download	Supported	Not supported. Instead, execute the `xc xt0 zt0 autorun` command.
force	Supported	The `-allff` option is not supported.
getCycleNum	Supported	This command is supported, but the result might be different from ICE mode due to difference in modeling.
getTime	Supported	The `dclk` and `fclk` units of time are not supported
memory	Supported	Some command options not supported.
release	Supported	The `-allff` option is not supported.
resetCycleNum	Supported	Not supported
resultWaveform	Supported	Not supported
resume	Supported	Supported
run	Supported	Supported except: ■ Amount to run cannot be specified using the number of clock cycles (only accepts simulation time).
suspend	Supported	Supported

© 2021 All Rights Reserved.

40

Probing Design Signals and Generating Waveforms

This chapter discusses how to probe signals and upload waveforms in SA and ICE mode using command mode. It provides information on various debug vision modes: FullVision (FV) and Dynamic Probe (DYNP) and also discusses about debugging in offline mode.

For information on how to use the xeDebug GUI for debugging designs, refer to Appendix 42, “Using the xeDebug GUI”.

This chapter includes the following sections:

- Tracing Signals Using Probes
  - Setting Probes
  - Uploading Probes
  - Probing Modes
  - Changing the Vision Mode at Run time
  - Probing SystemVerilog Design Constructs for SST2 and FSDB Waveforms
  - Viewing Two-Dimensional Arrays in SimVision
- Tracing Signals Using InfiniTrace
  - InfiniTrace Sessions
  - Using InifiniTrace in ICE Mode
  - Using infiniTrace Command Options in HWIT Mode
  - Using InfiniTrace in SA Mode Using xeDebug
  - Using InfiniTrace in SA Mode Using xrun
  - Comparing the InfiniTrace Feature in SA and ICE Modes
  - Bookmarking InfiniTrace Session

© 2021 All Rights Reserved.

- Tracing and Uploading Waveforms
  - Tracing and Uploading Waveforms in ICE Mode Using xeDebug Tool
  - Tracing and Uploading Waveforms in SA Mode Using xeDebug Tool
  - Tracing and Uploading Waveforms in SA Mode Using xrun
- Generating Waveforms in Batch Mode with Partitioned FullVision
- Generating FSDB Files for Specific Instances
- Debugging in Offline Session
  - Starting an Offline Session
  - Using Offline Debugging
  - Additional Debugging Features in Offline Session
  - Generating Waveforms in Offline Batch Mode
  - Generating Waveforms in Offline DYNP Mode
  - Compressing *.phy Files
  - Streaming Waveform Data

Tracing Signals Using Probes

Two vision modes to trace the design signals are Full Vision (FV) and Dynamic Probe (DYNP).

In FV, you can define any signal as a probe. There is no limitation on number of probes. You can define probes before running the design or after running the design. You can define and upload probes multiple times.

However, in DYNP, you must define probes before running the design. Also, the maximum number of probes that can be defined is limited.

Probes can be either virtual or physical. Physical probes are based on actual data obtained via emulation. Virtual probes are probes not obtained directly from emulated logic, but instead calculated in software from physical probe data.

When data tracing stops, either due to a trigger or exhausting the requested cycle count, the physical (raw) probe data can optionally be uploaded from the hardware to a disk file (by default called `trace.phy`, and can be renamed by using the xeDebug GUI or command options). After additional transformations that calculate the virtual probes, the data is written

© 2021 All Rights Reserved.

to the output file, called waveform database, which is displayed in the waveform tool. The default format of the output file is SST2. By using the run-time menus or commands options, you can specify a different name for the output file. You can choose to view the waveforms in the FSDB format by specifying the `-fsdb` option of the `xeDebug` command at the time of invoking xeDebug. After invoking xeDebug, you cannot change the waveform file format. The FSDB format is supported to enable easy debug of Palladium-generated databases when using Novas debugger tools like Verdi and Siloti. For more information on using Novas Tools, refer to Chapter 47, “Using Novas Tools.”

```
xeDebug -fsdb -gui &
```

The `database` command supports only one active database at a time. A request to open a database closes the current database before opening a new one. At the same time, `database` command directly passed to Xcelium has no limitation on the number of databases.

VXE software uses 64-bit integer to hold simulation time. If design is compiled with too fine time precision then it is possible that simulation time might overflow the 64-bit integer. In this case the `database -upload` command will issue an error. To avoid this problem it is suggested to compile the design with `1ns` time precision. You can set the time unit and precision as follows:

```
-vhdl_time_precision 1ns  //for VHDL
-timescale 1ns            //for Verilog
```

Setting Probes

Probes can be set in both ICE and SA modes, and in any part of the design.

If the design is compiled for FV (Full Vision) mode, after viewing results, more probes can be added and uploaded for viewing without re-running the emulation, either by requesting the new probes to be uploaded through the Debugger menus or by issuing a `database -upload` command. If the design is compiled for DYNP (Dynamic Probes) mode, after adding new probes, the design must be run before the trace for these probes can be uploaded and viewed in the waveform browser. In DYNP mode, adding the new probes only affects future runs. There is no way to retrieve the traced data for a previous run. In FV mode, you can add probes after a run and still retrieve your traced data from that run.

© 2021 All Rights Reserved.

Table 40-1 Probing Options According to Vision Modes in ICE and SA Modes

Debug Mode	FullVision (FV)	Dynamic Probes (DYNP)	Dynamic Target	InfiniTrace
				Hardware InfiniTrace (HWIT)
Vector Debug (VD)	Yes	Yes	No	No
Synthesizable TestBench (STB)	Yes	Yes	No	Yes
Logic Analyzer (LA)	Yes	Yes	Yes	No
Simulation Acceleration (SA)	Yes	Yes	Yes, but limited	Yes

The probe run-time commands also enable you to probe the CPF and 1801 objects into the probe set. The following probe commands are applicable to CPF and 1801:

- Probe commands for CPF objects

```
probe -addIllegalDomainConfiguration {<illegalDomainConfiguration> | *}
probe -addMode {<modeName> | *}
probe -addModeTransition {<modeTransitionName> | *}
probe -addPowerDomain {<powerDomainName> | *}
probe -addPowerMode {<powerModeName> | *}
probe -addIsolation {<isolationRuleName> | *}
probe -addRetention {<stateRetentionRule> | *}
```

- Probe commands for 1801 objects

```
probe -addpowerdomain [<namePattern>]
probe -addisolation [<namePattern>]
probe -addretention [<namePattern>]
probe -addsupplyport [<namePattern>]
probe -addsupplynet [<namePattern>]
probe -addsupplyset [<namePattern>]
probe -addswitch [<namePattern>]
probe -addpst [<namePattern>]
probe -addportstate [<namePattern>]
probe -addpststate [<namePattern>]
probe -addlevelshifter <shifterName>
```

For information on the description of the above mentioned probe commands, refer to the description of the probe run-time in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

© 2021 All Rights Reserved.

For more information on probing the 1801 low power objects, refer to the Probing Low-Power Objects and Viewing Waveforms section. For information on probing the CPF objects, refer to the Running and Debugging a Compiled Design with Low-Power Intent section.

Uploading Probes

The uploaded probed data can be used for:

- Working in online session on the emulator and viewing emulation results online immediately.
- Working in online session, but not viewing the results immediately; instead, the data is saved to a physical probe file for later analysis in offline debug mode.
- Working in offline session (without an emulator) and viewing the results from a physical probe file.

Uploading data from a design with numerous probes can be time consuming. It might be desirable to only use a subset of the total probes.

You can also find signal values for the current emulation time (a single time point) by issuing the `value` command from the xeDebug command line.

Probes can be set using either the GUI or the CLI. Refer to Probe Tab on page 1941 for using the xeDebug GUI options. Steps to set probes and view the waveforms using commands are documented in the following section.

Setting Probes through Commands

1. Assign probes to the specified nets in the design using the `probe` command.

   Note: When debugging the design using *xrun*, do not use the `probe` command to set probes in the DUT. *xrun* does not access the DUT correctly unless the simulation control is swapped into the simulator. For more information on setting probes using *xrun*, refer to the *Run-Time Commands* chapter in the *VXE Command Reference Manual*.

2. Create a probe during run time using the following command:

   ```
   probe -create <options>
   ```

   You can add all signals in the nets' input and fanout cone of logic to the trace signal list using the `probe -addcone -depth` <*n*> <*signal*> ... and `probe -addfanout -depth` <*n*> <*signal*>... command respectively.

   To add all register outputs of a specific module instance and its underneath hierarchies to the probe set, use the `probe -addinstreg` <*instanceName*> command. You can

© 2021 All Rights Reserved.

also add all input ports of the specified module instance and its underneath hierarchies to the probe set — To do this, use the `probe -input_port <instanceName>` command.

3. Upload the trace results from the emulator into the SHM or FSDB waveform database using the following command:

```
database -upload
```

For more information on the `probe` command and all options, refer to the description of the `probe` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Probing Modes

The required mode of probing is selected at the compile time. The following probing modes are supported in VXE:

- FullVision Mode (FV mode)

  Note: This is the default mode.

  - Selective FullVision (FV)

- Dynamic Probes Mode (DYNP mode)

FullVision Mode

FullVision mode enables you to probe and view the waveform of any design signal at run time, without recompiling or rerunning. With this mode, you can:

- Specify probes to be uploaded before running the design
- Add and upload additional probes, after running the design
- Add and upload probes as often as needed, without re-running the design.

FV mode provides full visibility to the design. Waveform of any net in the design can be viewed using FV mode. In FV mode, the compiler automatically adds enough probes to ensure the full visibility of all signals. Unlimited number of probes can be added either before or after running the emulation. All the nets in the design can be probed.

With FV mode, offline session can be used. When waveform is uploaded during online session, all the data which is necessary for FV mode is uploaded from emulator and is saved

© 2021 All Rights Reserved.

in the `*.phy` subdirectory. An offline session can be started afterwards in which the `*.phy` subdirectory is used to generate waveform for any nets in the design. The `*.phy` subdirectory is generated by `database -upload` command or `database -prepareOffline` command during online session.

The FV overhead is the resource required to capture signal values into Data Capture Card (DCC). The number of signals that can be captured per domain per step is fixed. If there are more signals to be captured, consequently either more domains or more steps are required. Trace depth is also affected by the number of signals. The fewer the signals the larger the trace depth.

The ***advantage*** of using the FV mode is:

- An unlimited number of signals can be probed. The signals to be probed do not need to be chosen before running.

The ***disadvantages*** of using the FV mode are:

- FV mode might have larger step count than DYNP mode for some designs and therefore, slower emulation speed. For more information, refer to the Emulation Speed Overheads section.
- The time to upload data from the emulator to the disk depends on the size of the design and can be significant for large designs. This is true even if the number of probes requested to be viewed on the waveform is small.
- Maximal trace buffer depth and upload time can be adversely affected when using conditional acquisition due to the frame-based tracing in FV mode. That is because a whole trace frame (typically 32 consecutive cycles) is captured at any time SDL requests to trace any of the cycles in the frame.

You can reduce the load and memory usage of the local host for FV computation by using the `CDN_FV_NO_LOCAL_HOST` environment variable. When this environment variable is set to `1`, the FV and DPA computation tasks are performed only on the remote hosts:

```
setenv CDN_FV_NO_LOCAL_HOST 1
```

Selective FullVision (FV)

When you need to debug only specific instances or modules in the design, FV can be applied only on those specific instances or modules. With selective FV mode, you can enable or disable specific instances and modules and set probes on the enabled region. Selective FV can be used in both ICE and SA modes.

Note: FullVision is applied only to the enabled region in the design.

© 2021 All Rights Reserved.

To enable selective FV, `compilerOption -add visionMode` option must be set to `FV`.

To enable or disable specific instances and specific modules, use the `instanceFV` command and the `moduleFV` command at compile time respectively. By default, FV is enabled on the entire design if nothing is specified with the `instanceFV` and `moduleFV` User Data commands.

The following command enables or disables FV on modules:

```
moduleFV -add {<LIB cell> <enable | disable>}
```

Use the following command to enable or disable FV on instances:

```
instanceFV -add {<instance_name> <enable | disable>}
```

For more information on `moduleFV` command and the `instanceFV` command, refer to the *User Data Commands* chapter of the *VXE Command Reference Manual*.

Advantages

- Faster upload speed can be achieved at run time.
- Smaller `*.phy` subdirectory size because fewer nets are probed.

Disadvantages

Depending on how a design is mapped to domains, all available DCCs might not be used because of limited domain-to-domain and board-to-board connections. Therefore, selective FV does not ensure linear improvement on performance and trace depth. For example, if you enable FV on half of the design, you might not get two times improvement on performance and trace depth. The actual improvement might be less than two times.

The following Table 40-2 marks out the differences between selective FV and FV mode.

Table 40-2 Differences between Selective FV mode and FV mode

	Selective FV	FV
Full visibility	No, only on enabled instances/modules	Yes
Adding probes and uploading waveforms	Yes, only on enabled instances/modules	Yes

© 2021 All Rights Reserved.

Offline debugging	Yes, only on enabled instances/modules	Yes
InfiniTrace	Yes	Yes
Speed overhead	Less	More
Maximum trace depth	Same or Larger	Same or Smaller

Reducing FullVision Overhead using Run-Time Partial FullVision

Waveform generation involves the following three stages:

1. **DCC upload**: Data is written from the emulator to the disk.
2. **fvCompute**: Data is processed to make a waveform.
3. **On-demand Waveform Generation**: Signals are displayed as waveforms.

FullVision computation, that is, the fvCompute process takes time because FV data is computed for the whole design. The time for FV compute grows significantly as the design size grows. You can reduce the FV overhead by limiting it to a small part of the design. Specify the module that you need to debug and the run-time software can generate the waveform for that module faster.

In an online session, partial FV enables you to reduce the FV overhead by partitioning the design at run time — that is, FV can be applied partially to compute only for some specified instances. You can specify the partitions before the clocks are run. These partitions can then be selectively calculated through specifying appropriate probes, or using fvCompute on the `*.phy` file — thus avoiding fvCompute on the entire design.

Isolating Instances

To apply FV partially at run time, isolate the instances you need to debug before the run, using the `database -partialfv -add <instance_name>` command.

When you specify the instances to be debugged, the software identifies the boundary of that instance and adds the signals on the boundary as physical probes. Those boundary signals partition the design into two parts: inside the module and outside the module. If you generate waveform for signals inside the module, the FV compute is done only for that module.

For example, suppose instance `Top.A` is isolated for partial FV.

- If you probe signals only *inside* `TOP.A`, the FV compute will be limited to only `TOP.A`.

© 2021 All Rights Reserved.

- If you probe signals only *outside* `TOP.A` (that is, the non isolated part), then only the non-isolated part will be calculated — and `TOP.A` will not be calculated.
- If you probe signals *outside* and *inside* `TOP.A`, fvCompute will happen on the entire design similar to FV.

The run-time software selects the partition to compute based on a set of added probes. Therefore, computation is limited to the smallest necessary sub-set of the design, and performance is fast as compared to computing the whole design.

A message is added to the `xe.msg` file to let you know about how the design is partitioned into multiple segments.

Excluding Instances for Partial FV Computation

The fvCompute process can be invoked in two ways:

- **With xeDebug**, fvCompute is invoked on executing the `database -upload` command. Only the partitions that have been probed are computed. For example, in the following command, only the partitions that have `u1` will be computed:

  ```
  probe -create u1 -depth
  database -upload
  ```

- **Without xeDebug**, you can invoke the fvCompute process directly to preprocess the `.phy` file by executing the following command from the unix prompt:

  `fvCompute` <*base_name_of_phy*>

  No probes are specified because xeDebug is not being used. For more information about the `fvCompute` command, refer to Optimizing Trace Data for Faster Interactive Offline Debugging section and the description of the `fvCompute` command in the *Programs and Utilities* chapter of the *VXE Command Reference Manual*.

  In Fullvision, fvCompute computes all nets in the design. With Partial FV, fvCompute computes only the partitions with the instances specified in the `database -partialfv -add` <*instance_name*> command.

  Alternately, you can specify a partition to exclude from computation by specifying the `-exclude` option, as follows:

  `database -partialfv -add` <*instance_name*> `-exclude`

  For details on the `-exclude` option and examples, refer to the description of the `database -partialfv -add` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

© 2021 All Rights Reserved.

Using Partial FV in Online Sessions

1. Define isolation on the instances you plan to debug, and partition the whole design into multiple small segments, as follows:

   `database -partialfv -add <instance name>`

   For example,

   ```
   database -partialfv -add u1
   database -partialfv -add u1.u0
   ```

2. Define the required probes using the `probe` command. For example,

   `probe -create u1 -depth 1`

3. Run the design by executing the `run` XEL command. For example,

   `run 1000000`

4. Generate the waveform using the `database -upload` command.

Using Partial FV in Offline Sessions

The partition information is saved in the `*.phy` file. The `.phy` file has multiple segments in run with partial FV. However, without partial FV, the `.phy` file has a single segment.

The software uses the partition information in the `*.phy` file to speed up the performance. If you add probes in smaller partition, the waveform generation speed will be fast. If the probes specified are in the isolated block, then only the isolated block portion is computed for FV.

The following example shows a comparison of complete and partial FV on a design with 20M nets:

- Complete FV: The `*.phy` file has one segment of 20M nets. To probe one small instance in a large design, the entire FV database will need to be computed.
- Partial FV: The `*.phy` will have multiple segments, such as:

  `segment 0 with 100K nets, segment 1 with 5M nets, segment 2 with 14.9M nets.`

  Only the segments that have probed nets will be computed. For example, if all probed nets are in segment 0, then only segment 0 is computed.

The steps to use partial FV in offline session are same as regular FV in offline mode.

You add probes and then use the `database -upload` command to generate the waveform. The run-time software receives the partition information from the `*.phy` file. The software then determines the partition that needs to be computed based on the probes that have been added. If all the probes are inside the smaller partition, the waveform is generated fast. If,

© 2021 All Rights Reserved.

however, some probes are inside the large partition, the software might take longer time for waveform generation.

The longer time is only observed when waveform is generated for the first time. This action might trigger FV computation for relevant segments. Once FV computation is done for those segments, the FV data is saved to the `*.phy` file for those segments. Next time, if you need to probe more signals in those segments, the waveform generation will be fast because the software reuses the saved FV data for those segments.

The isolations cannot be defined in the offline sessions because FV computation is based on the probes captured in DCC — Partial FV identifies boundary nets when isolating instances. These boundary nets are added into the DCC. Therefore, the instance isolation must be done online, and at 0 run time. Clocks can be run in simulation in all modes except SA mode. If, however, DUT is swapped into the emulator, you must add the partitions before running any clocks.

Examples

The following examples help you determine how the instances should be isolated for improved performance:

- Suppose you need to debug the whole design that has a big IP block `Top.A.B`, which is not to be debugged. In this case, set isolation on `Top.A.B`. When you generate waveform for signals outside `Top.A.B`, the FV computation will skip `Top.A.B`. The performance will increase depending on the size of `Top.A.B`.
- To debug `Top.A.B1, Top.A.B2`, and `Top.A,` set isolation on all three instances. If you generate waveform for signals inside `Top.A.B1` and `Top.A.B2`, the FV computation will be limited to those instances, and will be fast. If, however, you generate waveform for signals inside `Top.A`, FV computation will be more, however, it will still be faster than computing the whole design.

Restrictions and Limitations

Some limitations of partial FV are as follows:

- Partial FV does not make DCC upload faster, that is, the `database -prepareOffline` command still uploads DCC for the whole design.
- The boundary signals are added as run-time probes. Therefore, it might be possible that there are not enough resources to add those probes. In such cases, recompile the design with `compilerOption -add {numExtraCapturedNetsPerDomain <num>}` command so that the compiler reserves more resources for run-time probes.

© 2021 All Rights Reserved.

- Partial FV is not supported with save and restart operations.

Partitioning FullVision Database

Multiple FV partitions (`swfvDBs`) can be generated in the following ways:

1. Default partitions, which are enabled with `fvPartition on`. Default partitioning can only be done using the PPC. You can leverage the Parallel Partition Compiler (PPC) and use the partitions created by PPC as the default partitions in the FV database. PPC can be enabled in multiple ways. See Enabling Parallel Partition Compiler (PPC) for more information.
2. Run-time partial FV, which is enabled with `database -partialfv` command. See Using Run-Time Partial FV with Auto-Partitioning for more information.
3. Compile-time FV, which is enabled with `instanceFV` and `moduleFV` commands. See Partitioning Designs at Compile Time for more information.

During partial FV, you need to specify the instances to add the partitions before running and uploading the DCC. That is, you need to plan in advance about the portions to be debugged.

If, however, several boundary nets exists in the design, there is a possibility for the design to run out of resources. Such issues can be addressed at compile time by automatic creation of the default partitions in the FullVision database (`swfvDB`).

Enabling Creation of Default Partitions

You can enable or disable the default FV partitions at compile time by using the following compiler option:

```
compilerOption -add {fvPartition <on|off>}
```

Multiple segments are created after compile if default partitioning is enabled along with PPC. For syntax details and more information on the `compilerOption` command, see *User Data Commands* chapter in the *VXE Command Reference Manual*.

Enabling Parallel Partition Compiler (PPC)

PPC can be enabled in the following ways:

- By explicitly defining the partitions using the `partitionGroup` command, as follows:

  ```
  partitionGroup -add {<instance>[.][<prer>][*]<partition>}
  ```

  For example:

© 2021 All Rights Reserved.

```
partitionGroup -add {dut.a.b. part1}
partitionGroup -add {dut.a.b.r* part2}
partitionGroup -add {dut.a.c* part3}
partitionGroup -add {* part3}
```

For details on how to use the PPC, refer to Using Parallel Partition Compiler for Large Designs section.

- Using Auto-Partition: You do not define the partitions. The compiler automatically decides how to partition the design. To enable auto-partitioning, use the xeCompile `autoPart` command at compile time.
- Using Advanced Partitioning: To enable advanced partitioning, use the `compilerOption -add {advancedPart ON | OFF | <N>}` in your compile script.

Using Run-Time Partial FV with Auto-Partitioning

xeDebug supports the run-time partial FV with the default partitions. To gain finer granularity than the partitions defined by the PPC, use the run-time partial FV command:

`database -partialfv -add <inst>`

The run-time software determines whether or not to create new partitions or use the default partitions. For example, if the specified instance exists in one of the PPC partitions, and the nets in this instance are greater than 50% of the PPC partition, then the default partition is used. If, however, the specified instance exists in several PPC partitions, then a new partition is created.

In offline sessions, partial FV only computes the partitions required to display the probes specified by you.

For more details on partial FV, refer to Reducing FullVision Overhead using Run-Time Partial FullVision.

Without xeDebug, `fvCompute` can be used as follows:

- You can run fvCompute for a specified partition, as follows:

  `fvCompute -partition <partition_id>`

- You can run `fvCompute` on specified predefined isolated instances, as follows:

  `fvCompute <phy file> -pfv <list_of_instances>`

  To generate a list of predefined isolated instances, and the total number of computed blocks associated with the `*.phy`, execute the following command:

  `fvCompute <phy file> -listpfv`

  For example,

© 2021 All Rights Reserved.

```
fvCompute trace.phy -listpfv
top.m1 0/500 blocks computed
top.m1.u1 500/500 blocks computed
```

- If fvCompute is used without options — for example, `fvCompute <base_name_of_phy>`, then computation is done on the instances specified by the `database -partialfv -add` command. If, however, the `database -partialfv -add` command is not specified, then all partitions are computed.

Partitioning Designs at Compile Time

Use the `instanceFV -add` [`<instance_name>` `enable`] command at compile time that enables you to define the instances to debug at run time. During precompile, the PI probes are set along the boundaries of the `instanceFV` enabled regions.

Note: The `compilerOption -add` {`fvPartition <on|off>`} command is not supported with the `instanceFV` command. The compiler generates an error message if you use both the commands together. For more information on these commands, refer to the *User Data Commands* chapter of the *VXE Command Reference Manual*.

Reducing FullVision Samples

By default in SA mode, FullVision captures the data transitions for both behavior and RTL events. You can reduce the number of samples captured during FullVision, and force the FV computation for only the RTL events. For example, a SFIFO call is implemented as a behavior event. Several SFIFO calls might exist in the same time step, which can cause FullVision to capture all other samples that might not be useful for debugging. In such cases, consider forcing the FV computation to only the RTL events, thereby reducing the FV captured samples.

To reduce the samples, execute the `xc FvSimple 1` command before the first run. This command enables FV simple capture mode that ignores the behavior events, and captures only the RTL events. The number of captured FV samples are reported in the `xe.msg` file after FV upload. By default, the simple capture mode is off.

Note: The `xc FvSimple 1` command is supported only in SA mode.

Each sample requires FV samples events in 32 consecutive fclk cycles at a time. This is called an FV frame. By default, if RTL or behavior cycles are active in a frame, the entire 32-cycle frame is used. However, in FV simple mode, a frame is used only if RTL cycles are active within the frame.

The number of FV samples reduced by the `xc FvSimple 1` command depends on the design and testbench. If the design has behavior blocks or the testbench has SFIFO calls, the

© 2021 All Rights Reserved.

number of samples will be smaller than the default FV mode. Use the `xc FvSimple 1` command if your testbench has several SFIFO calls in one synchronization. For example, when one address is written at a time to a big block of memory in one synchronization.

Limitations

Behavior cycles cannot be debugged in waveforms when FV is run in simple mode. To get the behavior cycles captured during FV, turn off the FV simple mode using the `xc FvSimple 0` command.

Dynamic Probes Mode

DYNP mode allows the dynamic probing of any net in the design. In DYNP mode, you must define probes before running the emulation as waveforms of any nets in the design cannot be generated. Dynamic probes need to be configured on the emulator hardware.

The number of nets that can be probed depends on the number of emulator domains for which the design is compiled, and `numCapturedNetsPerDomain`.

Caution

If the resources are not sufficient for probing, adding a set of probes can fail.

To increase the number of probes that can be added:

- Increase the number of domains for compiling the design
- Increase `numCapturedNetsPerDomain`

Following are the ***advantages*** of using the DYNP mode:

- Enables fastest waveform upload.
- Avoids the costs related to emulation speed that occur in designs when large number of nets need to get probed in FV mode.
- Provides a larger maximum possible trace depth.

Following are the ***disadvantages*** of using the DYNP mode:

- Provides limited number of signals that can be probed in a specified run.
- Probes need to be defined before running the emulation.

© 2021 All Rights Reserved.

Supported Probing Strategies for Each Probing Mode

Different probing strategies are supported under different debug modes. Table 40-3 shows what functionality is available for all emulation debug modes.

Table 40-3 Palladium Z1 Probing Options by Debug Strategy

Debug Mode	Dynamic Probes (DYNP)	FullVision (FV)	InfiniTrace
Vector Debug (VD)	Yes	Yes	No
Static Target TestBench (STB)	Yes	Yes	Yes
Logic Analyzer (LA)	Yes	Yes	No
Simulation Acceleration (SA)	Yes	Yes	Yes

For designs compiled with the STB mode in ICE flow and in SA mode, InfiniTrace is supported. The support for InfiniTrace is turned on by default. If not required, you can disable it during compilation.

Comparison between FV and DYNP Modes

Following table summarizes the comparison between DYNP and FV modes .

Table 40-4 Comparison between DYNP and FV mode

Differences	DYNP	FV
Full visibility	No	Yes
Probes can be added after run and upload waveform	No	Yes
Offline debugging support	No	Yes
InfiniTrace support	Yes	Yes

© 2021 All Rights Reserved.

Differences	DYNP	FV
Speed overhead	No	Speed overhead happens with small number of designs. For more information, refer to the Emulation Speed Overheads section.
Maximum trace depth	Larger	Smaller
Conditional acquisition	Cycle-based	Frame (32 cycles) based

The following sections describe some of the main differences between FV mode and DYNP mode in detail:

- Emulation Speed Overheads
- Trace Buffer
- Conditional Acquisition

Emulation Speed Overheads

For most designs, FV doesn't have any overhead. However, for small number of designs, FV might cause larger step count resulting in slower emulation speed.

In FV, Palladium compiler needs to probe large number of signals. However, the number of probes depends on the step count, because there is a limitation on the number of probes that can be defined in one step. The Palladium compiler reports two step counts in the `xe.msg` file. One step count is for arranging the probes and the one is for design logic. The final step count is larger than these step counts. In some designs, the first step count is larger than the second step count. If the design was not compiled in FV mode, the step count number could be smaller.

Trace Buffer

During emulation, the trace buffer is used for capturing the trace data. If the trace buffer is filled up, the oldest data in trace buffer is replaced by the newest data. The trace buffer always contains only the latest trace data.

© 2021 All Rights Reserved.

When uploading a waveform, you can specify a time window ending at the last emulation cycle. This time window is called trace window. The trace window contains the newest captured data. The trace window size can be specified by using the `xeset traceMemSize` command. The trace window size must be smaller than trace depth. The maximum size of trace window that you can specify is the trace buffer depth.

Conditional Acquisition

In DYNP mode, the decision to capture or discard is taken at each fclk cycle.

In FV mode, the decision to capture/discard is taken per unit of frame, which is normally 32 fclk cycles. If a FCLK cycle within a frame needs to be captured, then the whole frame is captured. This has the disadvantage that a lot of unwanted fclk cycles are captured. The extra cycles can also be seen in the waveform, and the maximal trace depth is reduced because of these extra cycles.

When using conditional acquisition, the requested trace window size as specified by `xeset traceMemSize` refers to the cycles requested to be acquired by SDL, and not to the actual number of traced cycles. When using FV probing mode, the actual number of traced cycles can be higher than the number requested because the acquisition is frame based, so a whole frame (typically 32 cycles) is captured whenever any of the cycles in that frame is requested to be captured.

Parallel Computation for FullVision using Multiple CPUs and Multiple Hosts

The VXE software has the same computation engine for FullVision probes and Dynamic Power Analysis (DPA). Therefore, computation of FullVision probes can also make use of multiple CPUs and hosts as DPA does. For detailed information, refer to the following two sections of Chapter 32, “Analyzing Power Consumption of the Designs”:

- Parallel Computing for DPA Using Multiple CPUs
- Running DPA with Multiple Hosts

Using FullVision in Safe Mode

You can use FullVision in safe mode in SA mode when the waveform upload fails with verification errors. The errors are usually due to missing samples in the uploaded probe stream. Execute the `xeset fvSafeMode 1` command to use FullVision in safe mode. By default, safe mode is disabled for FullVision.

© 2021 All Rights Reserved.

Controlling Maximal Trace Depth

The maximal trace depth can be controlled only in the DYNP mode.

Following are the different scenarios by the combination of `traceDepth` parameter and `numCapturednetsPerDomain` parameter, which are specified through the `compilerOption` command during the compilation process:

- `traceDepth` and `numCapturedNetsPerDomain` both are not specified

  In this case, you do not need to know the number of probes that can be added and the trace depth. Here, the default value depending on the design is used. For large designs, the default value of `traceDepth` is 4M and the default `numCapturedNetsPerDomain` is 16K. For small designs, the default value of `traceDepth` is larger because the design does not need larger number of probes and `numCapturedNetsPerDomain` can use a smaller value. For example, a very small design can have default `traceDepth` of `4M`.

- Only `numCapturedNetsPerDomain` is specified

  In this case you might want to add more probes at run time because the default value of `numCapturedNetsPerDomain` is too small for the design. `numCapturedNetsPerDomain` value must be in the range of `512` to `1572864`.

- Only `traceDepth` is specified

  In this case, you might want more trace depth because the default value of `traceDepth` is too small for the design. For example, if in a huge design only 100 I/O signals are needed for a long period of time, the available trace depth can be 128M cycles (`numCapturedNetsPerDomain` is 512 in this case).

  Note: To get a large trace depth, specify the number of probe channels using the `numCapturedNetsPerDomain` option. This option triggers a probe routing mechanishm, which can provide a large trace depth. For example, to get 128M trace depth, specify `numCapturedNetsPerDomain` as 512. If, however, you specify `traceDepth` as 128M, the compilation will fail.

- Both `traceDepth` and `numCapturedNetsPerDomain` are specified

  In this case both long trace depth and large number of probes are required. This requirement is difficult to meet. The compiler calculates the product of `traceDepth` and `numCapturedNetsPerDomain`. If the product is larger than 8G, then compiler errors out. Otherwise, the compiler accepts `numCapturedNetsPerDomain` value and ignores `traceDepth`. The trace depth results in 8G/numCapturedNetsPerDomain.

© 2021 All Rights Reserved.

Specifying traceMemSize

Use `xeset traceMemSize` at run time to specify how many cycles to upload. `traceMemSize` must be smaller than the maximum trace depth.

You can set `traceMemSize` in the unit of simulation time.

For example:

```
run 1000ns
xeset traceMemSize 10ns
database -upload
```

The above example results in a waveform from `990ns` to `1000ns`.

`traceMemSize` can be set in the following ways:

- Set value and unit. This value is a floating point value. For example:

  ```
  xeset traceMemSize 0.012s
  xeset traceMemSize 1.2e-2s
  ```

- Set value only. In this case the specified unit uses the current value for `xeset timeUnit`.

- Set fclk cycles. Use `fclk` as the unit. In this case, the `database -upload` command is used to upload the specified number of samples, instead of a time window. For example:

  ```
  xeset traceMemSize 128000fclk
  ```

  Note: In SA mode, the `fclk` time unit is not supported; instead, `ns` is used as the default time unit.

Changing the Vision Mode at Run time

A design compiled for FV mode can be changed to DYNP mode at run time without recompiling the design either using the CLI or the GUI. Use the following command at run time to change the vision mode:

```
host -dynp <emulator_host>
```

Switching to the DYNP mode has the following advantages:

- Faster waveform upload performance
- Larger trace depth

© 2021 All Rights Reserved.

DYNP mode has different trace depth than FV mode. The compiler selects a default trace depth for DYNP mode, which is generally larger than the trace depth of FV mode. You can also explicitly specify a trace depth for DYNP at compile time.

Use the following command at compile time to set the trace depth in DYNP mode:

```
compilerOption -add {traceDepthDYNP <value>}
```

For more information on the syntax and values of the `traceDepthDYNP` option refer to *User Data Commands* chapter of the *VXE Command Reference Manual*.

However, switching to DYNP mode does not improve the emulation speed because the step count is not changed. Step count is defined at compile time to accommodate probes required by FV mode.

The vision mode can also be changed when running the design in the xeDebug GUI. Before downloading the design, select the *DYNP* check box adajcent to the *Vision Mode* field on the *Setup* tab to change the mode. For detailed information on the fields available on the *Setup* tab in the SA and ICE modes, refer to Starting a Debug Session in Appendix 42, "Using the xeDebug GUI."

Probing SystemVerilog Design Constructs for SST2 and FSDB Waveforms

You can probe the following complex SystemVerilog design constructs in both SA and ICE modes. After probing, the fields of these constructs can be viewed in the SST2 and FSDB waveforms.

- Unpacked Structures: A field of an unpacked struct can be an unpacked struct, an unpacked struct array, a packed struct, a packed struct array, a vector signal or a scalar signal.
- Packed Structures: A field of a packed struct can be a packed struct, packed struct array, a vector signal, or a scalar signal.
- Multi-Dimensional Arrays (MDAs): The MDAs are shown in the waveform as arrays with elements and can be expanded into sub-arrays upto the scalar bit.
- Unpacked Unions: A field of an unpacked union can be any integral data type or any unpacked type. An unpacked union represents a single storage.

When an instance is given in the `probe -create {<object> ... | <scope_name>}` command, all unpacked structs, packed structs, unpacked unions, and multi-dimensional arrays under the instance are probed. You can also probe these constructs directly by specifying the hierarchical full name of the constructs.

© 2021 All Rights Reserved.

Note: If an MDA or a field of unpacked and packed struct, or unpacked union is synthesized into a memory instance, it cannot be probed and cannot be displayed in the waveform.

The packed and unpacked structs are shown in the waveform with all fields as their members. The packed and unpacked struct arrays are shown in the waveform as arrays with elements as structs which can be expanded into their fields.

If a net is probed and the net is a bit of an unpacked or a packed struct, or a bit of a MDA, the whole unpacked or packed struct or the MDA is probed and the waveform shows the whole construct.

Example: Unpacked Struct with Unpacked Structs and MDAs as fields:

```
typedef struct{
   bit [7:0] a [7:0];
    byte b [7:0];
   shortint c [3:0];
   int d [1:0];
   integer e [1:0];
   longint f [1:0];
   time g [1:0];
} ust1;
typedef struct{
logic [7:0] a;
logic [15:0] b;
} ust2;
typedef struct {
   ust1 a;
   ust1 b[2];
} ust3;
typedef struct {
   ust2 a;
   ust3 b[2:4];
   ust1 c[2:1][0:1];
} ust4;
   ust1 obj1;
   ust1 obj2 [2:0];
   ust1 obj3 [2:0][1:0];
   ust2 obj4;
   ust3 obj5;
   ust3 obj6 [1:0];
   ust3 obj7 [1:0][3:2];
   ust4 obj8;
```

Example: Unpacked Struct with Packed Structs and MDAs as fields:

```
typedef struct packed{
reg [7:0] a;
reg [7:0] b;
} pst;
typedef struct {
shortint w[3][2][7];
pst x;
pst y[2]; } ust;
```


© 2021 All Rights Reserved.

```
ust obj1;
ust obj2[5];
ust obj3[7][6];
```

Using RTL and PDB Names for Probing SystemVerilog Design Constructs

The RTL names of sub-constructs and field bits of unpacked and packed structs, unpacked unions, and MDAs are not supported with the probe command. If the RTL names of sub-constructs or field bits of these constructs are issued in the probe command, the probe command fails with an error message stating that the given name is not found in the current scope. For example, the following commands will error out:

```
probe <construct>.<sub_construct>
probe <construct>.<sub_construct>.<fieldBit>
```

The PDB names are supported with the probe command. When a PDB name is issued with the probe command, the corresponding whole SystemVerilog construct will be probed.

The following example illustrates various valid and invalid probe commands:

```
typedef struct packed {
bit a;
bit b;
} pst;
typedef struct packed {
bit b;
pst p;
} ppst;
ppst p1;
```

- **Case 1- Probing Struct Names**: The packed struct name in the above example is p1. Here, the following command is valid:

  ```
  probe p1;
  ```

- **Case 2 - Probing Field Names**: The field names cannot be probed. Therefore, the following commands with the packed struct field names, p1.b, p1.p.a, and p1.p.b will error out:

  ```
  probe p1.b
  probe p1.p.a
  probe p1.p.b
  ```

- **Case 3: Probing PDB Names**: Each individual field PDB name probing causes the whole construct to be probed. The packed struct fields have PDB names as : \p1.b , \p1.p.a , and \p1.p.b. The following commands are valid.

  ```
  probe {\p1.b }
  probe {\p1.p.a }
  probe {\p1.p.a }
  ```


© 2021 All Rights Reserved.

Note: Cadence recommends using complete struct or MDA names with the `probe` command. Probing RTL names of sub-constructs or field-bits will not produce expected waveforms.

Probing Constructs in Generate Blocks and Packages

In Palladium, the hierarchy of generate blocks is different from Xcelium. However, in the waveforms, the `probe <struct_name>` command follows Xcelium naming format.

The `<struct_name>` in probe `<struct_name>` can be either `genKey[%d].structName`, or `genKey_%d_.structName`, such `gen[1].struct1` or `gen_1_.struct1`.

In the SA mode, with or without the `-rtlNameForGenerate` option, after swapping the design to the hardware, you can probe in either of the above specified Xcelium formats.

However, without the `-rtlNameForGenerate` option before swap-in, only the `probe genKey_%d_.structName` format is supported.

The unpacked structs, packed structs, or MDAs might be in a package. In such cases, the unpacked structs, packed structs, or MDAs are shown under the following hierarcy: `Packages->xc_ncwork.<package>::`

To probe the SystemVerilog constructs under a package, execute the following command:

```
probe <package>::<svConstuctName>
```

For example,

```
package ComplexPkg;
typedef struct packed {
shortint i, r;
} Complex;
Complex c1;
```

To probe the `c1` construct, execute the following command:

```
probe ComplexPkg::c1
```

Probing SystemVerilog Constructs in Compilation Unit

The unpacked structs, packed structs, unpacked unions, and MDAs might exist in the compilation unit.

When probing these design constructs directly, you need to prefix `_unit_::` in the ICE mode, and `unit_0::` in the SA mode after swapping the design to the hardware. For

© 2021 All Rights Reserved.

example, if the design construct is `a`, the `probe` command would be `probe _unit_::a` in ICE mode and `probe _unit_0::a` in the SA mode after swap-in.

In the waveforms, these constructs are shown under the following hierarchies:

- In ICE mode, `Packages->xc_ncwork._unit_::`
- In SA mode, `xc_ncwork._unit_0::`

In the following example, to probe the SystemVerilog constructs in ICE mode, execute the following command: `probe _unit_::a .`

```
`timescale 1ns/1ns
int a[2][3]='{'{1,2,3}, '{4,5,6}};
module test;
reg clk;
int count;
initial
begin
clk=0;
count=0;
end
always @(posedge clk)
begin
count <= count + 1;
for(int i=0;i<2;i++)
b1 : begin
for(int j=0;j<3;j++)
b2 : begin
a[i][j] = a[i][j] + count;
end :b2
end : b1
end
endmodule
```

Using Verilog Escaped Names for SV Constructs

You can use Verilog escaped names in packed and unpacked structs, unpacked unions, and MDAs. Escaped names are supported with the following limitations:

- In both SA and ICE modes, escaped names in packed and unpacked structs, unpacked unions, and MDAs cannot include the % character in complex scenarios.

  For example, escaped name, `\%^&*b()` in the following is not supported:

  ```
  typedef struct {
  int a;
  int b;
  } \|\[]myustr* ;
  typedef struct {
  \|\[]myustr* \!@#$a)( [2:0][1:0];
  \|\[]myustr* \%^&*b() [2:0][1:0][1:0][1:0];
  } \^&*(mycomplexstr) ;
  ```

  However, the % character is supported in simple scenarios like below:

© 2021 All Rights Reserved.

```
typedef struct {
reg \!a$%s ;
reg b;
reg c;
} ust0;
```

- In SA mode, escaped names cannot be used as field names of an unpacked or packed struct and unpacked unions if the field is a multi-dimensional array.

  For example, `\*^&*b()` in the following is not supported in SA mode as the field is a multi-dimensional array.

```
typedef struct {
mylogic a;
myarray b;
} \|\[]myustr* ;
typedef struct {
\|\[]myustr*  \!@#$a)(  [2:0][1:0];
\|\[]myustr*  \*^&*b()  [2:0][1:0][1:0][1:0];
} \^&*(mycomplexstr) ;
```

Viewing SystemVerilog Constructs with DUV Design Browser

In xeDebug GUI, the DUV Design Browser shows PDB names for the whole design hierarchy. All unpacked and packed structs, unpacked unions, and MDA individual field bits are shown in the PDB format.

You can select any signal in the DUV Design Browser to probe and show in the waveform browser. The waveform browser shows RTL names for the SystemVerilog constructs instead of the PDB names.

When an individual construct's field bit is selected to be probed and displayed through the DUV Design Browser, the field bit will be shown in waveform browser and the whole corresponding construct will be probed and be available in the waveform browser.

© 2021 All Rights Reserved.

Figure 40-1 PDB Names in DUV Browser

© 2021
All Rights Reserved.

Waveform Support Set for SystemVerilog Design Constructs

The following table lists waveform support for the SystemVerilog constructs — packed structs, unpacked structs, unpacked unions, and MDAs.

Table 40-5 SystemVerilog Design Constructs Waveform Support

Command / Generate Block / Package	Waveform Display
`probe <instance> -depth <n>`	■ The waveform shows unpacked struct and unpacked struct array (1D, 2D or MD) in Xcelium pattern ■ If an unpacked struct field is a packed struct, the waveform shows structs with field names ■ Packed structs are shown with field names in the waveform. ■ 2D arrays and MDAs are shown in Xcelium pattern.
`probe <unpackedStruct>`	The waveform shows field names, if the field is a packed struct .
`probe <packedStruct>`	The waveform shows field names.
`probe <MDA>`	The waveform shows MDA in Xcelium pattern.
`probe <unpackedUnion>`	The waveform shows field names.
Unpacked struct in package	The waveform shows unpacked structs in `Package-> xc_ncwork.<Package>::`
Packed struct in a package	The waveform shows packed structs in `Package-> xc_ncwork.<Package>::`
Generate block Xcelium pattern hierarchy for unpacked or packed structs	Shows MDAs in the waveforms
`probe <netName>`	The waveform shows the whole unpacked struct, packed struct, unpacked union, or MDA if the `<netName>` is a field or bit of unpacked or packed struct, unpacked union, or MDA.

© 2021 All Rights Reserved.

Supported Unpacked Union Datatypes

This section lists the supported unpacked union types. Currently, the unpacked unions are only supported in ICE mode. The objects of these unpacked union types can be used with various run-time commands as required:

- Unpacked union datatype

  Example:

```
typedef union {reg [7:0] fa;
      reg [7:0] fb [1:0];
      reg fx;
} strT1;
strT1 InststrT1;
```

- Array of unpacked union

  Example:

```
typedef union {reg [7:0] fa;
      reg [7:0] fb [1:0];
      reg fx;
} strT1;
strT1 InststrT1[8];
```

- Nested unpacked union

  Example:

```
typedef union {reg [7:0] fa;
      reg [7:0] fb [1:0];
      reg fx;
      strT1 fy;
} strT2;
strT2 InststrT2;
```

- Array of nested unpacked union

  Example:

```
typedef union {reg [7:0] fa;
      reg [7:0] fb [1:0];
      reg fx;
      strT1 fy;
} strT2;
strT2 InststrT2[8];
```

- Nested array of unpacked union

  Example:

```
typedef union {reg [7:0] fa;
      reg [7:0] fb [1:0];
      reg fx;

      strT1 fy[0:7];
} strT2;
strT2 InststrT2;
```


© 2021 All Rights Reserved.

- Array of nested array of unpacked union

  Example:

```
typedef union {reg [7:0] fa;
      reg [7:0] fb [1:0];
      reg fx;
      strT1 fy[0:7];
} strT2;
strT2 InststrT2[8];
```

The waveform display for unpacked union is done on the basis of the HDL-ICE Compiler netlist information — that is, either from the net information or from the pragma information. If an unpacked union member remains unused, it would be optimized and shown in the waveform as one bit.

Viewing Two-Dimensional Arrays in SimVision

When a two-dimensional array of bit types is probed, the waveform for the array is displayed as one unit in SimVision. For example, if the RTL source code has an array as follows:

```
wire [0:2] farm_light [0:999];
```

Then, the waveform for the array in SimVision is displayed as `farm_light[0:999]`, as shown in figure, where each one-dimensional array element is of three bits.

© 2021 All Rights Reserved.

Figure 40-2 Two-dimensional Arrays

Viewing Modports in SimVision

In SimVision, modports can be seen under interface scope. To view the interfaces, you must start SimVision with the snapshot.

In the following example, modport `MC/MR` appear as `Top.i.MC` and `Top.i.MR`. To view the interface `i` in SimVision Design Browser under `Top.Top.m0, ...., Top.m3, Top.INCR_inst`, you must start SimVision with the snapshot.

For information on how to view modports in Indago, refer to Viewing Modports in Indago. For information on the modport support, refer to Support for Modports section.

© 2021 All Rights Reserved.

Tracing Signals Using InfiniTrace

InfiniTrace allows you to access a virtually unlimited trace window. During the emulation run, data that includes emulation history and status is periodically dumped to the disk by stopping the emulation clocks at regular time intervals. When the emulation run is stopped, the saved data can be restored to any time point in the period where it was saved, and observe the emulation status. Different parts of the run can be replayed, using different SDL trigger conditions to search a specific time point within the saved emulation history.

You must download the design on an emulator to be able to use the InfiniTrace feature. InfiniTrace is available in STB and SA emulation modes, with or without a static target system, using either FV or DYNP probing mode. It is not supported in LA mode. In SA mode, InfiniTrace is supported with both xeDebug and *xrun*. In SA mode, InfiniTrace replays the emulation history of DUT part, but it cannot replay the history of the testbench part. It can replay only the testbench inputs.

InfiniTrace performance is slower for smaller designs, such as, designs with less than two boards. Additionally. the InfiniTrace mode does not support the following:

© 2021 All Rights Reserved.

- Wall clock time - if a number does not follow emulation time, it is assumed to be in emulation cycles.
- Dual clock domain, that is — designs having both controlled and uncontrolled clocks.

There is no additional gate capacity overhead associated with the use of InfiniTrace. It is enabled by default. It can be enabled or disabled at compile time by using the following command:

```
compilerOption -add {infiniTrace on | off}
```

Cadence recommends to keep the InfiniTrace feature enabled. In rare cases, where designs have large numbers of primary inputs, you can disable it to decrease the overhead.

InfiniTrace Sessions

The InfiniTrace mode involves the *Prepare* and *Observe* sessions. Additionally, designs can be run in the non-InfiniTrace mode, which is termed as the *normal* run. Following is a brief description of these types of sessions:

- ***Prepare*** session:

  In the *Prepare* session, the system collects data while the emulation is in progress. A special memory buffer records the data necessary to record the history of emulation. Emulation run is periodically paused to upload data from the memory buffers to the workstation's disk once the buffer is full. This can slow down the effective emulation speed. The target also pauses during the data upload. Except for the slow down, there is no difference between the normal run (without InfiniTrace) and the *Prepare* session. All operations that can be performed during a normal run can be performed during a *Prepare* session. The *Prepare* session data is saved in the `<session_name>.infiSession` sub-directory.

- ***Observe*** session:

  In the *Observe* session, the emulation status is restored to a specific time point by using the data collected during the *Prepare* session. In an *Observe* session, you can only perform operations that *observe* the simulation acceleration or emulation status, such as dumping memory content, getting values of nets, defining probes, and uploading waveforms and so on. You cannot perform operations that change emulation status and thus alter the history of the emulation, such as writing of memories, setting, forcing, or releasing nets, or running emulation. During *Observe* sessions, the output from the emulator to the target is frozen, so that the operations on emulator do not affect the target. In SA mode, only the DUT is observed, and the testbench running on the software simulator is frozen.

© 2021 All Rights Reserved.

To use InfiniTrace, you must first create a *Prepare* session to record the emulation history. Each session is identified by a specified name. The data for the session is saved in a subdirectory that derives its name from the session name. Then, in the *Observe* session you can replay the emulation history recorded in the *Prepare* session.

Using InifiniTrace in ICE Mode

InfiniTrace can be used in the following mode:

- Hardware-Mode InfiniTrace (HWIT)

Hardware-Mode InfiniTrace (HWIT)

In HWIT mode, the emulator hardware is required for replaying the emulation history during the *Observe* session.

With HWIT, you can enter a temporary *Observe* session from the *Prepare* session, and replay a part of the recorded history while in Prepare session. End the temporary *Observe* session and switch back to *Prepare* session by using the `infiniTrace -goto end` command.

Using infiniTrace Command Options in HWIT Mode

This section lists the commonly used `infiniTrace` command options. For the corresponding steps when using the xeDebug GUI, refer to InfiniTrace Tab on page 1951.

1. Start a *Prepare* session and assign a name for this session by using the following command:

   ```
   infiniTrace -prepare <session_name>
   ```

   Note: This command cannot be used in an *Observe* session.

2. Enable or disable the data capture by using the following command:

   ```
   infiniTrace -on | -off
   ```

   Data capture is enabled by default, after you start and name a *Prepare* session.

3. Remove the specified data set, including the directory and its contents by using this command:

   ```
   infiniTrace -rm <name>
   ```

   The session must be closed before issuing this command.

4. Open an *Observe* session by using the following command:

© 2021 All Rights Reserved.

```
infiniTrace -observe <name>
```

If a session is already open, it is closed.

5. Use the following command to specify the number of cycles to be uploaded:

```
xeset traceMemSize <size>
```

The `traceMemSize` value can be infinite, and it is not limited by maximum trace depth. You can specify any value for it. The `traceMemSize` run-time parameter must be smaller than the maximum trace depth. Trace depth is specified at compile time. Larger trace depths require more hardware resources. For InfiniTrace, there is no limit for trace depth.

6. Restore simulation acceleration status to the specified time point by using the following command:

```
infiniTrace -goto <time_specification>
```

In this command, the time is specified in the following format:

- `t` - An absolute time specification, where `t` specifies the time
- `+t` - Advances from the current time, where t specifies the time
- `-t` - Decrements from the current time, where t specifies the time
- `trigger` - Searches forward from the current time until a trigger point is encountered or reaches the current *Observe* session. The trigger condition is defined the same way as it is defined in normal run.
- `begin` - Goes to the first cycle
- `end` - Goes to the end cycle

7. Specify the time interval between automatic capture points by using the following command:

```
infiniTrace -stateSaveFrequency <value>
```

This command only applies to HWIT.

8. Display information about the automatic saves during the *Prepare* session by using the following command:

```
infiniTrace -getStateSavePoints
```

9. Close a session by using the following command:

```
infiniTrace -close
```

10. List the name of all available data sets that can be used for *Observe* sessions by using the following command:

```
infiniTrace -list
```


© 2021 All Rights Reserved.

11. List the time intervals where data capture is enabled in the current session by using the following command:

```
infiniTrace -intervals
```

This command can be used in either a *Prepare* session or an *Observe* session.

Using InfiniTrace in SA Mode Using xeDebug

InfiniTrace helps to save the state of the design during the run, so that, this state can be restored from any saved time points later to observe the simulation acceleration status. In SA mode, InfiniTrace only saves/restores the DUT part of the design. After restoring the state, you can dump out memory content, add probes and upload waveform of any net in the DUT.

In the *Prepare* session, the system collects data for DUT part of the design which is running in SA mode.

In the *Observe* session, the DUT can be restored at any specified time point by applying the data collected during *Prepare* session.

The testbench is not replayed in *Observe* session. An error is reported if you try to access the testbench while in *Observe* session.

During *Observe* session, the simulator is not used and the following command sets are disabled:

- `xc` commands
- Xcelium commands
- Commands related to testbench. For example, memory read, value, probe on nets in testbench.

To exit the *Observe* session, you must issue the `goto end` command so that the DUT and testbench are again synchronized with each other.

Using InfiniTrace in SA Mode Using xrun

This section describes support for the InfiniTrace feature with *xrun* at run time. The InfiniTrace feature functions quite similarly to the manner in which it functions with the xeDebug tool as described above, except for the following differences:

- When you use *xrun*, you cannot use the *Observe* session; only the *Prepare* session is available.

© 2021 All Rights Reserved.

When you are using *xrun*, the *Prepare* session is automatically closed if you issue any of the commands:

- `reset`
- `restart`

- You must swap in to the hardware run-time mode before opening the *Prepare* session when using the *xrun* tool at run time. For example:

```
xc on -run
xc infiniTrace -prepare <session_name>
```

When you swap from the hardware to the software run-time mode, the InfiniTrace feature is automatically disabled. If any *Prepare* session is open, it is closed before the swap to the software mode.

InfiniTrace Commands Supported with xrun

All `infiniTrace` commands EXCEPT for the following are supported when you use *xrun*:

- `xc infiniTrace -observe`
- `xc infiniTrace -goto`
- `xc infiniTrace -bookmark -goto`

Specifically, the following *Prepare* session commands are supported when you use *xrun*:

```
xc infiniTrace -prepare <session_name> [-overwrite]
xc infiniTrace -on | -off
xc infiniTrace -close
xc infiniTrace -list
xc infiniTrace -intervals
xc infiniTrace -bookmark -add | -rm <bookmark_name>
xc infiniTrace -bookmark -list
xc infiniTrace -isCaptureEnabled
xc infiniTrace -getCurrentMode [-text]
xc infiniTrace -getSessionName
```

Like all SA run-time commands that you use with *xrun*, the commands specific to the InfiniTrace feature must be executed with the `xc` prefix.

To observe the infiniTrace session, you must exit *xrun*, and start the xeDebug tool. Then, download the debug database, and open the *Prepare* session that you had created while running *xrun*.

Comparing the InfiniTrace Feature in SA and ICE Modes

The InfiniTrace feature functions with the following differences in the SA and ICE modes:

© 2021 All Rights Reserved.

- In SA mode, the time specification is in the unit of simulation time when using the `infiniTrace -goto <time>` command. In ICE mode, the time is specified in FCLK cycles.
- When specifying trace window size, the unit is simulation time in SA mode. You can specify any trace window size as long as the trace window is within the range where InfiniTrace is turned on during *Prepare* session.
- In SA mode, the `infiniTrace -goto <xtime>` command cannot be used to go to the exactly specified time. The data is replayed only for the simulation times saved during emulation. The replay can be for the simulation time less than or equal to the `<xtime>` specified.

  For example, if time steps saved in *Prepare* session are `5ns`, `10ns`, `15ns`

  ```
  infiniTrace -goto 7ns will stop at 5ns
  infiniTrace -goto 10ns will stop at 10ns
  infiniTrace -goto 11ns will stop at 10ns
  ```

All other operations are the same as in ICE mode.

Note: In SA mode, the simulation runs slower because of the need to save design snapshot and simulator inputs.

For more information on the commands used in InfiniTrace and its modes, refer to the description of the `infiniTrace` command and its options in the *Run-Time Commands chapter* of the *VXE Command Reference Manual*.

Bookmarking InfiniTrace Session

You can add bookmarks in the InfiniTrace session to mark specific time points. Then, in the *Observe* session, you can quickly reach those time points using the ~~the~~ predefined bookmarks.

Following are the common `infiniTrace` commands used for bookmarking:

1. Define a bookmark using the following command:

   ```
   infiniTrace -bookmark -add <book_mark_name>
   ```

   The current cycle is defined as a bookmark with specified name. Any text string can be used as the bookmark name.

2. Use the following command to remove a bookmark:

   ```
   infiniTrace -bookmark -rm <book_mark_name>
   ```

3. List the defined bookmarks using the following command:

   ```
   infiniTrace -bookmark -list
   ```


© 2021 All Rights Reserved.

The bookmark names and the corresponding cycle numbers are displayed.

4. Use the following command to go to a specific bookmark:

```
infiniTrace -bookmark -goto <book_mark_name>
```

A bookmark can be specified for both *Prepare* session and *Observe* session.

Tracing and Uploading Waveforms

Palladium Z1 supports waveforms of SST2 (also known as SHM) and FSDB formats. By default, waveforms are generated in SST2 format.

The traced waveform database can be viewed by uploading it to the appropriate waveform viewer that is selected based on the waveform format. The following sections cover information about tracing waveforms in different modes using either xeDebug or xrun:

- Tracing and Uploading Waveforms in ICE Mode Using xeDebug Tool
- Tracing and Uploading Waveforms in SA Mode Using xeDebug Tool
- Tracing and Uploading Waveforms in SA Mode Using xrun

Minimum Run Cycles and Intervals before Uploads

Before uploading the debug data, the design must be run for at least one cycle for the upload to be successful. Also, a minimum interval of one cycle should be observed before doing the next upload of the debug data. The data is available from the 0th cycle until one cycle before the time of uploading the data.

Tracing and Uploading Waveforms in ICE Mode Using xeDebug Tool

Following is an example of a command sequence to generate and view waveforms in ICE mode:

1. Set the waveform format at the time of invoking the xeDebug tool as follows:

```
xeDebug [-shm | -fsdb | -vcd]
```

   If you do not specify the waveform format while executing the `xeDebug` command, by default the format is set to SHM.

2. Create the waveform database to store the waveform output files by using the following command:

```
database -open <database_name>
```


© 2021 All Rights Reserved.

In the Vector Debug mode, this command is not required because the name of the waveform file is automatically set according to the stimulus vector filename. The vector name is set using the `vector` command.

3. Use XEL command `xeset`, to set the `traceMemSize` run time parameter to determine the number of samples for upload from the trace buffer into a file in SA, Logic Analyzer (LA), and STB modes. Setting this run time parameter affects all subsequent `run` and `database -upload` commands.

4. Add or delete probes during the run time using the `probe` XEL command. To create new probes, execute the following command:

   ```
   probe -create <probe_name>
   ```

   For detailed information on various options of the *probe* command, refer to the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

5. Run the design by executing the `run` XEL command.

   For example:

   ```
   run 1000
   ```

6. Upload the trace results to a trace file of the specified file format using the `database -upload` XEL command. If there is no data for upload, gaps are inserted into the waveform database. For example, if you run the following commands when the design was compiled for DYNP mode:

   ```
   probe -create netA
   run 1000
   database -upload
   probe -create netB
   run 1000
   database -upload
   ```

   You will see the waveform of netA from cycle 1 to 1998 (the last 2 cycles cannot be uploaded). But for netB, you will see the waveform for only cycles 1001 to 1998. The first half of the waveform will be blank.

   After using the `database -upload` command, you can use the `probe -waveform` command to add the objects within a probe to the waveform display.

   If you are using the xeDebug GUI instead of command-line interface, do either of the following ways to upload trace results:

   - Click the *Upload* button on the *Probe* tab in xeDebug GUI.
   - Turn on the *Auto Upload* option on the *Control* tab to automatically trace data, upload it and display the waveforms after each run. *Auto Upload* is set to *off* by default.

© 2021 All Rights Reserved.

7. View the waveforms in the waveform viewer depending upon the selected waveform file format, that is, SST2 or FSDB.

By default, the waveforms are generated in the SST2 or SHM format. Such waveforms can be opened in SimVision by using the following set of commands:

From the UNIX/Linux shell:

```
%         simvision -waves <sst2_database_name> &
%         simvision -waves &
```

From XEL command-line when running under the xeDebug script mode:

```
XE>       exec simvision -waves <sst2_database_name> &
XE>       exec simvision -waves &
```

If *<sst2_database_name>* is not specified on the command line, the SST2 format waveforms can be opened from the SimVision GUI. For more information on SimVision, refer to the Xcelium documentation set.

Note: In ICE mode, if the data is written from a non-zero time, the time fold (gap) between the time 0 and the starting non-zero time is not available. However, SimVision will show the waveform starting from the non-zero time.

For SHM waveform, the constant nets have values for the entire time window in the waveform.

For FSDB format waveforms, you need to use the Novas waveform tools, such as Siloti and Verdi. To open the waveforms in Verdi waveform tool installed on your machine, use the following set of commands:

From the UNIX/Linux shell:

```
%         verdi -nWave -showtrigger -ssf <fsdb_database_name> &
%         verdi -nWave -showtrigger &
```

From XEL command-line when running under the xeDebug script mode:

```
XE>       exec verdi -nWave -showtrigger -ssf <fsdb_database_name> &
XE>       exec verdi -nWave -showtrigger &
```

If *<fsdb_database_name>* is not specified on the command line, the FSDB format waveforms can be opened from the GUI. For more information on using Novas tools to view waveforms of FSDB format, refer to Chapter 47, “Using Novas Tools.”

When running xeDebug GUI mode, the waveform can be displayed by clicking the waveform icon at the top of the GUI. The waveform can also be opened automatically if the *Auto Wave* option in the *Control* tab is turned on.

Note: If you zoom Novas nWave to 100% while the xeDebug is either generating the waveform data or sending the commands to Novas nWave, you might see the nWave waveform displayed incorrectly.

© 2021 All Rights Reserved.

The xeDebug tool supports the VCD format. For VCD databases, probes cannot be incrementally added once transitions are written in a VCD file and it has no live waveform display. Also, you must load trace data into waveform window manually and this requires VCD to SST2 or FSDB translation on every database reload request. VCD mode is preferred for offline debugging.

To use VCD format with xeDebug tool specify the `-vcd` option with the `xeDebug` command:

```
xeDebug -vcd
```

Tracing and Uploading Waveforms in SA Mode Using xeDebug Tool

Following is an example of a command sequence to generate waveforms with xeDebug:

1. Set the waveform format at the time of invoking the xeDebug tool from the UNIX command prompt. Use the following command:

   ```
   xeDebug [-shm | -fsdb | -vcd]
   ```

   If you do not specify the waveform format while executing the `xeDebug` command, by default the format is set to SHM.

2. Create the waveform database to store the waveform output files by using the following command:

   ```
   database -open <database_name>
   ```

3. Create a probe during run time using the following command:

   ```
   probe -create cpu_test.RISC.control -all -depth all
   ```

4. Run the design in the software simulator using the following command:

   ```
   run 100ns
   ```

5. Swap the DUT to hardware and run using both the simulator and emulator:

   ```
   xc on -xt0 -zt0 -run
   run 100ns
   ```

6. Add probes to the DUT and run more with new probes:

   ```
   probe -create cpu_test.RISC.registers
   run 300ns
   ```

7. Upload the trace results from both the simulator and the emulator and create a waveform database using the following command:

   ```
   database -upload
   ```


© 2021 All Rights Reserved.

Generating Continuous Waveform in SA Mode

The `database -open -continuousupload` command starts continuous upload of a waveform database. When the DCC is full, the `-continuousupload` option will pause the design run to upload the waveform. You can use the `database -close` command to stop the continuous upload and close the database. In SA mode, you can start continuous upload of a waveform database when the design is either swapped in to the hardware or swapped out to the software. If you do swap-in and swap-out multiple times, a continuous and merged waveform is generated for the time window.

After continuous upload is enabled, the continuous upload of a waveform database is done automatically when you execute the `run` command. You do not need use the `database -upload` and `database -prepareoffline` commands.

The `database -open -continuousupload` command can be used as an alternative to InfiniTrace. Although it creates larger files once the waveform is uploaded, it does not require a second observe session.

Continuous upload is only supported in modes that allow design clocks to be stopped, such as STB or IXCOM single-domain designs with only controlled clocks and static target mode, Infinitrace (both ICE and SA modes), and VD. Modes not supported with continuous upload include the following:

- LA
- SA dual-domain with uncontrolled clocks
- SA in Dynamic Target mode

Note: The `-continuousupload` option can also be used in ICE mode as an alternative either to a TCL loop that uploads the DCC when full or to InfiniTrace.

Accessing SST2 Format Waveform Databases in SA Mode

In the online session for SA mode, the waveform data is stored into a pair of linked databases. One database contains the data from the simulator and the files, by default, are named as `trace.dsn` and `trace.trn`. Another database contains the data from the emulator and the files are named as `trace_hw.dsn` and `trace_hw.trn`. Note the suffix `_hw` in the names of the emulator files. Both the databases are located in the same waveform database directory named, by default, `trace.shm`, as shown in the directory structure below:

```
trace.shm/
    trace.dsn
    trace.trn
    trace_hw.dsn
    trace_hw.trn
```


© 2021 All Rights Reserved.

When you generate a new hardware database, xeDebug creates a new SHM directory, and links to the online software database. So, the new directory includes the online software data and the new hardware data.

The structure of the new directory is as shown below:

```
trace2.shm/
    trace.dsn -- hard link --> ../trace.shm/trace.dsn
    trace.trn -- hard link --> ../trace.shm/trace.trn
    trace2_hw.dsn
    trace2_hw.trn
```

Any previous files in the new database directory are moved into a subdirectory named `bak`. The hard links enable the waveforms of the software and hardware data to be merged or integrated. See Viewing Integrated SST2 Format Waveforms in SA Mode in Offline Session for information about the merged waveforms functionality.

To ensure that the online and offline waveform databases and trace files match, during the online session, xeDebug saves signatures in the files.

During an offline session with xeDebug tool, if you additionally open a hardware database generated in an online session, it is shown as a separate hierarchy.

During offline debugging and InfiniTrace observe sessions, the SST2 waveform database created in online session can only be viewed and not modified.

When the SimVision tool is invoked by a command like `simvision -snapshot tbench trace.shm` or from the xeDebug tool, the snapshot and the database files of both types (software and hardware) are shown under a single hierarchy, and the Waveform Viewer displays both the waveforms merged. You need to specify the snapshot name to view the merged waveforms.

Viewing Integrated SST2 Format Waveforms in SA Mode in Offline Session

In SA mode, use the SimVision waveform viewer to view SST2 format waveform databases to get an integrated view of the design objects of both software and hardware portions. You can also use the `xeDebug -offline` option to start an offline session.

This helps you to view, at the same time, the signal values written by the simulator and the emulator, and get a comprehensive look into the complete design.

Essentially, the following two features are offered through this integrated view:

- Access to testbench data in an offline session, which was not available in previous releases

© 2021 All Rights Reserved.

- Merged waveform functionality in an offline session, which means that the data from the simulator and emulator is merged into one line

  Note: The merged waveform functionality is not supported in an InfiniTrace *Observe* session in SA mode.

In the SimVision waveform window, you can differentiate the hardware data from software data by its cross-hatched background. When the design is in swap-in state and the signals are from DUT, then the background will be crosshatched. Refer to Figure 40-3 for a sample SimVision waveform window.

Note: During data collection on the emulator, the crosshatched areas might not have any data. This means that the design ran for a time more than the specified trace window size.

Figure 40-3 SimVision Window with Software and Hardware Data Integrated View

Steps:

1. Start an offline session of xeDebug and specify the *.phy subdirectory using the following command:

© 2021 All Rights Reserved.

```
xeDebug -offline <session_name> <phyfile_name>
```

For example:

```
xeDebug -offline . trace.phy
```

The dot (`.`) in the above command indicates to use the trace.phy subdirectory from the current design directory while opening the xeDebug GUI window (refer to Figure 40-4).

Figure 40-4 xeDebug GUI Window - Offline Session

2. Click the SimVision Waveform icon ( ) from the toolbar at the top of the xeDebug GUI window. The SimVision browser illustrated in Figure 40-5 is displayed.

© 2021 All Rights Reserved.

Figure 40-5 SimVision Browser

When this window opens, at first, the name of the file is displayed as *trace* (refer to red highlighted in Figure 40-5).

3. Click the *Probe and Display* button ( ) from the toolbar at the top of the SimVision browser. The *Design Browser & Probe Upload* window illustrated in Figure 40-6 is displayed.

© 2021 All Rights Reserved.

Figure 40-6 Design Browser & Probe Upload Window

4. From the design hierarchy, select the required design file. For example, select `xcva_top.xc_top.dum2` as shown in Figure 40-7.

© 2021 All Rights Reserved.

Figure 40-7 Design Browser & Probe Upload Window - Design Hierarchy and Nets

5. Specify the *Filter* criteria in the *Nets* group box and click the button. The nets that match the specified criteria are displayed within the group box.

6. Click the *Probe* button in the *Probing and Tracing* group box at the bottom of the *Design Browser & Probe Upload* window. After you add the required probes, the database name displayed in the SimVision browser changes from *trace* to *trace_offline* as shown in Figure 40-8. This database provides an integrated view of the design, which

© 2021 All Rights Reserved.

means that there is a single design hierarchy (`cpu_test`) displayed for the testbench and all the design components appear under it. Unlike the previous releases, you will not see two separate blocks for the testbench in which only one block displayed the probes.

Figure 40-8 SimVision Browser with Software and Hardware Data Captured during Online Session

After the database name changes to *trace_offline*, you can add more probes in both the testbench and DUT.

Note: Executing the `database -upload` command is necessary after adding DUT probes. The values for the DUT probes are available for both the runs on the software simulator and the emulator. However, the values for the run on the software simulator are available only if you had probed the net during the online session.

Creating the probe by using the following command displays the values during the offline session on the software simulator (with no grey dots in background) if the probes were added before swapping into the emulator during the online session:

```
probe -add cpu_test.RISC.control_inst -all
```

However, if you did not add this probe during the online session, you can see only the values from the hardware run during the offline session.

© 2021 All Rights Reserved.

In the offline session, while using the SimVision browser invoked from the xeDebug GUI, you need to click the *Send-to Waveform* button to get the value from the run on the software simulator. Refer to the figure given below.

To get the values from the run on the hardware, you must use the *Probe and Display* button followed by the *Upload DUV Probes* button.

Alternatively, use the buttons available on the *Probe* tab of the xeDebug window to view the values from the hardware run. For more information, refer to Probe Tab of Chapter 42, "Using the xeDebug GUI."

By using the options in the SimVision browser for adding a signal to the waveform, such as the *Click and add to waveform area* check box at the bottom of the SimVision *Design Browser*, you can get only the values from the run on the software simulator.

© 2021
All Rights Reserved.

Tracing and Uploading Waveforms in SA Mode Using xrun

Following is an example of a command sequence to generate waveforms with *xrun* using interactive command-line input mode:

1. Invoke *xrun* from the UNIX/Linux command shell in command-line mode:

   ```
   xrun -R -64bit -tcl
   ```

2. Add simulation probes to be used only by the software simulator using the following command:

   ```
   probe -add -shm cpu_test.RISC.control -depth all
   ```

3. Run the design using the following command:

   ```
   run 100ns
   ```

4. Swap the DUT portion of the design to hardware and run the simulation, using the following commands:

   ```
   xc on -xt0 -zt0 -run
   run 1ns
   ```

5. Set the name and format of the DUT waveform database using the following command:

   ```
   xc sigTrace -outfile -sst2 wave1
   ```

6. Add DUT emulation probes using the following command:

   ```
   xc sigTrace -addinst -depth 0 cpu_test.RISC.control
   ```

7. Run the design using the following command:

   ```
   run 100ns
   ```

8. Upload trace results from the emulator into a trace file using the following command:

   ```
   xc sigTrace -upload
   ```

Generating Waveforms in Batch Mode with Partitioned FullVision

Waveform generation performance (that is, the speed and memory footprint) can be improved in the following two ways:

- Generating multiple waveform files (FSDB or SHM) in parallel.

  For details on how to generate multiple FSDB/SHM files, refer to Generating Waveforms in Offline Batch Mode.

- Performing FV partitioning at compile time and doing fvCompute on those partitions. This operation may let fvCompute jobs consume less than 256 GB memory footprint. During

© 2021 All Rights Reserved.

compilation, a design is partitioned, and all partitions are compiled in parallel. Multiple FV side files (`swfvDB` files) are generated based on the number of compiler partitions. One swfvDB file is generated for each partition. There is no impact on the generation of the `*phy` file. The `*phy` file is generated for the full design.

All xeDebug offline session can be invoked parallel, which results in faster throughput. Cadence recommends that you use `SWFV_FREEZE_PHY` environment variable. This variable keeps the `.phy` file content intact. Without this option, the `.phy` file will be updated by adding more transition data into it. Currently, the `.phy` file does not support multiple read/write operations in parallel. For detailed information on how to perform FV partitioning, refer to Enabling FV Partitioning for PPC Designs.

In xeDebug offline sessions, you can generate multiple SHM/FSDB files for each partition. The SHM/FSDB files are generated simultaneously, for each partition. Multiple xeDebug offline sessions are launched in parallel to generate SHM/FSDB files for all the partitions. There exists one xeDebug offline session per partition.

This section explains how to generate multiple FSDB/SHM files with each `swfvDB` file. This operation is supported in both SA and ICE mode.

Note: It is possible that one partition has partial fields, elements, or bits, and another partition has whole construct, MDA, or vector. In this case, whole construct, MDA, and vector can be seen in the waveform using the `shm_merge` and `fsdb_merge` utilities. For details, refer to the Using shm_merge Utility and Using fsdb_merge Utility sections.

Figure 40-9 Multiple Waveform Files with Partitioned Software FV

© 2021
All Rights Reserved.

To generate multiple waveform files with partitioned software FV, follow the steps given below:

1. Enable FV partitioning at compile time. See Enabling FV Partitioning.
2. Load `swfvDB` files in xeDebug offline sessions. See Loading swfvDB Files in xeDebug Offline Sessions.
3. Generate parallel waveform files in offline sessions. See Generating Parallel FSDB and SHM Files in xeDebug Offline Sessions.
4. Process cross-partition vectors, SV constructs, or MDAs, and merge all waveform files using the `fsdb_merge` and `shm_merge` utilities. See Processing Cross-Partition Nets for Waveform Display, Using shm_merge Utility and Using fsdb_merge Utility sections.

Enabling FV Partitioning

You need to enable FV partitioning to generate multiple `swfvDB` files. Generation of `swfvDB` files is supported for PPC compiled designs. You can enable the FV partitioning on PPC designs using the following command in your compile script:

```
compilerOption -add {fvPartition on}
```

The `swfvDB` files can be found in the design directory under `dbFiles/`. For example, `swfvDB`, `swfvDB_0`, `swfvDB_1`, `swfvDB_2`, and so on.

For details, refer to Enabling FV Partitioning for PPC Designs section.

Loading swfvDB Files in xeDebug Offline Sessions

Specify the partition ID using the `-partition` option of the `debug` command to load the swfvDB file for the partition, as follows:

```
debug . -session <sessionID> -partition <partitionID>
```

`<partitionID>` is an integer and should be less than the number of partitions created by the Parallel Partition Compiler (PPC). For example, if the total number of partitions created by the PPC are 3, the partition ID can be either 0, 1, or 2.

With the `-partition <partitionID>` option, a session directory called (`partition`| `<sessionID>`)`_<partitionID>`.`session` is added. The `-session` and `-partition` options specify the session and partition names. With these options, the session will be `<sessionID>_<partitionID>`.`session`.

If you do not specify a session ID, the session directory will be created as follows for the specified partition: `partition_<partitionID>`.`session`.

© 2021 All Rights Reserved.

Generating Parallel FSDB and SHM Files in xeDebug Offline Sessions

Parallel FSDB/SHM file generation is supported only in offline mode. To enable parallel FSDB/SHM generation, create the database using the `-batch` option. You can specify the number of FSDB files using the `-numFiles` option.

```
database <file_name> -batch -numFile <num>
```

For details, refer to Generating Waveforms in Batch Mode section.

The FSDB/SHM writer creates the number of FSDB/SHM files on the basis of the design probes. The number of files in each partition might be different from the specified number in the `database` command. In case of multiple SHM generation, both `trn` and `dsn` files are present under the session directory.

The naming for the FSDB/SHM files is done as follows:

```
(partition|<sessionID>)_<partitionID>.session/<filename>.fsdb/
<filename><postfix>.fsdb
partition|<sessionID>)_<partitionID>.session/<filename>.SHM/
<filename><postfix>.trn
partition|<sessionID>)_<partitionID>.session/<filename>.SHM/
<filename><postfix>.dsn
```

To generate multiple FSDB/SHM files with each `swfvDB` file, generate the `*.phy` file in an *online* xeDebug session.

For each `swfvDB` file, one xeDebug offline session is started. For <*N*> `swfvDB` files, <*N*> xeDebug offline sessions can be invoked in parallel. All offline sessions use the same `*.phy` file.

Example

You have four `swfvDB` files, such as `.dbFiles/swfvDB_0, swfvDB_1, swfvDB_2,` and `swfvDB_3.`

In an online session, you generate a `*.phy` file called `trace.phy`. If, in the offline session, you need to generate five FSDB or SHM files for each partition, create a script called `offline_<partitionID>.xel`, with the following content:

```
debug  . -session <sessionID> -partition <partitionID>
host -offline  ../ <.phy filename>
database <filename> -batch -numFiles <num>
probe <inst> -depth <n>
…
probe <inst> -depth <m>
database -upload
```

where:

© 2021 All Rights Reserved.

- <*paritition*ID> can be either 0, 1, 2, or 3.
- <.*phy filename*> is `trace.phy`
- <*num*> is 5

When *partitionID* >=0, `numFiles` option <*num*> and probe commands are honored, but only nets in corresponding `swfvDB` are probed.

With the `-partition` <*partitionID*> option, a session directory is created with the following naming convention:

(`partition`|<*sessionID*>)_<*partitionID*>.`session`.

With the `-session` and the `-partition` options, the session name is <*sessionID*>_<*partitionID*>.`session`. Without the `-session` option, the session directory is created as `partition`_<*partitionID*>.`session`.

You must use the same file name for the `database` command across all xeDebug sessions for all partitions, and the same `*phy` file name for the `host -offline` command.

All xeDebug offline sessions for the corresponding partition can be launched in parallel.

Processing Cross-Partition Nets for Waveform Display

If all fields of a construct, all elements of a multiple dimensional array, and all bits of a vector are in the same partition, the construct, MDA and vector will be shown in the waveform as construct, MDA and vector.

If only partial fields of a construct, partial elements of a MDA, partial bits of a vector are in one partition, no SV construct, MDA, or vector will be shown in waveform in construct, MDA or vector format. Instead, the cross-partition SV constructs/MDAs/vectors can be placed in an additional SHM or FSDB waveform file automatically with full software database (swfvDB). To do this, use the `shm_merge` and `fsdb_merge` utilities. Cross-partition constructs, vectors, and MDAs are reported under <*partition*>.`session/tmp/crossPartition.list` file. This file is not generated under any partition if splitting of signals across partitions does not happen.

It is also possible that one partition has partial fields, elements, or bits, and another partition has whole construct, MDA, or vector. In this case, you will see the whole construct, MDA, or vector in the partition waveform file.

© 2021 All Rights Reserved.

Using shm_merge Utility

You can reconstruct the split signals to view correctly in the waveform and also view the waveform from different partitions as one master waveform using the `shm_merge` utility. This master waveform is identical to the waveform generated without partitioned waveform generation. The `shm_merge` utility performs the following operations:

1. Generates an SHM file that has all crossing partitions vectors, MDAs, and SV constructs.
2. Generates an SHM name list file that includes all SHM file names pertaining to the same `probe` commands. The *.`list` file also shows the SHM file that has cross-partition nets.

For information on how the crossing constructs, vectors, and MDAs in different partitions are processed for waveform display, refer to the Processing Cross-Partition Nets for Waveform Display section.

The syntax of the `shm_merge` utility is as follows:

```
shm_merge [-session <sessionID>] <filename> [<phyFile>.phy]
```

- `-session` <*sessionID*>: Specifies the session ID. The <*sessionID*> is the same as specified with the following command:

  ```
  debug <path> -session <sessionID> -partition <partitionID>
  ```

  Note: The `-session` option is optional. If session is not specified, the partition is used as the default session ID.

- <*filename*>.`shm`: Specifies the SHM file name. The <*filename*> is the same as specified with the following command:

  ```
  database <filename>
  ```

  Note: Specifying SHM file name is mandatory.

- <*phyFile*>.`phy`: Specifies the `.phy` file which is generated during online session.

The name list SHM file will be created with naming convention as:
[<*sessionID*>.]<*filename*>.`shm.list`.

All SHM files will be listed in the name list file in the following pattern:

(<*sessionID*>|partition)_[<*partitionID*>].`session`/<*filename*>.`SHM`/<*filename*><*postfix*>.`SHM`. [<*sessionID*>.]<*filename*>.

The name list file also includes the SHM file with crossing partition signals. All SHM files including the crossing partition SHM files can be viewed as follows:

1. Get the snapshot name by running the following command:

   ```
   xmls -snap
   ```


© 2021 All Rights Reserved.

For example, the command returns the snapshot `xc_ncwork.dut::module (SSS)`.

The snapshot name is also printed out at the end of `xmelab` under `xc_work/xmelab.log`.

2. Open all SHM files with the snapshot name, as follows:

```
simvision `cat <sessionID>.<filename>.shm.list` -snap <snapshot>
```

For example:

```
simvision `cat trace_pswfvDB.shm.list` -snap xc_ncwork.dut:module
```

For more information on the `shm_merge` utility, refer to the *Programs and Utilities* chapter of the *VXE Command Reference Manual*.

Using fsdb_merge Utility

A virtual FSDB file is used to open multiple FSDB files simultaneously. After all multiple FSDB files are generated with all partitions, use the `fsdb_merge` utility that performs the following operations:

1. Generates an FSDB file that has cross-partition constructs, MDAs and vectors, if needed.
2. Generates a virtual FSDB file that has names of all FSDB files in all partitions, and name of the *make-up* FSDB file that has crossing partitions constructs, MDAs, and vectors.

By opening the virtual FSDB file, signals in all FSDB files can be seen in the waveform window.

For information on how the crossing constructs, vectors, and MDAs in different partitions are processed for waveform display, refer to the Processing Cross-Partition Nets for Waveform Display section.

The syntax for the `fsdb_merge` utility is as follows:

```
fsdb_merge [-session <sessionID>] <filename>.fsdb [<phyFile>.phy]
```

- `-session` <*`sessionID`*>: Specifies the session ID. The <*`sessionID`*> is the same as specified with the following command:

  ```
  debug <path> -session <sessionID> -partition <partitionID>
  ```

  Note: The `-session` option is optional. If session is not specified, the partition is used as the default session ID.

- <*`filename`*>`.fsdb`: Specifies the FSDB file name. The <*`filename`*> is the same as specified with the following command:

© 2021 All Rights Reserved.

```
database <filename>
```

Note: Specifying FSDB file name is mandatory and the `.fsdb` extension must be appended to <*filename*>.

- <*phyFile*>.phy: Specifies the .phy file name. This parameter is optional. This option is used the generate the *make-up* file for crossing partition constructs, MDAs, and vectors. <*phyFile*>.phy is the same as specified with the following command:

  ```
  host -offline <phyFile>.phy
  ```

  If you do not specify the phy file, the make-up FSDB file will not be generated.

The naming pattern for the virtual file that is created on executing the `fsdb_merge` utility is as follows:

```
[session_ID.]<filename>.fsdb.vf.
```

The virtual FSDB file includes all FSDB files generated for all partitions, and the make-up FSDB file for crossing partitions SV constructs, MDAs, and vectors.

The virtual file includes all FSDB files with file name pattern as follows:

```
(<sessionID> | partition)_<partitionID>.session/<filename>.fsdb/
<filename>_<postfix>.fsdb
```

The makeup FSDB file `[<sessionID>.]<filename>.fsdb.makeup.fsdb` is also included in the virtual file, if generated.

To view the generated virtual FSDB file in Verdi or nWave, use the following commands:

```
verdi -ssf [<sessionID>.]<filename>.fsdb.vf
nWave -ssf [<sessionID>.]<filename>.fsdb.vf
```

For more information on the `fsdb_merge` utility, refer to the *Programs and Utilities* chapter of the *VXE Command Reference Manual*.

Generating FSDB Files for Specific Instances

You can generate instance specific FSDB files if the complete hierarchy from the design's top module is not required. For example, you might need to use the power calculation tool to calculate an IP's power consumption. The IP might be initialized as an instance in the design. In such cases, you can generate instance-based FSDB file.

Instance-based FSDB file generation is supported in both online and offline SA and ICE mode, and supported only for non-batch FSDB generation. It is not supported for batch mode FSDB generation.

© 2021 All Rights Reserved.

To generate FSDB files for a specific instance, specify the `-instance` option with the `database` <*database_name*> command.

The `-instance` option is only applicable to the FSDB files. This option is inapplicable to the SHM and VCD files. If the `instance` <*instance_fullname*> option is specified with the `database` <*database_name*> command, the specified instance's cell name is assigned as the FSDB files' top scope name. The hierarchical names of the signals in the FSDB files appear under the specified instance, however, the hierarchical names do not include the instance name.

For example,

- Instance name is `top.PAD0`
- Signals under `top.PAD0` are `top.PAD0.clk` and `top.PAD0.\gen_4_.dut.reset`.
- Instance `top.PAD0` cell name is `IP`.

The FSDB signals will have hierarchical names as `IP.clk` and `IP.\gen_4_.dut.reset`.

Note: Ensure that the `probe` commands also specify the full instance name in the `probe` <*instance_full_name*> `-depth` <*n*> command. The probe command fails if the full path of the instance is not specified in the `probe` <*instanceFullName*> `-depth` <*n*> command.

Example:

Following is an example of an offline script to generate FSDB file for the `PA0.\\gen_label[7]. u_cntb` instance:

```
debug .
host -offline trace.phy
database test1 -instance "top.PA0.\\gen_label[7].u_cntb "
probe -depth 0 "top.PA0.\\gen_label[7].u_cntb "
database -upload
```

Debugging in Offline Session

Offline debugging can be performed only in FullVision mode, and not in DYNP mode. This functionality allows you to run the design and save the trace results for later debugging without needing to run an emulator session. This frees the emulator for the use of others.

To perform offline debugging, you must first run an online debug session to generate the trace data. Then, release the emulator, open the trace data in offline session, and generate waveforms for the additional probes.

© 2021 All Rights Reserved.

In the offline debug mode, you only have limited functionality available. You can add probes and examine signals in the waveform viewer, but cannot actually debug the design using other debug features, such as inspecting memory contents, as you would while running the design on the emulator in online session.

You can specify a new waveform database to save the existing waveform file. If a new waveform database is not specified, the existing waveform file is overwritten. Before bringing up the waveform viewer, you must add (or import) probe signals and upload the trace data in the xeDebug tool. You can then use the *File* menu of the waveform viewer to view a previously generated waveform file.

You can use the vvmDebug tool that offers enhanced offline debugging features for Palladium Z1 generated databases. Before debugging the designs offline using vvmDebug, you need to create VVM sessions online. The vvmDebug tool helps quick access to the memories during the offline session. You can perform various run-time and debug operations, such as generating waveforms, dumping memories, creating bookmarks, SDL programs, and probes, debugging RTL files and gate-level netlists, and analyzing power consumption of designs.

For information on how to create VVM sessions and the operations that can be performed through vvmDebug, see Debugging Designs using vvmDebug.

Generating Trace Data and Preparing for Offline Session

In an online session, open a database using the following command to record the trace data:

```
database -open <database_name>
```

After running the design for the desired number of cycles, use the `database -prepareOffline` command to generate the trace data for offline debugging. Execution of this command writes the traced data to an output subdirectory named as `<database_name>.phy` without generating the SST2 and FSDB waveform file. You can generate multiple trace output subdirectories for different offline debug sessions. For example, results for different test runs can be captured in subdirectories named `trace1.phy`, `trace2.phy`, and `trace3.phy`.

You can then use the trace data from the `<database_name>.phy` subdirectory during an offline session started through xeDebug GUI or command-line interface.

The `database -prepareOffline` command generates the `*.phy` subdirectory for the time window ending at current cycle. The size of the time window can be specified by using the `xeset traceMemSize` command.

Preparing for offline session helps running the design faster and you spend less time on the emulator. However, the `*.phy` subdirectory needs a preprocessing step to make offline

© 2021 All Rights Reserved.

debug session run faster. The data saved in `*.phy` subdirectory is organized into blocks. Each block is about 2000 cycles. Each block has a flag to indicate whether it has been pre-processed. When a block is first accessed in offline session, it needs to carry out a pre-process step, therefore waveform generation is slower. But, when the second time the block is accessed, the block has already been pre-processed so, the waveform generation is faster. For example, the first time to create waveform for time 0 to 2000 is slower because of the pre-processing. But, if more signals are added in the same time window, the waveform generation is faster.

Optimizing Trace Data for Faster Interactive Offline Debugging

To achieve faster interactive offline debugging, pre-processing on the `*.phy` subdirectory created in online session is required. Pre-processing computes waveforms for all the physical probes and saves the waveform data back to `*.phy` subdirectory.

Launching the fvCompute process starts pre-processing. Specify the following command before `database -prepareoffline` command to automatically launch fvCompute process in background after prepare offline is carried out:

```
xeset fvPreprocess on
```

Alternatively, fvCompute process can be launched from UNIX command line after creating the `*.phy` subdirectory in online session using the following command at UNIX prompt:

```
fvCompute <base_name_of_phy> [-nice <n>]
```

User can specify a nice value to assign priority of the fvCompute process. The default nice value is `20` meaning lowest priority. For more information on specifying the nice value, refer to UNIX help for details.

Pre-processing on the `*.phy` subdirectory should be performed in offline session. However, performance is slow the first time the waveform is created. An offline session can be opened when fvCompute is still running, and multiple fvCompute processes can pre-process the `*.phy` subdirectory to save time.

By default, fvCompute times out if it stays idle for 180 seconds. To overwrite the default number of seconds, use the `CDN_MT_HOST_TIME_OUT` environment variable. For example:

```
setenv CDN_MT_HOST_TIME_OUT 3600
```

Starting an Offline Session

When using the xeDebug command-line interface, use the following command to open the different trace output subdirectories in offline session:

```
host -offline <*.phy>
```


© 2021 All Rights Reserved.

Then, use the `probe -create` command to add more probes to debug the design and the `database -upload` command to upload the trace data to generate waveforms.

Debugging in Offline Session to Save Emulator Time

Using FV and InfiniTrace, you can view waveforms of any signals in the design over a longer time window. Cadence recommends to run an InfiniTrace *Prepare* session in online session and then perform interactive waveform generation in offline session, so that the emulator is free. Avoid generating waveforms in online session to reduce the time needed to occupy emulator.

Cadence recommends to view waveforms for a limited time window and limited set of signals so that the waveform can be generated within a few seconds. You can then switch to a different set of signals or a different time window. For example, the `*.phy` subdirectory contains waveform data for a large time window, for example a few million cycles. You can generate waveforms for smaller time windows, such as a few thousand cycles. Use the `database -tracewindow` command to specify a time window to generate the waveform.

Using Offline Debugging

The emulator is required for generating the `*.phy` subdirectory in the online session. After the `*.phy` subdirectory is created switch to the offline debug session. In offline session, use the following commands to select a small time window and observe waveform for small set of probes:

1. Specify the following command to start an offline session:

   ```
   host -offline <*.phy>
   ```

2. Set a small trace window using the `database -tracewindow` command.

   For example:

   ```
   database -tracewindow -start 6000 -end 10000
   ```

3. Generate the waveform using the `database -upload` command. For example, the waveform from `6,000` to `10,000` cycles is generated.

You can also add and upload more probes. If you do not use the `database -tracewindow` command to specify a smaller time window, the waveform is generated for the whole time range saved in the `*.phy` subdirectory.

© 2021 All Rights Reserved.

Additional Debugging Features in Offline Session

The following debugging features can be used in offline session:

- Setting Trigger Points Using SDL Simulator with InfiniTrace
- Analyzing Power Consumption Using DPA

Setting Trigger Points Using SDL Simulator with InfiniTrace

The SDL simulator can be used when tracing signals using FV and InfiniTrace. The SDL simulator uses the FV engine to get the signal values used in SDL.

Search a trigger point in *Observe* session using the following command:

```
infiniTrace -goto trigger
```

In HWIT, SDL logic running inside emulator searches the trigger point.

You can use the `sdl` command to set up a trigger condition and check the trigger status in a similar way as you do in the online session.

Analyzing Power Consumption Using DPA

Normal toggle count (NTC) generation and weighted toggle count (WTC) generation can be performed in offline session, thus saving the time spent on the emulator.

Set the size of the time window in which NTC/WTC are generated using the following command:

```
dpa -tracewindow -start <start_time> -end <end_time>
```

Alternatively, the following command can also be used:

```
dpa -tracewindow -start <start_time> -size <size_of_time_window>
```

If a time window is not specified, the DPA operation is performed on the whole time range of the `*.phy`.

Use `dpa -setupToggleCount` command to specify the set of instances for NTC or WTC. Use `dpa -getToggleCount` command to get NTC or WTC.

© 2021 All Rights Reserved.

Sparse Sampling

In the offline session, the sparse data in the `*.phy` subdirectory can be used to generate NTC, WTC and TCF. The waveform too can be generated, but since the waveform is also sparse its not very useful.

Cadence recommends to perform sparse sampling in online session. Specify a sample rate before running the emulation, so that when the trace memory is uploaded, the data saved in the `*.phy` subdirectory is already sparse.

Examples of Using Offline Session

The following three example scenarios show running under the xeDebug script mode or GUI interface mode. The first example shows a normal run in online debug session to prepare trace data for an offline session. The second shows offline debug session using xeDebug commands in script mode. The third describes an offline debug session using xeDebug GUI interface.

Example 1: Online debug session to prepare trace data for offline debugging

1. Specify the directory path to the design database using the following command:

   ```
   debug <dbPath>
   ```

   For example:

   ```
   debug .
   ```

2. Specify the emulation host name using the following command:

   ```
   host <hostname>
   ```

3. Download the design to the hardware using the following command:

   ```
   download
   ```

4. You can choose the operation modes to be used while debugging the designs. Specify the following command to select the debug mode:

   ```
   configPM -la|-vd|-stb
   ```

   For example:

   ```
   configPM -stb
   ```

5. Create a new database using the following command:

   ```
   database [-open] <database_name>
   ```

   For example:

   ```
   database -open mytrace
   ```


© 2021 All Rights Reserved.

6. Run the design using the following command:

   ```
   run
   ```

   Specify the emulation cycles for which the design needs to be run, for example:

   ```
   run 20000
   ```

7. Save data for offline session in a `*.phy` subdirectory using the following command:

   ```
   database -prepareOffline
   ```

   This option is supported in the FullVision mode, but not in DYNP (Dynamic Probes) mode.

8. Exit the debug session using the following command:

   ```
   exit
   ```

Example 2: Offline debug session using xeDebug command-line interface

1. Specify the directory path to the design database using the following command:

   ```
   debug <dbPath>
   ```

   For example:

   ```
   debug .
   ```

2. Specify the name of the database or the `*.phy` subdirectory that will be used in the offline session using the following command:

   ```
   host -offline {<*.phy_database_name>}
   ```

   For example:

   ```
   host -offline mytrace.phy
   ```

3. Create a new database using the following command:

   ```
   database [-open] <database_name>
   ```

   For example:

   ```
   database -open newwave
   ```

4. Create a probe during run time using the following command:

   ```
   probe -create
   ```

   For example:

   ```
   probe -create -ports .
   ```

5. Specify time window to generate the waveform:

   ```
   database -tracewindow -start <time> [-end <time> | -size <size_of_window>]
   ```

   For example:

© 2021 All Rights Reserved.

```
database -tracewindow -start 1000 -end 2000 or
database -tracewindow -start 1000 -size 1000
```

6. Upload the trace results from the emulator into the SHM or FSDB waveform database, so that it can be used by the waveform browser using the following command:

```
database -upload
```

7. Exit the debug session using the following command:

```
exit
```

Example 3: Offline debug session using xeDebug GUI

1. Open design in xeDebug GUI.
2. Select the *Offline* radio button.
3. Select a `<database_name>.phy` trace output subdirectory to be used.
4. Click *Open Offline* button to start the offline debug session.
5. Add and display probes using the Design Browser tool.
6. Exit the offline debug session.

Generating Waveforms in Offline Batch Mode

The performance of waveform generation has been improved to support fast billion gate design in the offline mode. You can probe large number of signals and generate waveforms using the *batch* mode. The batch mode uses multiple threads and generates multiple files to improve the waveform generation performance. You can generate waveforms in batch mode for both SST2 and FSDB formats. To generate waveforms in FSDB format, specify the `-fsdb` option with `xeDebug` command.

Note: The batch mode is supported only in offline sessions for both ICE and SA mode for SST2 and FSDB waveforms.

In previous releases, waveforms were generated in a single waveform database. The waveform generation with probing large number of signals and long run time was slow. To speed up with large number of signals and long run time, generate the waveforms in the batch mode. The `.phy` file is generated in online mode, and waveform generation is done in the offline mode based on the generated `.phy` file.

This section explains the steps to generate the waveforms in batch mode, some important considerations, and discusses how to view the waveforms in Simvision and Verdi:

- ❑ Generating Waveforms in Batch Mode

© 2021 All Rights Reserved.

- Important Considerations
- Viewing Multiple Waveform Files
- Limitations of Waveform Generation in Batch Mode

Generating Waveforms in Batch Mode

Specify the `-batch` option with the `database` command to enable the batch mode, as follows:

```
database <database_name> [-batch [-numFiles <num>]]
```

The `-batch` option enables the batch mode. *num* specifies the number of SHM and FDSB files to be generated in the batch mode. The actual number of files might be less than the specified number, to better partition amongst the probes. Parallel threads are created to speed up waveform generation. The number of parallel threads are determined by the number of generated files.

For details on the `database` command and options, refer to the description of the `database` command in the *Run-Time Command*s chapter of the *VXE Command Reference Manual*.

Follow the given sequence of commands to generate the waveforms in offline batch mode:

1. Execute the following command:

   ```
   database <database_name> -batch [-numfiles <num>]
   ```

   The default value for <*num*> is 10.

2. Specify the probes:

   ```
   probe <instName> -depth <n>
   probe <instName2> -depth <n2>
   .............
   ```

   Note: By default, the depth is `1`, that is, only the given scope is included in the probe set.

3. Generate the waveform database by executing the following command:

   ```
   database -upload
   ```

4. Exit the offline mode.

Result:

Multiple probe commands `probe <inst> -depth <n>` are used to collect the probe set. The total number of probes include all the possible nets in Palladium netlist under the specified instances.

© 2021 All Rights Reserved.

fvCompute computes waveforms for all the physical probes and saves the waveform data to the `*.phy` file. When the `database -upload` command is issued, instances and signals are partitioned into multiple probe sets.

The design hierarchy is generated into multiple SHM and FSDB files. Finally, the signal transitions are written into multiple SHM and FSDB files.

These files are generated in the database directory, that is, `./<filename>.shm` for SST2 waveforms. For FSDB waveforms, multiple FSDB files are generated under directory ./ <*database_name*>.fsdb. A virtual file with name as <*database_name*>.fsdb.vf is also generated. You can use the virtual file to view waveforms of FSDB format. See Viewing FSDB Waveforms section for details.

Both batch mode and non-batch mode write transitions into respective SHM and FSDB databases.

Example:

```
debug .
host -offline phyFile
database dbXYZ -batch -numfiles 2
probe up_counter.vhdl_cnt_instance1[0].dut_ -depth all
……
probe up_counter.vhdl_cnt_instance1[6].dut_-depth all
database -upload
```

Important Considerations

This section lists some important considerations for generating the waveforms in the batch mode:

- By default, the batch mode is disabled in the xeDebug GUI, and the `database` <*database_name*> `-batch` command is ignored with a warning message.
- If probing is done without issuing the `database` <*database_name*> [`-batch` [`-numFiles` <*num*>]] command, then the default database name is `trace`, and waveforms are generated in the non-batch mode — that is, the single thread mode.
- The batch mode can only be specified with a database file name in the `database` <*database_name*> `-batch` command. Switching from batch mode to non-batch and vice-versa is not supported.

© 2021 All Rights Reserved.

Viewing Multiple Waveform Files

This section explains the steps to view the SST2 and FSDB waveforms in SimVision and Verdi respectively.

- Viewing SST2 Waveforms
- Viewing FSDB Waveforms

Viewing SST2 Waveforms

SimVision displays multiple waveform files under the same hierarchy.

Invoke Simvision by executing the following command:

```
simvision  -snapshot <snapshot_name> <waveform_filename>
```

For details on the `simvision` command, refer to the *SimVision Analysis Environment* guide on the Cadence Online Support.

In SA mode, snapshot is generated by default. You can get the snapshot name by executing the following command:

```
xmls -snapshot //For Xcelium
```

For example:

```
xmls(64): 18.03-s007: (c) Copyright 1995-2018 Cadence Design Systems, Inc.
snapshot xc_ncwork.top:module (SSS)
```

In ICE mode, you must generate the snapshot separately to view the waveform in one hierarchy. Without a snapshot, multiple hierarchies are displayed in the sidebar when SimVision is opened. Use the `xmvlog` and `xmelab` commands to generate snapshot in ICE mode. For example:

To generate a snapshot for a design with top module named '`top`' in `test.sv` file, execute the following command:

```
xmvlog  test.sv -sv
xmelab top
```

Example:

The snapshot name is `xc_ncwork.top:module`, and the waveform file name is `wave.shm`. Execute the following command to open the waveform file and load the specified snapshot.

```
simvision –snap xc_ncwork.top:module wave.shm
```


© 2021 All Rights Reserved.

The Source Browser opens with full debug capabilities. You can select the probed signals and send them to the waveform window. The probed signals in all the generated files reside under the design root.

Viewing FSDB Waveforms

You can view waveforms of FSDB format using standalone Novas tools, such as Verdi and nWave.

Verdi can display multiple FSDB files under the same hierarchy through an automatically generated virtual file. In Verdi, you can view the waveforms in the following two ways:

- *Directly through the Verdi GUI*: Open the virtual file through the Verdi *File* menu.
- *Through the command line:* Specify the virtual file using the following commands:

```
Verdi -ssf <database_name>.fsdb.vf
nWave -ssf <database_name>.fsdb.vf
```

Support for SV Constructs:

By default, SystemVerilog constructs are probed and displayed in SystemVerilog construct format in the waveform window.

For limited SystemVerilog construct support at run time, you can use the `MAX_MDA_SIZE` <*num*> environment variable to set the maximum element size for multiple dimensional arrays.

To set maximum size for vector elements, use the `MAX_VECTOR_SIZE` <*num*> environment variable. By default, the maximum value for both `MAX_MDA_SIZE` and `MAX_VECTOR_SIZE` is 1048576.

When specific instances are enabled for FV compute using the `instanceFV` command, a signal or a FV construct not specified in the instance FV list is not probed. A message is displayed to show the nets that are not FV enabled.

Limitations of Waveform Generation in Batch Mode

The following are not supported while generating waveforms in the batch mode. These limitations apply to both formats: parallel SST2 and FSDB waveform generation:

- `-list, -event,` and `-delete` options of the `probe` command
- CPF/UPF low power object probing and waveform generation
- CPF/UPF power loss X display

© 2021 All Rights Reserved.

- Assertions
- Incremental probe and incremental upload operations
- Waveform generation in batch mode is not supported in online sessions.

Generating Waveforms in Offline DYNP Mode

The system enables you to save the emulation time by generating the waveforms in offline DYNP mode. You can generate the `*.phy` file in online sessions by executing the `database -prepareOffline` command in the online mode. After the `*.phy` file has been generated, release the emulator, and generate the FSDB and SST2 waveforms in an offline session.

Before launching the offline session, you must complete the following steps in the online mode:

1. Add the required probes by executing the following command:

   ```
   probe -create
   ```

2. Run the design for the required number of emulation cycles, for example, `run 10000`.
3. Execute the following command to generate the `*.phy` file. All successfully added probes will be saved to the `*.phy` file.

   ```
   database -prepareOffline
   ```

To generate the waveforms in DYNP offline mode, perform the following steps:

1. Release the emulator by executing the following command:

   ```
   host -release
   ```

2. Execute the following command from the unix prompt:

   ```
   generateDYNPwaveform [-fsdb|-sst2] <name_of_phy> <name_of_waveform_file>
   [-session <session_number>]
   ```

Compressing *.phy Files

`*.phy` is a trace file generated in fullvision debug mode that contains the data captured during the design debug process. You use the `*.phy` file to do offline debugging — add probes and generate waveforms.

However, size of the `*.phy` file is large and affects the usage. VXE supports the Zstandard compression methods to compress the `*.phy` files.

© 2021 All Rights Reserved.

Use the `COMPRESSION_METHOD` environment variable to specify a compression method. The valid values are `fast` and `slow`. The compression method is `fast` by default. For high compression ratio, change the method to `slow`.

The following table provides some reference for `*.phy` generation time and `*.phy` size for `fast` and `slow` compression methods:

Table 40-6 Supported Compression Methods

Compression Method	Time to dump *.phy for 2 million cycles	phy size before fvCompute	Time of fvCompute using 11 hosts	phy size after fvCompute
`fast`	83s	258M	395s	880M
`slow`	149s	133M	624s	764M

Streaming Waveform Data

The system enables you to stream the waveform data while the data is being captured. Streaming the waveform data uses regular memories in place of DCC. Streaming enhances the capture speed when the number of probes is small because memories are used instead of the DCC. Data is captured in selected cycles without any frame structure similar to DYNP probing mode. Capturing is done for only the nets and cycles in the probed signals.

Waveform Streaming and DCC capturing are independent, and can work concurrently. In both mechanisms, the waveform data is saved into a different waveform database. You can specify the name of the waveform streaming database and the master database for DCC capture. The two databases can also be displayed together through the Waveform Browser.

Streaming is done directly into the final waveform database using the default waveform format — SST2 or FSDB.

Waveform streaming can be done in two ways:

- **Streaming mode**: Data is streamed in memories and uploaded continuously during the run.
- **Buffered mode** (Non-Streaming mode): Data is streamed in memories and not uploaded continuously — you need to request to upload the data at the end of the run. See Generating Waveforms in Buffered Mode.

Waveform streaming is supported in the following operation modes:

© 2021 All Rights Reserved.

- STB (excluding InfiniTrace observe sessions)
- LA
- All SA run-time modes — run, TBrun, NBrun, and Autorun, and with/without uncontrolled clocks.
- DYNP

Follow the steps given below to perform waveform streaming:

1. Before compiling a design, specify the depth and width of the memory buffer that will be used for streaming inside the emulator. Use the following options in your compile script to specify the waveform streaming size:

   ```
   compilerOption -add {waveformStreamDepth <depth>}
   compilerOption -add {waveformStreamWidth <width>}
   ```

   By default, depth is 256k and width is 63 bits.

   For information about the above mentioned commands, refer to the description of the `compilerOption` command in the *User-Data Commands* chapter of the *VXE Command Reference Manual*.

2. Create a waveform streaming database by executing the following command at run time:

   ```
   database -stream [-open] <streaming_db_name>
   ```

3. Add the required streaming probes, as follows:

   ```
   probe -stream [-create] <net_name> <net_name> ..
   ```

For more information on the `probe` and `database` command options, refer to the description of the `database` and `probe` commands in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Controlling Waveform Capture using SDL

The `acquire` and `no_acquire` statements in SDL are used to control capturing in both the master waveform database and the waveform streaming database. Time segments in which data acquisition is turned off appear as gaps on the waveform display. Waveform streaming stops when SDL stops running — that is, at the end of the post trigger segment.

Generating Waveforms in Buffered Mode

You can switch to the buffered mode — that is, the non-streaming mode. The buffered mode operation is similar to waveform capturing with a master database. In this mode, the data is

© 2021 All Rights Reserved.

continuously captured into the memory buffer and not streamed to the disk. When the buffer fills up, newer samples overwrite the oldest samples.

In the buffered mode, you need to request to upload the data at the end of the run. The upload operation only uploads the last set of samples that were captured into the buffer, while older samples are lost.

To switch between the streaming and buffered mode, use the `database -stream -startStream` and `database -stream -stopStream` command.

For example,

The following command enables the buffering mode and stops uploading:

```
database -stream -stopStream;
run 10000
```

The following command uploads the data in the streaming probe memory:

```
database -stream -upload;
run 10000
```

The following command resumes data streaming and uploading:

```
database -stream -startStream;
```

Streaming Waveforms for Designs With Uncontrolled Clocks

When a design has uncontrolled clocks, you can control the sampling rate, as well as the time stamp for the master database using the `-captureMode` option of the `database` command

The streaming database uses the same capture mode as the master database. Therefore, the time stamp is same for the streaming and the master database. Reopening the master database with a different time stamp reopens the streaming database with a new time stamp.

Note: Waveform streaming is supported in both SA and ICE mode (1X and 2X clocking mode).

Improving Streaming Speed

Data is captured into the buffer only when at least one of the probed signals makes a transition. Therefore, the overall streaming speed depends on the percentage of cycles in which data transitions happen in probed signals. Additionally, the number of probes affect the streaming speed.

Consider the following to improve the streaming speed:

© 2021 All Rights Reserved.

- Avoid increasing the number of probes at run time because it decreases the streaming speed.
- Consider increasing the buffer depth. Buffer depth increases data packet size, which improves the streaming speed.
- Streaming speed might improve when running the DBEngine process on the same workstation as xeDebug. However, workstation load also affects the streaming speed.

When running with dynamic targets, you might lose some probe data if the streaming speed doesn't advance at the same rate as emulation speed. The lost probed data appears on the waveform display as a gap.

© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

41

Debugging Designs and Controlling the Run on Palladium Z1

This chapter includes the following sections:

- Using the Palladium Xref Browser of SimVision GUI in SA Mode
- Running Designs with Uncontrolled Clocks Using xeDebug in SA Mode
- Debugging Apparent Emulator Misbehavior
- Controlling Emulation Hang in SA Mode
- Creating Checkpoints to Save Emulator's State
- Compressing Saved Design Snapshots in ICE Mode
- Stopping Simulation with Initial Blocks
- Restarting Simulator and Resetting Emulator in SA Mode
- Starting Simulator on Remote Workstation
- Executing TCL Commands from C Testbench
- Viewing Optimized Nets in Design Browser
- Converting Signal Names
- Reducing Compiled Database Size
- Accelerating Download Speed
- Controlling Execution of GFIFO and DPI Functions at Run Time
- Controlling Execution of DPI Functions at Run Time
- Using RTL Names with Palladium Designs

© 2021 All Rights Reserved.

Using the Palladium Xref Browser of SimVision GUI in SA Mode

When you compile the designs for SA mode, the IXCOM compiler generates an instrumented snapshot for the Xcelium simulator. At the time of debugging, this snapshot is used, and it appears to be quite different from the original RTL design sources in the Source browser windows of the SimVision GUI. So, you might experience difficulties while watching active annotation during simulation or setting breakpoints.

To resolve these issues, you can use a side-by-side browsing tool, called the *Palladium Xref Browser*, which consists of essentially two source browsers that cross-link the instrumented snapshot back to the corresponding lines in the original design files.

To enable access to this browser, you need to include the `+xref_browser` option of the `ixcom` command at the time of compiling the design. When you include this option, the IXCOM compiler ensures that the mapping information from the original source browser to the instrumented sources is automatically generated. When you invoke the *Palladium Xref Browser*, it searches for the mapping file and returns an error if the file is not found.

This browser is available whether you invoke xeDebug independently by using the `xeDebug -gui` UNIX command, or by using the `xrun -xeDebug -gui` command. It is provided in the SimVision GUI under the *New* menu option. It supports both online and offline sessions. This browser is available **only in SA mode** from the SimVision GUI, and allows only object-level and file-level cross referencing.

You can open only one instance of the *Palladium Xref Browser* at a time. When you invoke the *Palladium Xref Browser*, two source browsers open side by side, and display the current simulation point. These source browsers are:

- The source browser in which your original sources are displayed. It is called the *User Source Browser*.
- The instrumented source browser in which the instrumented sources are displayed. This is the normal source browser window, which is titled *Source Browser #*. The # suffix in the source browser window title indicates the number of source browser windows that are currently open, for example *Source Browser 1*.

Figure 41-1 illustrates the two source browser windows of the *Palladium XRef Browser*. The instrumented source browser is displayed on the left and the *User Source Browser* on the right. In the following figure, the corresponding assertion is highlighted in both browsers.

© 2021 All Rights Reserved.

Figure 41-1 Palladium Xref Browser

Using the Source Browsers of the Palladium Xref Browser

Both windows in the *Palladium Xref Browser* are standard SimVision source browser windows and, therefore, have the standard menus: *File*, *Edit*, *View*, *Select* (disabled in the *User Source Browser*), *Format*, *Simulation*, *Windows*, and *Help*. The user source browser has neither the *Palladium action* toolbar, nor the *Scope selector*.

The *Palladium Xref Browser* automatically starts a processing loop to detect the changes in the instrumented source files, as these occur, either as a result of simulation advancing, or because of user interaction, such as changing the scope. If the instrumented source browser is blank, the *User Source Browser* is blank. You can initialize the instrumented browser by selecting an object in the design browser.

If the *User Source Browser* is closed, the *Palladium Xref Browser* feature terminates, but the instrumented source browser remains open. If the instrumented source browser is closed, the *Palladium Xref Browser* feature terminates, but the *User Source Browser* remains open. To re-enable the *Palladium Xref Browser*, re-invoke it from the SimVision pull-down menu; this will re-open its closed browsers.

© 2021 All Rights Reserved.

You can manually open a different file using the menus, or change the scope by using the scope dropdown box. Changes in the instrumented source browser scope or file affect the *User Source Browser*. But if no mapping information available, there is no change in the *User Source Browser*.

In the *Palladium Xref Browser*, the objects, for example signals, can be highlighted in either of the browsers. Simulatenously, the corresponding objects, are highlighted in the other browser. However, you can perform object-manipulation operations, like probing, forcing, or value annotation only in the instrumented source browser. Selecting objects in the other SimVision browsers, even another Design Browser has no effect on the *Palladium Xref Browser*.

Offline Debugging

This feature is available when you start an offline session of xeDebug, that is, without the simulator. During offline debugging, you do not have access to simulator-related features, such as breakpoints and pointer advancing. However, active annotation from the waveform in the instrumented source browser is available.

Assertions

Assertion failures also display the relevant assertions in the original source code. You can invoke the SimVision Assertion browser to display the failed assertion, and select the assertion to be sent to the SimVision source browser. Then, you can invoke the *Palladium Xref Browser* to display the original sources.

Limitations

The *Palladium Xref Browser* does not support the following functionality:

- Mapping original user design objects to the equivalent instrumented design objects to be used for probing and other standard functions is not supported. In the source browser, you can highlight strings in the text window, but you need to rely on mapping data, including column information, to find the corresponding instrumented object. Automated probing of user design objects is not supported.
- The IXCOM compiler does not provide line-to-line mapping information to allow correspondence points within the file other than objects. So, breakpoints in user sources and live position updates are not possible.
- Certain standard SimVision operations work orthogonally to this feature. So, the reset, restart, and save operations do not disable the *Palladium Xref Browser*.

© 2021 All Rights Reserved.

Running Designs with Uncontrolled Clocks Using xeDebug in SA Mode

At the time of compiling designs for SA mode using the IXCOM compiler, you can use a free-running or uncontrolled clock-generator module that can be enabled or disabled independently of the controlled clock of the simulator for the purpose of driving a dynamic target. By default, this is a free-running clock, but the system provides ways to stop it.

For information about how to model uncontrolled clocks in SA mode manually and automatically by using the *ixclkgen* utility, refer to the Modeling Clocks in SA Mode chapter.

This section describes how to run these designs in SA mode with uncontrolled clocks while using the xeDebug tool, and control the data capture and display at run time.

The features described in this section are supported only with the `xeDebug` and `xrun -xeDebug` commands; the `xrun` command without the `-xeDebug` option is not supported for uncontrolled clocks.

Capture and Display Control

The `-captureMode` option of the `database` command controls how the waveform display behaves with a design having uncontrolled clocks:

```
database -open <name> -captureMode [controlled | uncontrolled] [-event]
```

It is not possible to change capture mode on a database that is already open. This option can only be specified while the design is swapped-in to the hardware. It is applicable only for designs with uncontrolled clocks.

Refer to the description of the `-captureMode` option of the `database` command in the *Run-time Commands* chapter of the *VXE Command Reference Manual*.

© 2021 All Rights Reserved.

Table 41-1 Effect of the `-captureMode` and `-event` Options

Options	Time Stamp	Sampling
`-captureMode controlled`	controlled (simulator time)	**DYNP Mode**: One sample per simulator time slot **FV Mode**: One FV frame per simulator time slot
`-captureMode controlled -event`	controlled (simulator time)	**1x Mode**: Two samples per FCLK **2x Mode**: One sample per FCLK
`-captureMode uncontrolled`	uncontrolled (free-running)	
`-captureMode uncontrolled -event`	uncontrolled (free-running)	

Alternatively, you can select *Controlled*, *Uncontrolled*, or *Controlled with Event* from the *Capture Mode* box in the *Trace Data Capture* group box on the Control tab of the xeDebug GUI. For more information, refer to Control Tab of Chapter 42, "Using the xeDebug GUI."

If you open the waveform database using the `-captureMode uncontrolled` option of the `database` command, the probed data from the Xcelium simulator and the data from the Palladium Z1 emulator do not merge in the waveform browser. This is because both use a different time scaling, and there is no way to align the transitions correctly along the time axis.

The following limitations ensure that controlled and uncontrolled timestamp cannot be mixed in the same waveform database:

- If a database had been opened with the `-event` option while the design is swapped out to software mode, after the design is swapped in to the hardware mode, the sampling rate will honor the `-event` option as specified in Table 41-1 on page 1842.
- If the `-captureMode` option is set to `uncontrolled` while the design is swapped in to the hardware, then, if you choose to swap out to the software mode, the database is automatically closed.

Using the run Command with Uncontrolled Hardware Clocks

Issuing the `run` command starts the following:

© 2021 All Rights Reserved.

- Xcelium simulator
- Controlled clocks in the DUT
- Uncontrolled clocks in the DUT
- Capture clock

A non-blocking run allows the Xcelium simulator to run in the background when running in swapped-in mode. You can enter the commands targeted for both the simulator and the emulator while the simulator is running.

To enable a non-blocking run, use the `-nowait` option of the run command.

```
run -nowait [other-run-options]
```

Note: To allow your run-time command scripts to be compatible with ICE mode, the `-wait` option is accepted and ignored.

To stop a non-blocking run, you cannot use the combination of `CTRL+C` keys. But, you can stop it by using any of the other methods described in Stopping the Design Clocks and Data Capture.

You can switch back from a non-blocking run to a blocking run at any point by using the `waitWhileBusy` command.

Stopping the Design Clocks and Data Capture

The `clockConfig` XEL command determines how clocks are stopped. It helps in determining the clocks that will be stopped by the various events, such as SDL triggers or Xcelium breakpoints, that usually cause the run to end.

```
clockConfig -stopMode
            [controlled | *controlled | uncontrolled | both | none | *none]
```

Note: Specifying either the `uncontrolled` or the `both` option enables pausing the clocks for various run time services such as SDL `EXEC`, SDL `DISPLAY` or `TX`, continuous waveform upload, or any other run time service that requires pausing the clocks.

Alternatively, you can select the *Controlled*, *Uncontrolled*, or *Capture/SDL* check box adjacent to the *Stop Mode* field in the *Run Control* group box on the Control tab of the xeDebug GUI. For more information, refer to Control Tab of Chapter 42, “Using the xeDebug GUI.”

The following tables specify the effect of the `clockConfig -stopMode` command on:

- Controlled clocks (Table 41-2)

© 2021 All Rights Reserved.

- Uncontrolled clocks (Table 41-3)
- SDL and capture clocks (Table 41-4)

In these tables, “no” means the clock does not stop, and “stops” means the specified clock stops when the event happens. When the design is running in blocking mode, the control returns to the command prompt when the controlled clocks are stopped.

Table 41-2 Effect of using the clockConfig -stopMode Command on Controlled Clocks

Type of Event	**Option Used for** `clockConfig -stopMode`			
	controlled *controlled	**uncontrolled**	**both**	**none *none**
SDL Trigger	stops	no	stops	no
Xcelium Breakpoint `$stop` System Task `run` expires	stops	stops	stops	stops
`CTRL + C` keys `stop` command	stops	stops	stops	stops
`stop -emulation` command	stops	stops	stops	stops

The following table specifies the effect of the `clockConfig -stopMode` command on uncontrolled clocks.

© 2021 All Rights Reserved.

Table 41-3 Effect of using the clockConfig -stopMode Command on Uncontrolled Clocks

Type of Event	**Option Used for** `clockConfig -stopMode`			
	controlled ***controlled**	**uncontrolled**	**both**	**none** ***none**
SDL Trigger	no	stops	stops	no
Xcelium Breakpoint `$stop` System Task `run` expires	no	stops	stops	no
`CTRL + C` keys `stop` command	no	stops	stops	no
`stop -emulation` command	stops	stops	stops	stops

The following table specifies the effect of the `clockConfig -stopMode` command on SDL and capture clock. The `sdl -haltOnTrigger` command is disabled in the designs that have uncontrolled hardware clocks.

© 2021 All Rights Reserved.

Table 41-4 Effect of using the clockConfig -stopMode Command on SDL and Capture Clock

Type of Event	Option Used for `clockConfig -stopMode`					
	controlled	***controlled**	**Uncontrolled**	**both**	**none**	***none**
SDL Trigger	stops	stops	stops	stops	stops	stops
Xcelium Breakpoint `$stop` System Task `run` expires	stops	no	stops	stops	stops	no
`CTRL + C` keys `stop` command	stops	stops	stops	stops	stops	stops
`stop -emulation` command	stops	stops	stops	stops	stops	stops

Using Uncontrolled Clocks with Other Debugging Operations

The limitations and explanations in the following sub-sections apply only to the designs that have uncontrolled clocks; Other types of designs are not affected.

Dynamic Target Mode

If you switch to the Dynamic Target mode by using the `configPM -dynamicTarget` command, the clocks (whether controlled or uncontrolled) generated in the DUT become free-running clocks. All these design clocks will remain free running until stopped with the use of the `stop -emulation` or `configPM -staticTarget` command. Use of the `clockConfig -stopMode` command is not applicable, and, therefore, does not have any effect on the design running in the Dynamic Target mode.

InfiniTrace

InfiniTrace is not supported with designs that have uncontrolled clocks.

© 2021 All Rights Reserved.

SDL

The SDL feature is associated with capture clock. Stopping SDL means stopping capture (and vice versa). The SDL state is always clocked using FCLK, which is the fastest clock in the system.

When SDL is enabled, it is also associated with the “busy” status. Which means that SDL and the system is considered busy as long as SDL is running.

The SDL `EXEC` feature is supported with all the settings of `-stopMode`. When `-stopMode` is selected as `controlled`, `*controlled`, `none`, or `*none`, the `EXEC` action is executed without pausing the uncontrolled clock domain. This means that the `EXEC` action might be executed with thousands of cycles after the condition that triggered it, possibly resulting in an EXEC overflow condition.

The SDL display feature is supported with all the `-stopMode` options, but a buffer overflow might happen when `-stopMode` is set as `controlled`, `*controlled`, `none`, or `*none`, thereby resulting in a loss of display messages.

Debugging Apparent Emulator Misbehavior

In rare cases, the behavior of a certain net and its driving logic in the emulator might appear to contradict the design RTL or design netlist. Possible reasons for such a contradiction include the following:

- Multiple Drivers on the Net
- Net Behavior is Altered by userdata
- Net is Forced at Run Time
- Compiler Bug
- Hardware Misbehavior

Multiple Drivers on the Net

Sometimes the output net of a certain gate might seem incorrect relative to the inputs of that gate. A possible reason is that the output net has other drivers in the design besides the gate in question. In the ICE flow, by default, multiple (non-tristate) drivers on a net cause compilation to fail. However, some options, such as `ignoreMultiDrv`, allow the compile to pass. In the IXCOM flow, multiple drivers do not cause compilation failure.

© 2021 All Rights Reserved.

Multi-driven nets are reported in the `tmp/drc.rpt` file and the drivers of each such net are listed. For example,

```
Warning #100011 (multi-driven net): Ignore multiple drivers on regular
    net ex_req
List of drivers:
    pin O of instance ex_req_reg.U0.U00 of type BUF (buf)
    pin O of instance U1.U00 of type O2 (or).
Remaining driver is:
    pin O of instance U1.U00 of type O2 (or)
```

If you observe that a gate's output is incorrect relative to its inputs, search the `tmp/drc.rpt` file for the instance name of the misbehaving gate. Do not try to search for the name of the misbehaving net. A given net can have multiple names, and only one of the names (usually not the name you search for) will appear in `drc.rpt`.

Net Behavior is Altered by userdata

A net's behavior can be altered by `userdata` such as `tieNet`, `clockAssign`, and `clockSource`/`clockFrequency`. If a net behaves unexpectedly, check whether the net's behavior is being altered by userdata.

Unfortunately, a net might have multiple names, and the name used in userdata might be different from the name of the net whose unexpected behavior you observed. You can find all the names for a given net using the `signal -alias <signalname>` command in xeDebug after download. If you do not want to download the design, you can invoke xeDebug and give the following commands at the XE prompt:

```
debug .
crt openPDB rt
signal -alias <signalname>
```

Find all the names for the misbehaving net, and check whether any `userdata` using an alternate name has modified the net's behavior.

Net is Forced at Run Time

Sometimes, you might execute a run-time script which uses an XEL `force` command to force a net high or low. If you overlook the existence of this `force` command, the net's constant value might seem to be a misbehavior.

In xeDebug, you can use the `signal -isForced <signalname>` command to find out whether a given net is forced. If the net is forced, the command returns `1`, even if the signal is forced using a different (but equivalent) name. You can list all the names for a given net using the `signal -alias <signalname>` command.

© 2021 All Rights Reserved.

Compiler Bug

If you suspect a compiler bug, check whether the behavior is consistent with the gate level netlist. If the behavior is consistent with the gate level netlist but not with the RTL, the problem might be a synthesis bug, or a bug in IXCOM. You can identify some synthesis bugs using the Conformal LEC program. For more information, see the Using Conformal to Verify IXCOM-Generated Output and Using Conformal to Verify HDL-ICE Compiler-Generated Netlist sections in the VXE User Guide.

Hardware Misbehavior

If a design behaves differently on different emulators or on different domains, it suggests that there could be a problem with the emulator hardware. You can identify emulator hardware problems using the `xeDiags` program. For more information, see the *Running Diagnostics* section in the *Palladium Z1 Planning and Installation Guide*.

Controlling Emulation Hang in SA Mode

The system enables you to control the hanging of the emulator between two synchronization (tbsync) events. A tbsync event refers to a situation when emulator's controlled clock is stopped for some time.

Emulation might sometimes hang due to the following reasons:

- Existence of asynchronous loops: You can avoid the asynchronous looping errors using the `precompileOption -rm {breakECMloop}` option in your compile script. In case of errors, refer to the `PDB/hdldb.xcva_top.loops.bad file` to fix the issue.
- Presence of a 0 delay loop — that is, no delay combinational feedback loop. The simulation might become unresponsive if the testbench has a 0 delay loop. Following is an example of HDL with 0 delay loop:

  ```
  reg A;
  initial #1 A = 0;
  always @(*) A <= ~A;
  ```

  To detect a 0 delay loop, add the `-linedebug` option to `xrun` and press Ctrl-C while running emulation.
- Simulator waits for an event from the emulator because `$export_event` might be unspecified. For information on the syntax and usage of the `$export_event` directive, refer to the Performance Considerations for Modeling Clocks in SA Mode section.
- Simulator compiles the interface or the module again even if it has already been compiled by the IXCOM Compiler. To resolve this issue, inspect both the IXCOM compile script

© 2021 All Rights Reserved.

and the Xcelium compile script. If you find a duplicate file name, remove it from the Xcelium compile script.

- The `$ixc_ctrl` compiler directives are incorrectly specified in the behavioral modules to guide the Behavioral Compilation. For information on various compiler directives and their uses and syntax, refer to the Using $IXC_CTRL Directives to Guide Behavioral Compilation section.

You can control the hanging of the emulator in various ways, as follows:

- Aborting the Emulator Hang Using Files
- Controlling Behavioral Cycle of Waveforms
- Forcing TBsync Events in TBrun Mode

Aborting the Emulator Hang Using Files

You can abort the emulation hang while running the designs in SA mode. When the emulator stops responding and provides no messages for a while, issue the following command from another xterm to abort the emulation hang.

```
Linux> touch <run_dir>/tmp/uaPollAbort
```

The Xcelium process forces the emulator controlled clock to stop and return to the simulator prompt. You can, then, debug online or upload waveforms.

Before returning the prompt, the Xcelium process prints the critical Emulation Control Module (ECM) information. Cadence recommends that you provide the ECM information to the cadence support for analysis:

Sample Output

```
INFO(xc): /home/qshen/fb_old/test/ixvc/xc_unit/junk/tmp/uaPollAbort detected!
Aborting ...
--- UACMD: signal -get xc_top.tbcPO @1999
0
--- UACMD: signal -get xc_top.callEmuPI @1999
0
--- UACMD: signal -get xc_top.sendPO @1999
0
--- UACMD: signal -get -radix dec xc_top.simTime @1999
1999
--- UACMD: signal -get -radix dec ixc_time.nextClkTimePO @1999
2000
ERROR(xc): uaPollEmu abort @1999!
```

The system adds one time unit to the hardware time to accommodate the waveform tools that do not expand the final time unit.

© 2021 All Rights Reserved.

Note:

- The file based abort does not control the other types of hangs such as simulator hang in HDL loops, PLI, or DPI code while processing a TBCall service.
- In some situations, the test might not print messages in normal run. In such cases, you should check if the DUT clock advances in time.
- The file based abort replaces the `xc uaPollLimit` command. The `xc uaPollLimit` command is now deprecated.

Controlling Behavioral Cycle of Waveforms

The hardware run often hangs when the behavioral logic cannot be terminated. Multiple behavioral cycles in one time step might exhaust the trace buffer. To debug this issue, you can limit the behavioral cycles and force the emulator to stop capturing the waveforms after reaching the maximum behavioral cycle threshold by executing the following command:

```
xmsim> xc ecmset maxBpCycle <n>
```

You can execute the above command at the beginning of the hardware run or in the middle of emulation. To break the emulator hang, execute the following command from another xterm:

```
Linux> touch <run_dir>/tmp/uaPollAbort
```

Forcing TBsync Events in TBrun Mode

You can apply the GFIFO or SFIFO streaming technology for transaction modeling to achieve the ICE speed with the NBrun mode. However, when switching from the NBrun mode to the TBrun mode for debugging, the test cases might hang because no tbsync event is programmed into these BFMs.

In the TBrun mode, TB and emulator have to execute in turns. When the emulator is running, the TB threads waiting for the GFIFO events are only scheduled but cannot continue after the GFIFO functions are executed. These scheduled TB threads continue only by a tbsync event that stops the DUT. If you do not program any tbsync event in the BFMs, the emulator keeps running and TB hangs due to lack of tbsync events.

To resolve such hangs, execute the following `xc` commands to generate the tbsync events:

- To specify the maximum number of FCLK cycles between the tbsync events, execute the following command:

  ```
  xmsim> xc ecmset maxSyncCycle <n>
  ```

  Here `n` is the maximum number of FCLK cycles between the tbsync events.

© 2021 All Rights Reserved.

- To specify the maximum number of FCLK cycles between a GFIFO call and a tbsync event, execute the following command:

  ```
  xmsim> xc ecmset maxGfifoSyncCycle <n>
  ```

  Here `n` is the maximum number of FCLK cycles between a GFIFO call and a tbsync event.

Note: The above `xc` commands are not supported in the NBrun mode.

Creating Checkpoints to Save Emulator's State

The system enables you to save and restore the DUT states in the emulator while running the designs in the hardware mode. You can use various `marg_ckpt` functions to create checkpoints to save the emulator's state at various intervals. The checkpoints are stored in the host memory.

You specify the maximum number of checkpoints to be created. The default value is 1. You need to call the `marg_ckpt` functions to generate the checkpoints. The recent checkpoints are retained and the old checkpoints are removed automatically when the number of live checkpoints exceeds the specified maximum number. All live checkpoints are assigned unique ID's. You can select one of the live checkpoints to be saved to the disk.

For faster checkpoint performance, specify a list of memories to be excluded from all checkpoints. However, you must reload the specified memories after a checkpoint restore.

To restore a checkpoint file to the emulator, you must start a new xmsim session, then specify the checkpoint file after the `xc on -tbrun`, `xc on -nbrun`, or `xc on -run` commands.

Note: You can only restore a checkpoint file after the first `xc on -run, xc on -tbrun,` or `xc on -nbrun` command and before the first `run` command.

You can also save the emulator state to a disk file directly without creating a live checkpoint.

When a checkpoint file is saved from the emulator, only the emulator state is saved. The xmsim state is not saved. Also, the emulator timestamp is not saved. The emulator state includes clockgen states, scheduled clockgen events, active forces, registers values, and un-excluded memories.

When a checkpoint is restored to the emulator in a new xmsim session, only the emulator state is restored, the xmsim state is not changed. The emulator timestamp is synchronized with the current xmsim time after restore.

The following `marg_ckpt` functions help you create checkpoint files to save and restore the emulator states.

© 2021 All Rights Reserved.

Table 41-5 marg_ckpt Functions

Function	Description
`int marg_ckpt_exclude_memory(const char* path)`	Specifies DUT memory to be excluded from checkpointing. Returns `0` for success and non-zero for failure.
`int marg_ckpt_keep_alive( int num_ckpts);`	Specifies the number of checkpoints to be kept alive. If `num_ckpts == 0`, all checkpoints are removed. Returns `0` for success and non-zero for failure.
`int marg_ckpt_emu2live();`	Creates a live checkpoint from the emulator. The checkpoint is then added to the live checkpoint list. If a checkpoint is created successfully, a unique ID of the new checkpoint is returned. Similar to other `marg get handle` functions, `0` is considered a valid checkpoint ID. Returns `-1` for failure.
`int marg_ckpt_live2disk( int live_id, const char* name);`	Saves a live checkpoint to a disk file. Returns `0` for success and non-zero for failure.
`int marg_ckpt_emu2disk( const char* fname);`	Directly saves the emulator states to a disk file and does not occupy a live checkpoint space. Returns `0` for success and non-zero for failure.
`int marg_ckpt_disk2emu( const char* fname);`	Directly restores the emulator states from a disk file to the emulator. Valid if called immediately after the first `xc on -tbrun`, `xc on -nbun` or `xc on -run` command. Returns 0 for success and non-zero for failure.

The header file for these functions is `marg_ckpt.h`, which is installed in `<`*`vxe_install`*`>/share/vxe/etc/ixcom/marg_ckpt.h`.

You can also issue the `xc ckpt` command from the command prompt to execute the `marg_ckpt_emu2disk(), marg_ckpt_disk2emu()`, or `marg_ckpt_exclude_memory ()` function.

© 2021 All Rights Reserved.

Restrictions and Errors

- All the `marg_ckpt` functions that checkpoint either from and to the emulator must be executed in the hardware mode to be successful. However, the run mode does not matter. You can choose to run the designs in TBrun, NBrun, or Autorun mode.
- A non-zero failure value is returned if the `marg_ckpt` functions are called while running the designs in the software simulation mode.
- All functions that create checkpoints must be executed when there are no active DPI tasks. A DPI task is active when a task has not returned to its caller.
- All functions that restore a checkpoint to the emulator must be executed after the `xc on -tbrun`, `xc on -nbrun`, or `xc on -run` command and before the swap-in. The purpose of restoring a checkpoint is to replace the emulator swap-in.

  Note: The checkpoint files cannot be restored in the InfiniTrace sessions.
- The memories that are excluded by the `marg_ckpt_exclude_memory()` function will be reset to `0` at restore. You must load those memories after restore, if required.

Note: The checkpoint save and restore should not be used if you have a BFM with C proxy model such as SCEMI. The BFMs and C proxy must be in sync. The `marg_ckpt` functions can only save BFM states and not the proxy states. Therefore, restoring BFM states alone does not work unless you reset the BFMs to an initial clean state.

Compressing Saved Design Snapshots in ICE Mode

You can compress the design snapshots in the ICE mode to save the disk space and reduce the I/O time for reading and writing the snapshots.

To reduce the size of the resultant snapshot, use the following `-compress` options while saving the design state information:

save <*savename*> [-compress [{Low | Medium | High | None}]]

The system supports snapshot compression at the following levels:

- Low or L: Low reduction with high performance
- Medium or M: Medium reduction with good performance
- High or H: High reduction with low performance
- None or N: No reduction with high performance

© 2021 All Rights Reserved.

The compression is done individually on various saved design snapshots. That is, the resultant saved snapshot is not consolidated. The compression is performed as the snapshots are being written. Therefore, no temporary additional disk space is required in the compression process.

If the `-compress` option is not specified with the `save` command, or if `-compress` is specified without compression options, snapshots will be compressed at the lowest level by default.

The `-compress` option is not supported in the SA mode because the simulator compresses the saved snapshots automatically.

Note: For a given design, each higher level of compression results in an overall smaller saved snapshot. However, the actual results depend on the design attributes and the design state.

For information on how to specify the compression levels through the xeDebug GUI, refer to the Using Setup Tab in ICE Mode section.

Stopping Simulation with Initial Blocks

The system enables you to stop the simulation after the RTL initial blocks are executed and swapped in. To do this, use the `xc on -wait_init_stop` command and then run the design.

Note: The `-wait_init_stop` option must be specified together with the `-run`, `-tbrun`, and `-nbrun` options.

The following example shows how to control the simulation before SDL is loaded, using the `xc on -wait_init_stop` command.

```
xc on -zt0 -xt0 -tbrun -wait_init_stop
run
sdl -load trig.tdf
sdl -setFile trig.tdf
sdl -enable
sdl -verify
run
```

Restarting Simulator and Resetting Emulator in SA Mode

When you need to run multiple tests of the same design in SA mode, you do not need to exit from the emulator and connect again.

© 2021 All Rights Reserved.

Releasing the emulator and then re-connecting to it takes time. While running a design, you can reset the emulator to the initial status and restart the simulator to run another test without exiting from the emulator — using the `sim -restart` command.

To run another test of the same design, do not exit from xeDebug and execute the `sim -restart` run-time command at the xeDebug prompt.

For example,

```
xeDebug --xmsim -svseed random +UVM_TESTNAME=reg_test_1 --
debug .
host hsv-sc25
xc on -xt0 -zt0
run
sim -restart -svseed random +UVM_TESTNAME=reg_test_2
xc on -xt0 -zt0
run
```

Starting Simulator on Remote Workstation

In the SA mode, you can start the simulator on a remote workstation, if the workstation is available and accessible to you. Use the `-sim` option with the first `host` command to launch the simulator on a remote workstation, as follows:

```
host [-sim <simWorkstation>] {<workstation>|<emulator>} [<other options>...]
```

The workstations for the simulator and DBEngine can be different. The requirements for the remote workstation are similar to that for remotely running the DBEngine:

- `rsh` or `ssh` must run without a requiring a password.
- An Infiniband connection must exist to the target emulator SCD.
- The workstation operating system must support the specified simulator version.

You cannot change the remote workstation after the initial `host` command. To change the remote workstation to launch the simulator, you must release the current host and then re-execute the `host -sim` command.

Executing TCL Commands from C Testbench

You can execute the TCL or XEL commands from a C testbench for designs compiled in SA mode. Use the `marg_xe_cmd` function in your testbench to execute the required command, as follows:

```
int marg_xe_cmd(const char* cmd, const char** tclResult);
```


© 2021 All Rights Reserved.

The `marg_xe_cmd` is a C function that issues a command to the xeDebug process. The function returns `0` for success and non-zero for errors. You need to specify the TCL command and a string pointer to return the TCL command's result.

- *`cmd`*: Specifies the command to be executed from the C testbench.
- *`tclResult`*: Specifies a string pointer to return the command's TCL result. You can also specify a NULL pointer, if the result is not required.

Every TCL command might produce some output. The output is not returned to the C function as a string. The output is written to `stdout` and `xeDebug.log` file.

The `marg_xe_cmd` function is defined in `<install>/share/vxe/etc/ixcom/marg_user.h`.

Example

```
#include <marg_user.h>
const char* tclR;
if(marg_xe_cmd("sdl -enable", &tclR))
     vpi_printf("fail in marg_xe_cmd.\n");
vpi_printf("marg_xe_cmd TCL result:%s.\n", tclR);
```

Unlike the commands executed in xeDebug TCL prompt, the `marg_xe_cmd` function can only be executed in a task or another function. Run-time commands that do not change the state of simulation can be executed through the `marg_xe_cmd` function. For example, `value`, `memory -dump`, and so on.

Restrictions

- Only the debug commands are allowed in this function.
- Commands that advance time or stop the run, and change the simulation are not supported, such as `run, reset, restart,` and `stop`.

Viewing Optimized Nets in Design Browser

Design optimization includes removing duplicate logic and generating aliased nets, inverted nets and virtual nets. After optimization, a design uses smaller amount of emulation logic. Design optimization also includes design analysis — that is, reducing logic to constant 0 or constant 1 and removing the loadless logic. The gate count of the design includes all physical DADB nets, inverted nets, virtual nets, and aliased nets. The optimized nets that are not included are power nets, ground nets, and dead logic.

© 2021 All Rights Reserved.

You can use the `gateCount` utility that gives gate count per module prior to optimization. The `gateCount` utility gives an estimate, however, the Design Browser displays the actual final gate count. The final gate count (in the Design Browser) might be larger than the gate count report generated by the `gateCount` utility. The gate count report and the final gate count includes instrumentation.

You can also view some optimization during precompile using the `precompile -debug` command, which generates the `tmp/clean.rpt` file. This file includes a list of nets, which are tied either HIGH or LOW. The information in this file is analogous to the optimized information for power and ground nets that can be viewed in the DUV Design Browser.

The optimized nets can be viewed in the Design Browser while running the designs in both online and offline SA and ICE modes.

To enable the visibility of the optimized nets in the Design Browser, execute the following command at run time:

```
xeset -gui dbShowOptimization 1
```

The Design Browser takes time to load because the gate count is calculated at this time and is not cached.

© 2021 All Rights Reserved.

Figure 41-2 Optimized Nets in Design Browser

The `xeset -gui dbShowOptimization 1` command forces the following changes:

- The *Instance panel* displays instance names with the number of gates for all its sub-hierarchies, in parentheses, next to the instance. For example, `DUT` (`gates=33469`)
- The *Nets panel* displays nets with tag indicating the type of optimization performed on nets:
  - Power
  - Ground
  - Loadless logic
  - Optimized drivers

© 2021 All Rights Reserved.

For more information, refer to the description of the `xeset -gui dbShowOptimization` command in the *Run-Time Commands* chapter of the *VXE Command Reference* manual.

Converting Signal Names

At run time, if you need to convert signal names in your design from RTL to gate format or vice-versa, use the `convertName` run-time command, as follows:

```
convertName -rtl|-gate <netName>
```

Examples:

```
XE> convertName -rtl {top.hw.test.test_struct.\ss.s1.x}
top.hw.test.test_struct.ss.s1.x
XE> convertName -gate top.hw.test.test_struct.ss.s1.x
top.hw.test.test_struct.\ss.s1.x
```

You can also add the signals in a file, and specify the file with the `-inputFile` option, as follows:

```
convertName -rtl|-gate -inputFile <input_filename>
```

For example,

```
XE> convertName -gate -inputFile abc.txt
```

The above command converts the signal names in `abc.txt` to gate format and displays the translated names on the console. However, you can also save the translated signal names in a file using the `-outputFile` option of the `convertName` command.

You can also use the `convertName` command as a parameter to the other XEL run-time commands that accept net names in only gate format.

For details about the command options, refer to the description of the `convertName` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Reducing Compiled Database Size

You can reduce the size of the compiled database by removing the files that are not needed during the design download and debug. Use the `shrinkDB` command to remove a few unwanted files, and reduce the size of the compiled database to save the disk space.

You can execute the `shrinkDB` command in two ways:

- Use the `shrinkDB` command directly within your compilation script.

© 2021 All Rights Reserved.

- Execute the `shrinkDB` command after the compilation from xeCompile.

For more information about the files removed and examples, refer to the description of the `shrinkDB` command in the *Programs and Utilities* chapter of the *VXE Command Reference Manual*.

Performing Process-Based Save and Restart Operations

The Xcelium simulator supports the process-based save and restart operations in SA mode. The process-based operation saves the entire process state — that is, the memory, file pointers, and so on. You can save and restore the design snapshots in both *xrun* and xeDebug. The `save` command saves the current state of the emulator and the simulator, or the run-time environment. The `restart` command reloads an existing snapshot that was saved using `save` command.

Prerequisites

Ensure that you use **Xcelium 18.03.006** or higher version to perform the process-based save and restart operations.

Supported Modes

Process-based save and restart operations are supported in the following modes:

- **xrun** mode: `xrun -R` (without the `-xedebug` option)
- **xeDebug** mode: `xeDebug` or `xrun -R -xedebug --`

Note: Save and restart must be done in the same mode — that is, either *xrun* or xeDebug. `xrun -xedebug` is treated as the usual xeDebug mode.

Enabling Process-Based Operations

Use the Xcelium option `-process_save` to enable the process-based save and restart operations. This option supports the `$save` system task from within the Verilog code. With this option, the tcl `save` command uses the process-based save and restart instead of the regular save and restart mechanism.

- To enable the process-based save and restart operations in xeDebug, execute either of the following commands:

```
xeDebug --xmsim -process_save -- ...
```


© 2021 All Rights Reserved.

or

Execute the following command in xeDebug using *xrun*:

```
xrun -R -64bit -xedebug -process_save ...
```

- To enable the process-based save and restart operations in `xrun`, execute the following command:

```
xrun -R -64bit -process_save ...
```

For detailed information about the `-process_save` option, refer to the *Save/Restart User* guide in the Xcelium documentation set on the Cadence Online Support site.

Restarting Snapshots in xrun Mode

In the *xrun* mode, you can do a warm restart. In this mode, simulation runs in the same directory as the saved simulation, open files are modified, and the GUI sessions connected earlier are reconnected. You can perform a warm restart using the tcl `restart` command, as follows:

```
xrun -R -64bit -process_save ...
restart <snapnameshot_name>
```

Note: Xcelium cold restart is currently not supported with both legacy and process-based save and restore operations.

Saving and Restarting Snapshots in xeDebug SA mode

You can save the simulation state and then restore snapshots later in xeDebug SA mode. To do this, execute the following commands:

```
save [-path <path>] [-simulation] <snapshot_name> [-overwrite]
restart [-path <path>] <save_name> [-savedb]
```

For more information on the save and restart operations with VXE, refer to the `save` and `restart` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*. For details on how to perform the save and restart operations with Xcelium, refer to the *Save/Restart User* guide in the Xcelium documentation set on the Cadence Online Support site.

Accelerating Download Speed

The download speed for big designs has been enhanced in both SA and ICE mode. DBEngine performs hardware initialization while the `host` command is executed, thereby reducing the time taken for download.

© 2021 All Rights Reserved.

Fast downloading is enabled by default in ICE mode. It is, however, disabled by default for SA mode. You can enable or disable fast downloading using the environment variables and `xeset` commands, if required:

ICE mode:

```
setenv XE_ICE_FAST_DOWNLOAD [off |on | 0 |1]
```

SA mode:

```
setenv XE_SA_FAST_DOWNLOAD [off | on | 0 | 1]
```

Alternately, use the following commands at run time before the `host` command:

```
xeset iceFastDownload [off|on|0|1]
xeset saFastDownload [off|on|0|1]
```

The `xeset` commands get higher priority than the environment variables and an appropriate warning is displayed about the overriding of the environment variable setting.

Support for Target Relocation during Fast Downloading

With fast download, all target relocation commands must be put in a separate file and specified with the `xeset targetRelocationScript` command. Specify this command before the `host` command:

```
xeset targetRelocateScript <xxxx.xel>
```

If the `xeset targetRelocationScript` command is not specified, any target relocation command(s) issued after the `host` command would generate an error message:

```
ERROR (legacy-52192): command 'TargetAssign -add' cannot be executed after hardware
assigned, use 'xeset targetRelocateScript <*.xel>' before host command when
fastDownload enabled.
```

Target relocation is done per the following order of priority:

1. If test server is used to reserve the target location, test server generates target relocation file `relocateTarget.xel` which is loaded by xeDebug.
2. xeDebug loads `<topCell>.tp` if the file exists.
3. xeDebug loads the `*xel` file specified using the `xeset targetRelocateScript <xxxx.xel>` command.

© 2021 All Rights Reserved.

Example:

A design is compiled in ICE mode with `target 0_T1` on emulator `hsv-scd103`. Suppose the design needs to be relocated to `0_T0`. When fast download is disabled, you can perform target relocation in the following ways:

1. Use the test server to reserve the target using the `-relocateTarget` option, as follows:

```
// Content of test_server_reloc.xel

targetLocation -add myPod Tgt hsv-scd103 0_T0
targetAssign -add myTgt myPod

//Content of run script run0.xel:

debug .
host hsv-scd103
download
exit

// test server command with -relocateTarget

$ test_server COUNTER -host hsv-scd103 -reserve -relocateTarget
-xel="test_server_reloc.xel"
$ xeDebug run0.xel
```

2. Use <*topCell*>.`tp` to reserve a target:

```
// Content of COUNTER.tp

targetLocation -add myPod Tgt hsv-scd103 0_T0
targetAssign -add myTgt myPod

//Content of run script run0.xel:

debug .
host hsv-scd103
download
exit

$ xeDebug run0.xel
```

3. Use target relocation command(s) in the run script:

```
//Content of run script run1.xel:

debug .
host hsv-scd103
targetLocation -add myPod Tgt hsv-scd103 0_T0
targetAssign -add myTgt myPod
download
exit
$ xeDebug run1.xel
```

4. Use the `targetRelocateScript` variable to perform target relocation:

```
// Contents of target_reloc.xel

targetLocation -add myPod Tgt hsv-scd103 0_T0
targetAssign -add myTgt myPod

//Content of run script run2.xel

debug .
xeset targetRelocateScript target_reloc.xel
host hsv-scd103
```


© 2021 All Rights Reserved.

```
download
exit
$ xeDebug run2.xel
```

Note: Method # 3 is not supported when fast download is enabled, however, you can use all other methods, that is, #1, #2, and #4.

Controlling Execution of GFIFO and DPI Functions at Run Time

Controlling the execution of GFIFO functions in the design at run time by switching off and switching on selected GFIFO functions in the design helps improve the emulation speed. GFIFO functions are typically used for monitoring the design. At times, you might need to run a test suite with only the high-level GFIFO monitors switched on and the detailed, low-level GFIFO monitors switched off to improve the performance. If one of the tests fails, the failed test can be run again with the appropriate subset of low-level GFIFO monitors switched on for debugging.

For some designs, your code for GFIFO functions contributes to most of the emulation time and the hardware spins idly while your GFIFO code is being executed. You can check this in the `xcprof.out` file to determine whether it shows a large percentage of emulation time spent in `gbfull` behavioral waits. If the results of the execution of such code are not needed immediately, the emulation can be sped up by omitting the execution of these GFIFO functions in the emulation run and enabling the IXCOM run-time software to save the call information for the selected GFIFO functions in a database and replay these GFIFO calls later in a new run that does not need to occupy hardware. This behavior enables the emulation to finish sooner so that the hardware can be released for other jobs, and the required data is generated in a post-processing step that does not need to lock the hardware.

Switching on and switching off GFIFO functions at run time provides more flexibility and control than adding `if` statements in the RTL. With this feature, no recompilation is required if you change the set of GFIFO functions that are switched on or switched off, you have a great degree of control over the GFIFO functions you want to switch on or switch off. GFIFO functions can be switched off in software or in hardware.

The mechanism to turn off GFIFO was introduced in a prior release. This mechanism collects and sorts all GFIFO functions in hardware, brings them out to software, and executes only the GFIFO functions that have been switched on in software. It is useful if GFIFO functions switched off in software need to be saved during emulation and replayed later without using the hardware. This mechanism might also improve performance in designs that show a large percentage of time spent in `gbfull` behavioral waits in the `xcprof.out` file.

© 2021 All Rights Reserved.

This release introduces the mechanism to switch off GFIFO functions in hardware. This mechanism switches off GFIFO functions in hardware itself, so that these are not even collected and sorted in hardware. GFIFO functions that are switched off in hardware cannot be saved for later replay. This mechanism might improve performance in designs that show a large percentage of time spent in `gbfull` and/or `lbfull` behavioral waits in the `xcprof.out` file.

Note that only those GFIFO functions that send one-way information from hardware to software and are otherwise not critical to the correct functioning of the design should be switched off. For instance, a GFIFO function that sends transactions back to the hardware using SFIFO functions should NOT be switched off. Else, the design will not work.

You can switch on (or switch off) and save (or not save) calls to individual GFIFO functions or groups of GFIFO functions by using the `xc gfifo_control` commands.

Note: The `xc gfifo control` commands are not supported with multi-process GSFIFO.

You can achieve varying degree of control in the execution of GFIFO functions:

- Fine-grained control by targeting individual GFIFO functions and specifying their full hierarchical names
- Medium-grained control by targeting some or all GFIFO functions belonging to a module instance or channel
- Coarse-grained control by targeting some or all GFIFO functions under an instance hierarchy or those belonging to a group of channels having names that share the same prefix or suffix

Use the `*` (asterisk) wildcard while specifying the instance names, function names, and channel names in the `xc gfifo_control` commands described below to achieve medium and coarse-grained control. The use of the `*` wildcard is described in the Switching On or Switching Off GFIFO Functions section.

Switching On or Switching Off GFIFO Functions

Execute the following command at the `xmsim` or `xedebug` run-time prompt to switch on or switch off GFIFO functions:

```
xc gfifo_control [sw|hw] <instance_name> <function_name> on|off
```

By default, at the beginning of emulation, all GFIFO functions are set to `on`. The `on` and `off` attributes of the GFIFO functions are checked to determine whether or not the GFIFO function should be executed. If neither the `sw` nor the `hw` argument is specified, the matching GFIFO functions are switched off in software.

© 2021 All Rights Reserved.

You can also switch on or switch off these functions based on the channel assigned to these by using the following command:

`xc gfifo_control [sw|hw] channel` ***<channel_name>*** `on|off`

Important

Any `xc gfifo_control` commands that specify the `hw` argument must be issued **before** the design is swapped into hardware. GFIFO functions switched off in hardware cannot be saved for replay later.

Note that only OTB channel names can be specified with the `xc gfifo_control hw` <*channel_name*> `on|off` command. The SystemVerilog channel (named `SV_channel`) cannot be switched on|off in hardware. Specify the <*instance_name*> and <*function_name*> arguments instead of <*channel_name*> to control GFIFOs in `SV_channel` in hardware.

Using * (Asterisk) to Specify GFIFOs

You can use the * (asterisk) wildcard for specifying the <*instance_name*>, <*function_name*>, and <*channel_name*> in one of the following formats:

- Only the `*` (asterisk) wildcard
- Name with the `*` (asterisk) wildcard in the beginning or at the end, but NEVER BOTH in the beginning and end
- Complete name without any wildcards

Note: Only the `*` (asterisk) wildcard is allowed with these names.

Saving GFIFO Call Information

You can specify the GFIFO call information to be saved at run time. The GFIFO functions for which the call information is saved can be replayed later in a run that does not need to acquire the hardware. To control whether or not the call information for the selected GFIFO functions is saved, use the following command.

`xc gfifo_control` <*instance_name*> <*function_name*> `save|nosave`

Specify the `save` argument to save the call information for the selected GFIFO function(s), and `nosave` to not save the call information. By default, at the beginning of emulation, all GFIFO functions are set to `nosave`.

© 2021 All Rights Reserved.

Before the `save` attribute is specified for any GFIFO function, the `xc gfifo_control savedir` command must be issued to specify the directory in which GFIFO call information should be saved.

Note: The call information is saved only when the hardware is running.

You can also specify the channel name for saving the call information for the GFIFO functions for a channel. To do this, use the following command:

```
xc gfifo_control channel [sw|hw] <channel_name> save|nosave
```

The `on`, `off`, `save`, and `nosave` options can also be specified together in the same command, if required, as follows:

```
xc gfifo_control <instance_name> <function_name> on|off save|nosave
xc gfifo_control channel <channel_name> on|off save|nosave
```

The `on` and `off` and the `save` and `nosave` attributes are orthogonal, and all four combinations are permitted —that is, the attributes can also be used as follows:

- `on` with `save`
- `on` with `nosave`
- `off` with `save`
- `off` with `nosave`

To specify instance names with components that are escaped identifiers (that is these have spaces), enclose the entire instance name in curly brackets. For example:

```
{\top+cpu .\core&0}
```

This is similar to the way escaped instance names are specified in other TCL commands.

If multiple `on, off, save,` and `nosave` commands are supplied, these are executed in order in which these are specified. If a later command specifies a setting for a particular GFIFO function that conflicts with a setting specified by a previous command, the setting specified by the later command is used.

Viewing Settings for GFIFO Functions

After specifying the `on`, `off`, `save`, and `nosave` settings, you can confirm that the attributes are set correctly by using the following command:

```
xc gfifo_control print
```

The names of the GFIFO functions that are `$display` or `$fdisplay` functions, are printed in the following format:

© 2021 All Rights Reserved.

```
<instance_name>.gfdL <line_num>_<uid>
```

Here:

- <*line_num*> refers to the source line at which the `$display`/`$fdisplay` function is invoked.
- <*uid*> indicates a unique integer ID assigned by the IXCOM compiler.

The `xc gfifo_control print` command prints the hardware `on|off` status as well as the software `on|off` and `save|nosave` status of each GFIFO function. Following is the format of the output:

```
<inst_name>.<func_name> [hw-on|off] [sw-on|off] [save|nosave]
```

For example:

```
top.s1.F1  hw-on    sw-on    nosave
top.s1.F2  hw-on    sw-on    nosave
top.s2.F1  hw-on    sw-on    nosave
top.s2.F2  hw-on    sw-on    nosave
```

Specifying Save Directory for GFIFO Calls

Use the following command to specify the directory in which the GFIFO calls should be saved.

```
xc gfifo_control savedir <save_directory>
```

Note: You must execute this command before specifying the `save` option for any GFIFO function — that is, before executing any of the following commands:

```
xc gfifo_control <instance_name> <function_name> save
Or,
xc gfifo_control <channel_name> save
```

Replaying GFIFO Functions

Use the `xc gfifo_control replay` command to replay the GFIFO functions for which call information was saved in a previous run.

Note: The GFIFOs are saved only when the hardware is running. Therefore, the `replay` command should be issued at the same time when the design was swapped into the hardware during the emulation run that was used to save the GFIFOs.

The replay run does not need to acquire the hardware. However, it needs to follow the same initialization sequence as that of the save run. To do this, replace the swap-in command (`xc on`) with the `replay` command.

The syntax of the `xc gfifo_control replay` command is as follows:

© 2021 All Rights Reserved.

`xc gfifo_control replay <save_directory> [str] [sws] <start_time> [<end_time>]`

- <*save_directory*> is the directory in which GFIFO calls were saved in a previous run.
- `str` and `sws` are optional arguments that indicate the replay mode. See Selecting the Replay Mode section for details on the replay modes.
- The <*start_time*> and <*end_time*> refer to the interval in the previous run during which the GFIFO call information was saved. GFIFO calls made within the specified interval are executed with the arguments that were saved in the previous run. The <*end_time*> is optional and if not specified, the end time is assumed to be till the end of emulation.

  The time specification should be a simple non-negative integer, no units are permitted after the integer.The time should be specified as a positive integer or real number optionally followed by a time unit, such as 3.2us, 7.67e3 fs, 5, and so on. If a time unit is not specified the time value is assumed to be in the time unit of the design.

Note: GFIFO functions that are saved or replayed should be pure monitoring functions, which should not send transactions (or any data) back into the hardware using SFIFO.

Important Considerations for Saving and Replaying GFIFO Functions

Save and replay operations are suitable for GFIFO functions that only look at their arguments and do not update or look at any global state. Most monitoring functions fit into this category or can be made to, through minor restructuring.

Following are some important considerations for the `replay` command with examples:

- Replay Command with Start Time > 0

  If the start time is greater than 0, the replayed GFIFO functions should not depend on strict chronological execution order to work correctly.

  For example, if in a normal run, GFIFO functions are called at time 3, 5, and 10. Then, the call(s) at time 5 should not depend on any global state set by the call(s) at time 3, and the call(s) at time 10 should not depend on the global state set by the call(s) at time 3 and time 5 because you may choose a start time during replay that is greater than 5.

- Replay Commands with Non-Contiguous Intervals

  In the following example, multiple `replay` commands are run when the intervals are non-contiguous:

```
xc gfifo_control replay 0 5
xc gfifo_control replay 100 150
```


© 2021 All Rights Reserved.

Your code for GFIFO calls at time 100 and beyond should not assume that prior GFIFO calls between time 6 and time 99 have executed, because in this case they have not been executed.

- Replay Commands Going Back in Time

  In the following example, the second `replay` command goes back in time.

  ```
  xc gfifo_control replay 100 150
  xc gfifo_control replay 5 10
  ```

  Note: The code for the GFIFO call at time 5 should not assume that the only other GFIFO calls that could possibly have executed earlier, are those prior to time 5 or at time 5.

- Controlling GFIFOs in Replay Run

  The `replay` command can only replay the GFIFOs that were saved in the save run. You can further filter the saved GFIFOs to be replayed later by executing the `xc gfifo_control <instance_name> <function_name> on|off` command in the replay run prior to providing the `replay` command.

  By default, the `replay` command replays GFIFOs in a single-channel GSFIFO design in a single thread (that is, the `xmsim` thread) and replays GFIFOs in a multi-channel GSFIFO design in multiple threads (that is, the `xmsim` thread for the `SV_channel` and one OTB thread for each OTB channel).

  In multi-threaded replay, the threads run independently at their own speeds. Therefore, GFIFO calls across channels might not be replayed ordered by time. However, GFIFO calls within the same channel are always ordered by time.

Selecting the Replay Mode

The following replay modes are available with the `replay` command:

- Single-Threaded Replay Mode
- Synchronize with Simulator Mode
- Concurrent Save Replay Mode

Single-Threaded Replay Mode

The Single-threaded replay (STR) mode replays GFIFO calls made in a multi-channel GSFIFO design in a single thread (that is, the `xmsim` thread). This mode might be slow. However, it replays GFIFO calls ordered by time across all channels.

© 2021 All Rights Reserved.

To enable the STR mode, specify the `str` option with the `replay` command, as follows:

```
xc gfifo_control replay <save_directory> str <start_time> [<end_time>]
```

This mode helps improve the performance of designs bottlenecked by GFIFO workload that cannot be run with multi-channel GSFIFO because of thread-safety issues in your code, or other restrictions of multi-channel GSFIFO.

Such a design can be compiled for multi-channel GSFIFO with all monitoring GFIFO calls being saved and switched off during the hardware run. Running with multi-channel GSFIFO enabled can make the design run much faster. Then, you can replay the GFIFOs in single-threaded replay mode to avoid thread-safety issues.

Synchronize with Simulator Mode

The Synchronize with Simulator (SWS) mode also replays GFIFO calls ordered by time across channels. In addition, it always moves the simulator to the call time of each GFIFO before replaying the GFIFO call(s) at that time.

Note: The SWS mode can be selected only when replay is completed in a single thread (that is, when the design is a single channel design or the `str` argument is specified).

To enable the SWS mode, specify the `sws` option with the `replay` command, as follows:

```
xc gfifo_control replay <save_directory> sws <start_time> [<end_time>]
```

This mode is useful if a SystemVerilog or UVM testbench is processing the data returned by GFIFO calls and the testbench needs to be at the correct time when the GFIFO calls are invoked. During the replay, disable the DUT and any portions of the testbench unrelated to GFIFO processing. This avoids any redundant tasks, such as simulating the DUT or irrelevant portions of the testbench on a slow software simulator while the replay is being done. Disabling the DUT and portions of the testbench depend on the design and can be done by freezing specific clocks and/or control signals.

Concurrent Save Replay Mode

For faster turn-around time and reducing disk space requirements, you can enable concurrent save replay and, thereby, run the process that replays GFIFOs **concurrently** with the process that saves GFIFOs.

Without Concurrent Save Replay, you need to wait for the emulation run that saved GFIFO calls to finish before launching the simulation process to replay the GFIFO calls. For designs with many GFIFO calls, the save run can generate files that consume a large amount of disk space, and the need to wait for the save process to finish disallows any pipelining between the save and replay processes.

© 2021 All Rights Reserved.

With Concurrent Save Replay, you can launch the replay process before the save process has finished. The replay process reads the data files as soon as these are produced by the save process and replays the GFIFO calls in those files. The replay process waits for more data from the save process when it catches up with the save process. You can also specify that each file produced by the save process is deleted as soon as the replay process has consumed it. This results in a smaller amount of live data at any time, therefore, requiring lesser disk space.

Note that when a GFIFO function is switched off in hardware, it is not invoked in hardware and cannot be brought to software. So, even if the GFIFO is additionally marked as switched on in software or as being saved, it will be neither be executed nor saved.

To enable Concurrent Save Replay, use the following `xc gfifo_control` command:

```
xc gfifo_control replay <save_directory> [str] [sws] [csrd|csrp] <start_time>
[<end_time>]
```

Specify the `csrd` or `csrp` option to enable the Concurrent Save Replay mode and generate faster replay results.

The `csrd` option deletes each data file produced by the save process as soon as the replay command has consumed it. This results in a smaller amount of live data at any time, thereby reducing disk-space requirements. To issue additional replay commands once the `csrd` option has been used, regenerate the save data.

The `csrp` option preserves the data files produced by the save process. Therefore, additional replay commands can be serviced if the `csrp` option is specified.

Note: The replay and save processes can be multi-threaded. Therefore, you can choose to launch them on different hosts to avoid contention for CPU cores, memory and so on.

The `csrd` and `csrp` options can be used together with the `str` and `sws` options.

The `csrd` option always runs until the end of emulation, you cannot specify the <*end_time*> with the `csrd` option.

Note: To prevent the replay process from reading saved data from a previous run of the save process, delete any data saved earlier, before launching the save and replay processes. To prevent the save and replay process from overwriting each other, you MUST launch the replay process from a directory different from that of the save process.

For details of the process of launching the design run in a directory different from the one in which it was compiled, refer to the xmsim documentation.

© 2021 All Rights Reserved.

Controlling Execution of DPI Functions at Run Time

You might have linked some C functions for debugging, for displaying the design state, or for changing the design state. You can invoke such void DPI-C or C functions with constant arguments for debugging the design state at any time from the `xmsim` or `xedebug` run-time prompt by using the `xc dpicall` command as described below:

```
xc dpicall [-scope <scope_name>] <func_name> <arg1> <arg2> ...
```

The function can be a DPI-C `tbimport` function, a DPI-C GFIFO function, or an arbitrary C function.

The `-scope` option is useful when invoking a DPI-C context function. First, the command sets the scope to the specified scope by calling the `svSetScope()` function. Then, the DPI function <*func_name*> is called with the specified arguments.

The <*arg1*> <*arg2*> . . . arguments can be an integer, a number (for example, `5'b10101`), or a string.

Restrictions in Invoking void DPI-C and C Functions

The return value of the C function and its argument types are restricted. The function called in the `xc dpicall` command must conform to the following restrictions:

- The number of arguments must be <= 16.
- The type of each argument should be one of the following:
  - `char`, `int`, `long`, `long long`
  - unsigned versions of `char`, `int`, `long`, `long long`
  - `svBitVecVal`
  - `char*`
- The value specified for any argument of type `svBitVecVal` must be prefixed with `&` (ampersand), for example, `&5'b10101`. Otherwise, the argument will not be passed correctly.
- The bit width provided in the value for `svBitVecVal` arguments must match the bit width expected by the function. For example, `&5'b...`
- Any argument value that begins with the characters `0` through `9`, `-`, or `` ` `` is passed as a `long`, and is compatible with the argument types `char`, `int`, `long`, `long long`, and their `unsigned` versions. Any argument value that begins with `&` is passed as

© 2021 All Rights Reserved.

`svBitVecVal` of the size specified in the value. Any other argument is passed as `char*` that is, as a string.

- The return value of the DPI-C and C functions should be void.

© 2021 All Rights Reserved.

Using RTL Names with Palladium Designs

VXE supports the original Verilog and SystemVerilog names at compile and run time with Palladium designs, that is, you can specify the valid RTL names as an input to various XEL compile and run-time commands in both SA and ICE mode. RTL names are supported in both online and offline sessions. xeDebug GUI has an RTL Design Browser that shows various objects in RTL format for probing the objects. For details on the RTL Design Browser, refer to RTL Design Browser.

The RTL information is saved into an RTL database in QTDB sub-directory under the name `<design>.rtldb`. This includes name and type information available to the HDL-ICE Compiler and the block names available to the IXCOM parser.

Several user-data commands share the same C++ function and accept a list of gate-level net names. All such commands have been enhanced by enabling the use of RTL names of SystemVerilog objects, sub-objects, blocks, and so on. You can set a scope to any valid RTL scope, such as, a block or an array of instances, even if the scope does not exist in Palladium database.

To enable the support of RTL names, set the `XE_IMPORT_RTLNAME` environment variable at compile time. All run-time commands also accept the valid RTL names for design objects, sub-objects, ranges, and so on.

Using RTL Names for Forcing Signals

The `force`, `release`, `deposit`, and `value` commands can accept any valid RTL object, sub-object or a range of sub-objects. For example, for object: `struct {bit[7:0] b;} s[7:0]`, all the following `value` commands are valid:

- `value s`
- `value s[0]`
- `value s[1:0]`
- `value s[0].b`
- `value s[0].b[0]`
- `value s[0].b[1:0]`

However, Xcelium has a few limitations for these commands. For example, separated bits of a packed structure cannot be forced.

© 2021 All Rights Reserved.

42

Using the xeDebug GUI

A design compiled using either xeCompile, IXCOM tools, or *xrun* can be set up and run on the Palladium Z1 emulator using the xeDebug GUI. This chapter describes how to use the xeDebug GUI in ICE and SA modes.

Note: If you will be using the SimVision tool for viewing results, set up the `LD_LIBRARY_PATH` environment variable before invoking the xeDebug GUI. For details, refer to Setting Up the Environment for Palladium Z1.

Invoking xeDebug GUI

Execute the following command from the UNIX command line to invoke the xeDebug GUI:

```
xeDebug -gui [-shm|-fsdb|-vcd] \
             [-init <tcl_file>] ... \
             [-log <name>] [-append_log] [-append_key] \
             [--xmsim <option> ... [--]] [script [<arg> ...]]
```

For more information on invoking the debugger tools, refer to Invoking the Debugger Tool(s).

© 2021 All Rights Reserved.

Figure 42-1 Initial Screen

When the xeDebug GUI is invoked, only the *Setup* tab as illustrated in Figure 42-1 is displayed with the following fields:

1. *Database and Work Directory* group box:

   a. *Design Directory* field displays the location of the compiled design database. The path to the compiled design database can be changed manually or by using the directory tree displayed in the area marked as ***Step 4*** in Figure 42-1. However, to bring the change to effect, click the ✔ button adjacent to the field.

   b. *Work Directory* field displays the location of directory from which the xeDebug GUI was invoked. This field is linked to the design directory.

   c. *User Session* drop-down list box allows you to enter a new session name, or select an existing session subdirectory.

      If you specify a new session name, a directory by that name is created. The initial default user session entry is . (a dot) that means use current directory.

      If you select an existing user session directory, the tab shows design properties in case the directory has a valid compiled database; otherwise a message about missing database is displayed.

      Note: You can specify multiple user sessions, but can only open one session at a time. To use a session of a different name, you must release the emulator and choose another session to open.

© 2021 All Rights Reserved.

Individual user sessions are saved in a subdirectory specific to that session. This directory also stores files associated with that session, such as the trace waveform database, message files, and so on.

2. *Compiled for* field displays the mode for which the design was compiled, that is, *Simulation Acceleration* or *In-circuit Emulation*.

3. Design-related information such as, *Library*, *Cell*, *Last Compiled*, *Phy Files*, and *Compiled with User Data* is displayed in non-editable format.

4. The *directory tree* is displayed to provide the information about the location of the compiled design database. This area can be used to select the directory that contains the compiled design database and the path displayed in the *Design Directory* field is updated accordingly.

5. *Open Design* button when clicked opens the selected design for running and debugging. The layout of the *Setup* tab changes and provides you more fields based on the mode in which you are running the design, that is, SA or ICE mode.

   Note: The selected design is opened only if the specified design directory has a valid compiled database; otherwise a message about missing a database is displayed.

   For the steps that can be performed on the *Setup* tab after opening the design in SA mode, refer to Using Setup Tab in SA Mode; and for ICE mode, refer to Using Setup Tab in ICE Mode.

6. Select the *Regression* option to debug your design in the *Regression* Mode. In the regression mode, you can skip loading the Palladium database and instead load a smaller version of the database. The overall run-time environment preparation time is reduced and the `host` command execution is faster.

   The critical nets needed for the basic run are preserved and are called the preserved nets. Certain restrictions apply to the behavior of the run-time commands. For information about the behavior of various run-time commands in the regression mode, refer to the Improving Debug Turnaround Time with Debugging in Regression Mode section.

7. *Command prompt* area that displays the command executed on click of a button from the tab currently selected in the xeDebug GUI. This area also allows you to directly write and execute a supported command. For example, when you click the *Open Design* button on the *Setup* tab, the following command is executed and displayed in the *command prompt* area:

   ```
   debug .
   ```

   Note: The *command prompt* area is visible on all tabs that allow debugging functions.

© 2021 All Rights Reserved.

Using xeDebug Menu Bar and Toolbar Options

The xeDebug GUI window consists of a top area, containing drop down menus for *File*, *Edit*, and *Help* on the Menu Bar, and a secondary Toolbar with icons allowing you to access various debugging browsers and viewers. Below these are a series of tabs that enable access to various debugging functions.

Note: The tabs displayed in the xeDebug GUI and the fields that each tab comprises can vary based on the mode for which the design was compiled.

Menu Bar

Before or during a debug session, you can use the available menu items to configure the xeDebug GUI and manage content defined on the tabs.

The menu bar contains the following:

- File Menu
- Edit Menu
- View Menu
- Windows Menu
- Help Menu

File Menu

The *File* menu items allow you to control various properties of the xeDebug GUI, determine how you want to see or save data, or what data to maintain for the debugging between sessions.

Clicking *File* displays the following menu items:

- *Preferences...* (For details, refer to Defining the Preferred Settings.)
- *User GUI Setting...* (For details, refer to Specifying the User GUI Setting.)
- *View User Data...* (For details, refer to Viewing the User Data.)
- *User Design Information...* (For details, refer to Viewing the User Design Information.)
- This menu item is activated after the design has been downloaded to the emulator in ICE mode or the session has been started in SA mode.

© 2021 All Rights Reserved.

- *Testbench Probe Selection...*
- *Finish*
- *Exit*

Figure 42-2 File Menu Items

Defining the Preferred Settings

Clicking *File – Preferences...* displays the *Preferences* dialog box that has the following tabs:

- Message Tab
- Filter Tab
- Variables Tab
- GUI VariablesTab
- Fonts Tab
- GUI Tab Settings

Tip

The *Preferences* dialog box can be accessed either from the *File – Preferences...* menu item or the tools icon on the right side of the toolbar.

Message Tab

The *Message* tab (shown in Figure 42-3 on page 1882) allows you to select the check box corresponding to the type of messages that must be echoed while running and debugging the design. The supported types of messages are:

- Report: Controls the printing of report messages.

© 2021 All Rights Reserved.

- Progress: Controls the printing of progress messages.
- Information: Controls the printing of information messages.
- Warning: Controls the printing of warning messages.
- Error: Controls the printing of error messages.

Figure 42-3 Message Tab

Filter Tab

The *Filter* tab (shown in Figure 42-4 on page 1882) allows you to define the filter setting preferences, like whether the global (*glob*) or regular (*regex*) expressions should be used, and whether case sensitivity should be used or not. By default, the expressions style is set to global and case sensitivity is on.

Figure 42-4 Filter Tab

Variables Tab

The *Variables* tab (shown in Figure 42-5 on page 1883) allows you to view and configure the run-time parameters set using the `xeset` command on the command prompt. Specify the values and click *Apply* for the run-time parameter settings to take effect.

© 2021 All Rights Reserved.

Figure 42-5 Variables Tab (ICE Mode)

For information about setting the run-time parameters from the command prompt, refer to the `xeset` command description in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

- In SA mode, keep in mind the following regarding the *Variables* tab:
  - The following run-time parameters are not supported, and therefore, not visible on the *Variables* tab:
    - `depositSimMode`
    - `probePrimary`
    - `suspendEnable`

© 2021 All Rights Reserved.

- The *Variables* tab has the `sdlAutoSimBreakpoints` run-time parameter, which is not available in ICE mode.
- The `autoResumeRun` run-time parameter can be set to `On` only after swapping the design to run in emulation mode instead of software simulation mode.

GUI VariablesTab

The *GUI Variables* tab (shown in Figure 42-6 on page 1885) allows you to view and configure the GUI-related run-time parameters set using the `xeset -gui` command on the command prompt. Specify the values and click *Apply* for the GUI-related run-time parameter settings to take effect. After applying the selection, you must close and re-open the Design Browser for the options to take effect.

For information about setting the GUI-related run-time parameters from the command prompt, refer to the `xeset` command description in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

© 2021 All Rights Reserved.

Figure 42-6 GUI Variables Tab

Fonts Tab

The *Fonts* tab (shown in Figure 42-7 on page 1886) allows you to change the font of the text displayed in the various dialog boxes of the xeDebug GUI. You can change the font size, font family, or font type of a selected widget.

Note: The font changes are applicable only to the current debug session. When you exit, and re-invoke the xeDebug GUI, the default font settings are displayed.

© 2021 All Rights Reserved.

Figure 42-7 Fonts Tab

You can click the *Decrease Font Sizes* or *Increase Font Sizes* button to decrease or increase the font size of the xeDebug GUI. Click the *Reset Font Sizes* button to reset the font size to its original size.

The following table specifies the keys on the keyboard that you can use to change the font size:

Table 42-1 Keys to Change the Font Size on xeDebug

Keys	Description
CTRL+MINUS SIGN (-)	Decreases the font size
CTRL+PLUS SIGN (+)	Increases the font size
CTRL+ZERO (0)	Resets the font size to the original size

You can select the *All fonts of this type* or *Select widget* radio button to apply a new font to the xeDebug GUI. To apply a font, perform the following steps:

1. Select the *All fonts of this type* radio button to apply a font family to the xeDebug GUI. You can select *fixed*, *sanserif*, or *serif*.

Or,

© 2021 All Rights Reserved.

Select the *Select Widget* radio button to apply the selected font to the selected widget. You can select *console* or *SDL editor*.

2. Specify the font type as *arial*, *courier*, *helvetica*, *lucida*, *new century schoolbook*, *times*, or *terminal*.

3. Click the button to apply the selected font family or type.

GUI Tab Setting

You can show or hide certain tabs in the xeDebug GUI using the *File - Preferences - GUI Tab Setting* dialog. To hide a tab, unselect the corresponding checkbox and click *Apply*.

Figure 42-8 GUI Tab Setting

Specifying the User GUI Setting

Clicking *File – User GUI Setting...* displays the *User GUI Setting* dialog box (shown in Figure 42-9 on page 1888) that allows you to manage the GUI preferences.

© 2021 All Rights Reserved.

Figure 42-9 User GUI Setting Dialog Box

Viewing the User Data

Clicking *File – View User Data* displays the *View User Data* dialog box shown in Figure 42-10 on page 1888. You can use this dialog box to view the User Data saved during a previous session.

Figure 42-10 View User Data Dialog Box

© 2021 All Rights Reserved.

The *View User Data* dialog box has the following group boxes:

1. *Select user data to display*

   This group box lists all the User Data items with a check box adjacent to each. However, only those User Data items are active for which you had defined values at compilation time.

   To view the defined value for a specific User Data, select the check box adjacent to it.

2. *Display User Data*

   The values for the selected User Data items are printed in this group box.

Viewing the User Design Information

Clicking *File – User Design Information...* displays the *Design Information* dialog box shown in Figure 42-11 on page 1889. You can use this dialog box to view the design-specific information displayed on different tabs, such as *Probes* and *Waveform Format*.

Figure 42-11 Design Information Dialog Box

Edit Menu

During a debug session, you can select the *Edit* menu items to perform editing operations such as Cut, Copy, Paste, and Delete, on the selected text.

Clicking *Edit* displays the menu items shown in Figure 42-12 on page 1890.

© 2021 All Rights Reserved.

Figure 42-12 Edit Menu Options

View Menu

During a debug session, you can increase, decrease, or reset the font size of the xeDebug GUI. You can also view names of the current emulator and DBEngine host.

Clicking *View* displays the menu items shown in Figure 42-13 on page 1890.

Figure 42-13 View Menu Options

The *Current Debug State* option displays names of the current emulator and the DBEngine host being used.

You can also use the following keys on the keyboard to change the font size.

Keys	Description
CTRL+MINUS SIGN (-)	Decreases the font size
CTRL+PLUS SIGN (+)	Increases the font size
CTRL+ZERO (0)	Resets the font size to the original size

© 2021 All Rights Reserved.

Windows Menu

Displays all open windows in the xeDebug GUI.

Help Menu

Clicking *Help* displays the menu items that allow you to access information specific to the currently active tab, the Palladium Z1 manuals, and Cadence-specific support information.

Toolbar

The bar under the menu bar, known as the toolbar, displays the icons used for accessing the debugging browser and viewer tools. Some of the icons represent the following tools:

- Design Browser
- Waveform Viewer
- DUV Design Browser
- Symbols and Values Browser
- Register Viewer Browser
- Wave Viewer Window
- Low Power Verification
- Miscellaneous Tools

The top area of the xeDebug GUI window is common to all the tabs. Figure 42-14 on page 1892 shows this common area and identifies each icon.

© 2021 All Rights Reserved.

Figure 42-14 Common Access Toolbar

Note: The power tower icon on the toolbar requires a special license to enable its functionality. For details, refer to DPA Licenses in Chapter 32, "Analyzing Power Consumption of the Designs."

Design Browser

The Design Browser can be accessed from either xeDebug tool or *xrun* tool.

Note: The Design Browser is accessible only after you have opened a session in the xeDebug tool.

The Design Browser allows you to examine cells, instances, and nets of a design. It can also be used to examine the design hierarchy, and transfer object names to other windows through drag and drop.

If you specify the `-fsdb` option at the time of invoking xeDebug, the Novas tool design browser is displayed; otherwise, the SimVision Design Browser (as shown in Figure 42-15 on page 1893) is displayed.

Choosing Signals

You can choose signals either by entering the information directly into xeDebug GUI tabs and browsers, or by dragging and dropping the desired signal name from the Design Browser into the xeDebug GUI tabs and browsers.

Note: When running a design in SA mode, the SimVision Design Browser provides a *Show/Hide Generated* menu item that enables you to filter signals and instances based on name.

© 2021 All Rights Reserved.

The Design Browser can be used as a source of information for any text field in xeDebug that expects a signal or instance name. Some of the tabs that make use of such information are:

- *SDL* tab
- *Probe* tab
- *Force* tab
- *Deposit* tab (in SA mode)

To transfer an object name (instance or signal) into one of these forms:

1. Highlight the object on the Design Browser using the left mouse button.
2. Click with the left mouse button at the target insertion point in the xeDebug GUI.
3. Click the *Get Object from Browsers* icon on the xeDebug toolbar.

Figure 42-15 Design Browser (SimVision Version)

© 2021 All Rights Reserved.

Waveform Viewer

The second icon on the toolbar can be used to display the Waveform Viewer. If the Waveform Viewer is running already, clicking this icon button will update the waveforms. This button is not available (is grayed out) until you have opened a session.

If you specify `-fsdb` option at the time of invoking xeDebug, the Novas tool's Waveform Viewer is displayed, otherwise the SimVision Waveform Viewer is displayed.

Note: If the markers for SDL triggers are not visible in the SimVision Waveform Viewer, access the *File* menu and click *Reload Database* in the SimVision Waveform Viewer window.

Figure 42-16 SimVision Waveform Viewer Window

DUV Design Browser

The DUV Design Browser (as shown in Figure 42-17 on page 1895) allows you to examine design hierarchy of the entire design in ICE mode, or only the DUT portion of the design in SA mode.

© 2021 All Rights Reserved.

This design browser is available only when using the xeDebug tool, and can be used to drag and drop signals to the xeDebug GUI windows.

Note: The DUV Design Browser is not supported while debugging in the regression mode.

Figure 42-17 DUV Design Browser

To traverse the design hierarchy and view net names in the DUV Design Browser follow the steps mentioned below:

1. Expand or contract a design hierarchy by double clicking the instance name at the top of the browser.
2. To view all the nets or I/O terminals (ports) that belong to a specific design instance, select the instance name with the left mouse button, then double click on the line that

© 2021 All Rights Reserved.

shows the number of nets (on the left) or the number of terminals (on the right). This will expand the list of nets or terminals.

3. You can show or hide generated objects by selecting the *Gen* check box visible at the top right corner of the *DUV Design Browser* window. To refresh the list you need to close it if it is open and then open it again and do nothing if it was not open.

To drag and drop a net or instance name from the browser into one of the xeDebug GUI tabs or browsers:

1. Highlight the object name (instance name or net name) with the left mouse button.
2. Drag and drop the object name from the browser to the xeDebug GUI tab using the right mouse button.

Saving List of Nets to a File

1. In the *Nets* group box of the DUV Design Browser window (refer to Figure 42-18), click the *Nets: <n> of <n>* field to display the hierarchy view of the nets available in the currently accessed design and select the nets you want to export.

Figure 42-18 DUV Design Browser Window - Exporting Nets

2. Click *Export Nets...* from the *Probing and Tracing* group box. The *Export Browser* dialog box illustrated in Figure 42-19 is displayed.

   Note: To export all nets from the design, click *Export Nets...* without selecting any specific net from the hierarchy in the *Nets* group box.

3. Specify the file in which the selected nets should be exported using the *Export Browser* dialog box.

© 2021 All Rights Reserved.

Figure 42-19 DUV Design Browser Window - Export Browser Dialog Box

Selecting Probe Type

While probing through the DUV Design Browser, you can choose the type of probe — that is, regular or streaming probe. Automatic upload cannot be done for streaming probes.

In the *Probing and Tracing* group box, select the required probe type next to the *Probe* option:

- *Regular*: Creates regular probes at run time. This is equivalent to issuing the `probe -create -signal` command.
- *Streaming*: Adds the nets as the streaming probes at run time. This is equivalent to issuing the `probe -stream -create` <*net_name*> <*net_name*>...`-waveform` command.

Figure 42-20 DUV Design Browser Window - Probing and Tracing

For detailed information on probing, refer to the Setting Probes and Streaming Waveform Data sections.

© 2021 All Rights Reserved.

Probing Nets Saved in a File

1. Click *Import* from the *Probing and Tracing* group box of the DUV Design Browser window. The *Import File* dialog box illustrated in Figure 42-21 is displayed.

Figure 42-21 DUV Design Browser Window - Import File Dialog Box

2. Select the file for import and click *OK*.

Tracing Nets

1. Choose an object to trace and click *Trace* from the *Probing and Tracing* group box of the DUV Design Browser window. The *Palladium Connectivity Tracer* dialog box illustrated in Figure 42-22 is displayed.
2. Select a net or an instance from the list. The *Trace In/Out* and *Trace Inst* buttons are enabled. You can click the buttons to trace the data.
3. The *Trace In/Out* button can toggle to *Trace In* or *Trace Out*. The *Trace Inst* button can toggle to *Trace Net*.

Click *Export* to export the data to a file. The *Export Connectivity Data* dialog box is displayed. Specify the file in which the selected connectivity data should be exported.

© 2021 All Rights Reserved.

Figure 42-22 DUV Design Browser Window - Connectivity Tracer Dialog Box

RTL Design Browser

The RTL Design browser shows SystemVerilog objects in the RTL format. The RTL Design browser shows the scope and signal names:

- The *Scope* pane has a single hierarchy tree starting from the top design cell.
- The *Data* pane has a separate tree for every variable from the selected scope.

Both panes support multiple selection. The RTL Design browser also supports drag and drop. The browser is not enabled by default.

Note: To enable the RTL Design Browser in xeDebug, set the `XE_IMPORT_RTLNAME` environment variable at compile time.

© 2021 All Rights Reserved.

Figure 42-23 RTL Browser

The object types in the RTL database are categorized as follows:

Scope Types:

- Cell
- Instance
- Block
- Array of instances and blocks

Data Types:

- Library
- Net
- Vector – packed array
- Struct (packed or unpacked)

© 2021
All Rights Reserved.

- Union (packed or unpacked)
- MDA – unpacked array
- Terminal
- Array of objects of any data type

Similar to the DUV Design Browser, the RTL Design Browser provides the following options for probing objects:

- *Probe*: Creates probes from the selected objects.
- *Display*: Displays probes to waveform immediately.
- *AutoUpload*: Uploads the probes automatically.
- *Upload*: Uploads the trace results from the emulator into the waveform database.
- *Value*: Shows the value of the selected signals.
- PDB: Opens up the DUV Design Browser that shows the gate object corresponding to the selected RTL object.

Symbols and Values Browser

The Symbols and Values browser window, shown in Figure 42-24 on page 1902, is typically used for managing symbols in SDL programs, probe definitions, and force/release/deposit/value operations.

Note: This browser is accessible only after you have opened a session and downloaded the design.

The *Symbols & Values* window is divided into two regions. The left side of the window contains the *Signal List* and *Symbol Editor*. The right side of the window allows you to view emulated signal values during the run.

The *Signal List* group box serves as an intermediate repository for signal names. Use the *Get Net* button in this group box to port a net into this window from the Design Browser. From the Symbols and Values window, you can drag and drop into SDL or other forms by first selecting the net name (or a set of nets) with the left mouse button, and then dragging with the right mouse button.

The *Symbol Editor* group box at the bottom of the window allows you to define symbols that can then be used in other tabs, such as the *SDL*, *Probe*, and *Force* or *Deposit* tabs.

© 2021 All Rights Reserved.

Figure 42-24 Symbols and Values

Use the *Symbol Editor* group box to create symbols out of a collection of design signals. These symbols can be used for different purposes such as follows:

- Define SDL expressions
- More efficient and faster computation of set/force/get of signal values (through the *Control* tab)

Use the following procedure to create symbols:

1. Enter desired net or signal names into the *Signal List* group box (at the top).

   Double-click a single signal or highlight a group of signals, and click the button to enter the selected signal(s) into the *Symbol Editor* group box.

2. In the *Symbol Editor* group box, assign a name to the *Symbol* and click the button adjacent to the text field. It will be entered into the table in the *Symbol & Value* group box on the right of the window.

You can also display, but not edit, the symbol value in this window. The values can be displayed in either of the following forms:

- `bin` - binary
- `hex` - hexadecimal
- `dec` - decimal
- `oct` - octal

© 2021 All Rights Reserved.

Register Viewer Browser

At run time, you can view detailed information about various registers used in your design, from the xeDebug GUI. Use the *Register Viewer Browser* in the xeDebug GUI to view the information about registers, such as, register name, address, previous and updated values, and so on, any time during the debug session.

Pre-requisites

- Register Viewer requires an IP-XACT (`.xml`) file with hierarchical data structure as input. A hierarchy of XML tags required by the *Register Viewer* is as follows:

```
ipxact:component
   vendorExtensions:hdl_path - optional
   ipxact:memoryMaps
     ipxact:memoryMap
         ipxact:name
         ipxact:addressUnitBits - optional
         vendorExtensions:hdl_path - optional
         ipxact:addressBlock - 1-infinity
            ipxact:name
            ipxact:baseAddress
            ipxact:range
            ipxact:width
            vendorExtensions:hdl_path - optional
            ipxact:register - 0-infinity
               ipxact:addressOffset
               ipxact:size
               vendorExtensions:hdl_path - optional
               ipxact:field - 0-infinity
                     ipxact:name
                     ipxact:bitOffset
                     ipxact:bitWidth
                     vendorExtensions:hdl_path - optional
         ipxact:registerFile - 0-infinity
            ipxact:name
            ipxact:addressOffset
            ipxact:range
            vendorExtensions:hdl_path - optional
            ipxact:registerFile - 0-infinity - see above
            ipxact:register - 0-infinity - see above
```

For a description of `ipxact` namespace, refer to the IP-XACT specification at Accellera site.

Loading and Viewing Register Values

The `.xml` files can be huge in size. However, you can view selected registers of interest during the debug session.

Click the icon in the toolbar to launch the *Register Viewer Browser*. The registers are displayed in the *Register Viewer* window. The complete register hierarchy is available in a separate *Select Registers* window.

© 2021 All Rights Reserved.

You can add registers or remove them from the active list in the *Register Viewer*. The register list can be configured to save automatically and can then be restored at next debug session.

The *Register Viewer* enables you to perform the following operations:

- Automatic update of register values at the end of a run. The updated registers are displayed in a different color.
- Drag and drop operation — when objects are dropped, they are inserted as design objects.
- Change the radix of data output: `hexadecimal`, `binary`, `octal`, and `decimal`.
- Save the values of selected registers to a file.
- Enable or disable the selected registers.

Figure 42-25 Register Viewer

To load registers, click to select an `.xml` file containing register information. The *Register Viewer* displays the following information:

- Names and fields of registers
- Value of registers — a register has a value irrespective of its fields
- Size of registers or bit positions of a field inside a register
- Previous and the updated values of different registers
- Address of the selected registers. For example, `0x011`
- Complete hierarchical path of the selected registers. For example, `top.mid.bot.obj`

To save all registers from the *Register Viewer* to a file, select the *Save Watch List* option. The data is saved in text format and includes timestamps. When you save additional register values again in the same file, the content of the file gets appended.

© 2021 All Rights Reserved.

You can also change the display of the *Register Viewer* and select the columns to be viewed, using the *Settings* option. *Names*, *Bits*, and *Values* columns are always displayed.

The other options provided in the *Register Viewer* are as follows:

- : Launches the *Select Registers* window that provides hierarchical view of registers. Refer to Figure 42-26.
- *Refresh*: Updates all register values according to the run.
- *Auto*: Select this checkbox to automatically update the register values at the end of run.
- *Poll*: Refreshes the *Register Viewer* automatically after a specified time interval. Specify the time interval using the `xeset -gui pollingRate` [<*value*>] command.
- : Removes registers from the *Register Viewer*. You can also use the keyboard *Delete* key to remove the registers.
- and : enable and disable the registers. Values for the disabled registers are not displayed in the *Register Viewer* even if the values change during the design run.
- : Displays all information about the selected registers, such as address, size, and HDL path.

Figure 42-26 Select Registers Window

Name	Bits	address	hdl_path
mix			
map0			
addressBlock0			
PC_0	7:0	0	up_counter.dut_vlog.count
PC_1	7:0	0	up_counter.vlog_cnt_instance2[0].u_dut_vlog.count
PC_2	7:0	0	up_counter.vlog_cnt_instance2[10].u_dut_vlog.count
PC_3	7:0	0	up_counter.dut_vhdl.cntGEN(0).botVHDLcnt.botCountVHDL
PC_4	7:0	0	up_counter.dut_vhdl.cntGEN(1).botVHDLcnt.botCountVHDL
PC_5	7:0	0	up_counter.vhdl_cnt_instance1[0].u_dut_vhdl.cntGEN(0).botVHDLcnt.botCountVHD
PC_6	7:0	0	up_counter.vhdl_cnt_instance1[0].u_dut_vhdl.cntGEN(1).botVHDLcnt.botCountVHD
PC_7	7:0	0	up_counter.vhdl_cnt_instance1[6].u_dut_vhdl.botVHDLcnt.botCountVHDL
PC_8	7:0	0	up_counter.vhdl_cnt_instance1[7].u_dut_vhdl.botVHDLcnt.botCountVHDL
MY	7:0	F4240	up_counter.dut_vlog.count
PC	7:0	1E8480	
F1	0		up_counter.dut_vlog.count[0]
F2	3:1		up_counter.dut_vlog.count[3:1]
F3	4		up_counter.dut_vlog.count[4]
F4	5		up_counter.dut_vlog.count[5]
F5	7:6		up_counter.dut_vlog.count[7:6]
SINGLE	3:0	1E8480	

The *Select Registers* window shows all registers from the specified `.xml` file.

To add registers to the Register Viewer:

Select the registers from the hierarchical list and click to add the registers. Ctrl + Shift and Ctrl+A buttons enable multiple selection.

To search for specific registers:

© 2021 All Rights Reserved.

In the text field, specify an expression to select objects from the list that match the specified expression. The search text field supports regular expressions in glob style. By default, search is case sensitive and compact.

The compact search shows only the matching registers. For example, if a design has three registers: `reg1`, `reg2`, and `reg3`, then search for `reg1` will display only `reg1`. If, however, compact search is disabled, then search for `reg1` will display all three registers. You can change the search settings from *Settings* in the *Register Viewer* window.

Example

A testcase is provided in the gift directory at the following location:

`<install_dir>/share/vxe/gift/tutorial/register_viewer`

The directory contains a Makefile with several targets, `src` subdirectory with a Verilog file, a `.csv` file, and the `csv2xml1.tcl` script that can be used to convert a `.csv` file into an `.xml` file.

Wave Viewer Window

The *Wave Viewer* window in the xeDebug GUI provides quick assistance for waveform operations, and also displays the full state of the current SDL program.

The Wave Viewer supports the following operations:

- Load a data file (SST2 or PHY).
- Load an SDL file.
- Set a trigger expression for search.
- Go to a specific time point with or without triggering enabled.

Prerequisite

- To perform all operations listed in the preceding sections, you must have the Palladium database open, otherwise, an error message will be displayed at run time.
- Before loading a data file through the *Wave Viewer*, ensure that the file is already closed in the online session.

© 2021 All Rights Reserved.

Figure 42-27 Wave Viewer Browser

- In *Data File Path*, specify the data file (SST2 or PHY) which needs to be loaded for the active session.
- In *SDL File*, specify the SDL file to be loaded. You can also specify the trigger expression, if required.

  To reset the SDL file or expression, clear these fields and apply your changes using the button.
- The *Trigger* checkbox enables the SDL program and trigger expression during the run.

Furthermore, you can use the goto buttons to go to specific simulation time — you can go to the beginning or end of the simulation time or go forward/backward in simulation time. These buttons are provided at the bottom of the window.

If the *Continue Run* checkbox is selected, the `goto` operation does not reset the state of the SDL program. This option is identical to the continue run functionality for SDL programs.

The current state of the SDL program with all global triggers and instance values are displayed on the right pane in the *Wave Viewer*.

Low Power Verification

Clicking the lightbulb icon on the toolbar invokes the *Low Power Verification* window that helps in debugging low-power-aware designs.

© 2021 All Rights Reserved.

The layout of this window differs in ICE and SA modes. Refer to Figure 42-28 on page 1908 and Figure 42-29 on page 1909. For more information about the CPF *Low Power Verification* window, also refer to Using the xeDebug GUI for Low Power Verification of Chapter 33, “Specifying CPF Low-Power Intent on Palladium Z1.”

Figure 42-28 CPF - Low Power Verification Window (ICE Mode)

© 2021 All Rights Reserved.

Figure 42-29 CPF - Low Power Verification Window (SA Mode)

The various group boxes of the *Low Power Verification* window enable you to perform the following actions:

- Run time control
  - Enable or disable use of CPF in the design.
  - Specify that the power-loss values should be displayed as either `X` or real emulator values, if the CPF logic is enabled for the design.
  - Construct SDL statements.
- Get current power state values of the various CPF objects. In SA mode, you can use the Xcelium power commands on CPF objects or HDL objects.

© 2021 All Rights Reserved.

- Probe the CPF objects. In SA mode, you can also probe power domains, HDL objects in CPF expressions, and/or the retained values of HDL objects.
- Log the CPF events.
- View the CPF data by using:
  - A text window for displaying the CPF file.
  - The table view to display the CPF data from the Palladium Z1 emulator.

    The table displays all the CPF objects of the selected CPF object type. The CPF object name is displayed as a button that you can select to probe that object. If a type is grayed out, it means that CPF objects of that type do not exist in the design.
  - The standard hierarchical browser to display the CPF data in tree view.

For layout of the 1801 Low Power Verification window, refer to Figure 42-30 on page 1911. For more information about 1801, refer to Chapter 34, “Working with IEEE 1801 Low-Power Designs on Palladium Z1.”

© 2021 All Rights Reserved.

Figure 42-30 1801 - Low Power Verification Window

Miscellaneous Tools

There are tools available only for specific settings, as follows:

- ***Dynamic Power Analysis*** (the power tower icon on the toolbar)

  Invokes the Dynamic Power Analysis (DPA) menus. Requires a special license for use. For more information on DPA, refer to Chapter 32, “Analyzing Power Consumption of the Designs.” The DPA tool is not supported while debugging in the regression mode.

- ***Get Objects from Browser***

  This tool is used in conjunction with most of the SimVision and Novas browsers.

  When using the SimVision browsers, the *Get Objects from Browser* tool is typically used to get object names from the Schematic Browser, Design Browser, Source

© 2021 All Rights Reserved.

Browser, or Waveform Browser. When using the Novas tools, the *Get Objects from Browser* tool is typically used to get objects names from the TurboTracer viewer.

It allows you to highlight a net or instance in the browser and transfer the net name into empty fields on the xeDebug GUI tabs by clicking the button.

- ***Schematic Browser***

  The Schematic Browser displays the schematic view of a design as generated from its HDL source code. Signals can be dragged and dropped into waveform viewer.

- ***Exit***

  Exits the xeDebug tool.

Starting a Debug Session

When the xeDebug GUI is invoked, the initial *Setup* tab (shown in Figure 42-1 on page 1878) has the same fields in both SA and ICE modes. For details about the layout of the initial *Setup* tab, refer to Invoking xeDebug GUI.

However, after you open the design, the fields displayed on the *Setup* tab are different in SA mode (shown in Figure 42-31 on page 1913) and ICE mode (shown in Figure 42-35 on page 1919). The following sections discuss these differences.

Using Setup Tab in SA Mode

In SA mode, the *Setup* tab handles the tasks such as specifying the emulator, viewing emulator status, selecting vision and debug modes, and specifying *xrun* options.

After the design has been opened, the *Database and Work Directory* group box displays the information in view only mode (as shown in Figure 42-31 on page 1913). To start a debug session, perform the following steps:

1. Select whether to use the emulator for debugging in *Online* or *Offline* session.

   Note: If you select the *Offline* session, the layout of the *Setup* tab is different and steps discussed below are not required. Instead, refer to Setup Tab in SA Mode for Offline Debugging for the steps that follow the selection of *Offline* session.

   In the *Configuration* group box below the *Emulator* options, the following fields are displayed in disabled format:

   - *State* of the design run as *Simulation*

© 2021 All Rights Reserved.

- Buttons adjacent to *Change To* field
- Options accessible on clicking the *--> Show Hot Swap Options* button

Figure 42-31 Setup Tab after Opening the Design in SA Mode

2. Specify the emulator in the *Emulator* field and click the button.

a. A record of the emulators selected in the previous debugging sessions is maintained in a drop-down list. You can access this list by clicking the down-arrow button adjacent to the *Emulator* field. This is a handy shortcut for selecting an emulator.

b. You can use the button to check domain availability of the specified emulator. Clicking this button runs the `test server` program, and displays the available domains and their configuration. You can fine-tune results of the `test_server` program by using it directly on the command prompt, with its options.

Figure 42-32 shows an example of the *Test Server* dialog box displaying the results.

© 2021 All Rights Reserved.

Figure 42-32 Sample Test Server Dialog Box

```
Test Server
Emulator: mfg-scd5          Hardware: Palladium Z1        Configmgr: V15.1.1
Rack 0 has 3 clusters
Cluster 0 has 6 logic drawers            CCD: ONLINE
Logic drawer  0 has  8 domains               Logic drawer: ONLINE
Domain    Owner         PID             T-Pod    Design          ElapTime     Reser
0.0       NONE          0               --  --   NONE            --           --
0.1       NONE          0               --  --   NONE            --           --
0.2       NONE          0               --  --   NONE            --           --
0.3       NONE          0               --  --   NONE            --           --
0.4       NONE          0               --  --   NONE            --           --
0.5       NONE          0               --  --   NONE            --           --
0.6       NONE          0               --  --   NONE            --           --
0.7       NONE          0               --  --   NONE            --           --
Logic drawer  1 has  8 domains               Logic drawer: ONLINE
Domain    Owner         PID             T-Pod    Design          ElapTime     Reser
1.0       NONE          0               --  --   NONE            --           --
1.1       NONE          0               --  --   NONE            --           --
1.2       NONE          0               --  --   NONE            --           --
1.3       NONE          0               --  --   NONE            --           --
1.4       NONE          0               --  --   NONE            --           --
1.5       NONE          0               --  --   NONE            --           --
1.6       NONE          0               --  --   NONE            --           --
1.7       NONE          0               --  --   NONE            --           --
Logic drawer  2 has  8 domains               Logic drawer: ONLINE
                                                   Close
```

Note: To view the test server output in a compact format, select the *Short* option before launching the test server — the test server output looks similar to the screen shown below:

```
Emulator: hsv-scd903        Hardware: Palladium Z1        Configmgr: V18.6.0.8      System Status: ONLINE
Rack 0 has 2 clusters
Cluster 0 has 6 logic drawers            CCD: ONLINE
    LD  Status    D0    D1    D2    D3    D4    D5    D6    D7
     0  ONLINE     -     -     -     -     -     -     -     -
     1  ONLINE     -     -     -     -     -     -     -     -
     2  ONLINE     -     -     -     -     -     -     -     -
     3  ONLINE     -     -     -     -     -     -     -     -
     4  ONLINE     -     -     -     -     -     -     -     -
     5  ONLINE     -     -     -     -     -     -     -     -

Cluster 1 has 6 logic drawers            CCD: ONLINE
    LD  Status    D0    D1    D2    D3    D4    D5    D6    D7
     6  ONLINE     -     -     -     -     -     -     -     -
     7  ONLINE     -     -     -     -     -     -     -     -
     8  ONLINE     -     -     -     -     -     -     -     -
     9  ONLINE     -     -     -     -     -     -     -     -
    10  ONLINE     -     -     -     -     -     -     -     -
    11  ONLINE     -     -     -     -     -     -     -     -

Rack 1 has 2 clusters
Cluster 3 has 6 logic drawers            CCD: ONLINE
    LD  Status    D0    D1    D2    D3    D4    D5    D6    D7
    18  ONLINE     -     -     -     -     -     -     -     -
    19  ONLINE     -     -     -     -     -     -     -     -
    20  ONLINE     -     -     -     -     -     -     -     -
    21  ONLINE     -     -     -     -     -     -     -     -
    22  ONLINE     -     -     -     -     -     -     -     -
    23  ONLINE     -     -     -     -     -     -     -     -

Cluster 4 has 6 logic drawers            CCD: ONLINE
    LD  Status    D0    D1    D2    D3    D4    D5    D6    D7
    24  ONLINE     -     -     -     -     -     -     -     -
    25  ONLINE     -     -     -     -     -     -     -     -
    26  ONLINE     -     -     -     -     -     -     -     -
    27  ONLINE     -     -     -     -     -     -     -     -
    28  ONLINE     -     -     -     -     -     -     -     -
    29  ONLINE     -     -     -     -     -     -     -     -


X: Domain does not exist
D: Domain is disabled
-: Domain is available
<num>: Domain is downloaded or reserved



Target Pod Information:
Rack 1:
All length 30 HDSB: T24, T25
All length 5 T-Pods: T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10
All length 30 T-Pods: T26, T27, T28, T29, T30, T31, T32, T33
Available T-Pods: T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T24, T25, T26, T27, T28, T29, T30, T31, T32, T33
Locked T-Pods: NONE
Reserved T-Pods: NONE
Unavailable T-Pods: NONE

Exit code: 0
```


© 2021 All Rights Reserved.

a. In the *dbEngine Host* field, you can explicitly specify a different workstation for running DBEngine. By default, if you have specified an emulator, DBEngine runs on the local workstation.

Note: InfiniBand connection is required from the host to the emulator.

b. You can also launch the simulator on a different workstation — specify the workstation in the *Simulator Host* field. By default, the simulator runs on the local workstation.

c. In the *Board* field, you can specify the name of the board preference (`.bp`) file. The `.bp` file is useful in identifying the specific boards (logic drawers) or domains to which you want to download the design.

d. In the *!Board* field, you can specify the name of board restriction (`.br`) file.

Note: If you do not have either of the `.bp` or `.br` files, enter the board numbers or domain numbers to which the design must be downloaded. Board or domain numbers should be separated by spaces.

e. If the download of the design on the specified emulator fails, the xeDebug tool retries to download for the number of times specified in the *Retry* field and at the time interval in seconds specified in the *Interval* field. By default, the interval is set to 10 seconds.

f. Click the *I/O Assignments* button to open the window for assigning target locations.

To skip download of all target cables, select the check box adjacent to the *Target: Skip All* field.

Skipping download of cables allows you to download a design to a set of domains that lacks some of the target cables specified in the compile. If you skip downloading a cable, the behavior of inputs and bidirectional I/Os assigned to that cable is unspecified, and the pins of that cable (if it is present) do not drive the target.

To tri-state all target pods, select the *Tri-state I/O* check box.

For details refer to the Using xeDebug GUI to Select Targets at Run Time section in the *Palladium Target System Developer's Guide*.

g. You can also specify the *xrun Options*, if required.

h. To allow a design compiled in FullVision mode to run in Dynamic Probe (*DYNP*), select the check box adjacent to the *Vision Mode* field. This is particularly useful if increase in upload speed and maximum trace depth is preferred. However, note that switching to Dynamic Probe mode will not affect the emulation speed and step count.

© 2021 All Rights Reserved.

3. Click *Start Session* to begin the debugging session.

   The following buttons corresponding to the *Change To* field get enabled:

   - *Sim* (selected by default for Simulation Acceleration mode)
   - *Run* (for Signal-based Acceleration mode: *Emulation* and Simulation)
   - *TBrun* (for Transaction-based Acceleration mode)
   - *NBrun* (for advancing emulation and simulation time concurrently)
   - *Autorun* (for Emulation mode)
   - *Match* (for DUT in Emulation and Simulation while doing a equivalency match)

   Clicking *Show Swap Options* displays various hardware swap options — Figure 42-33. For more information, refer to Using Hot Swap Options section.

Figure 42-33 Hot Swap Options on Setup Tab

   Several fields get enabled when you open a session. In an open session, the *Start Session* button gets disabled and instead the *End Session* button gets enabled. The *Vision Mode* field gets disabled indicating that you cannot change to Dynamic Probes mode after the debug session starts.

   In addition, the other tabs of the xeDebug tool based on the design intent get enabled. For a description of each xeDebug GUI tabs, refer to Running the Design Using the xeDebug Tabs.

4. The *Restore last debug environment* button enables you to restore the debug environment from the last session.

   Note: In SA mode, this button is enabled only when the design is swapped to the emulator. In ICE mode, this button is enabled when the design is downloaded.

5. In the *SST2 (SHM) Wave Database* field, specify the name of the waveform database to be opened. If you specify a name that does not match an existing waveform database name, a new waveform database is created and the old waveform database is closed. Also, all probes from the old waveform database are copied into the new waveform database.

© 2021 All Rights Reserved.

To enhance waveform generation, you can generate waveforms in offline batch mode. The batch mode uses multiple threads and generates multiple files to improve the waveform generation performance. Specify the number of files that should be generated in the *NumFiles* field. For more information, refer to Generating Waveforms in Offline Batch Mode section and the description of the `database <database_name> [-batch [-numFiles <num>]` command in the *Run-Time Command*s chapter of the *VXE Command Reference Manual*.

6. In the *Save/Restart Cosimulation State* group box, specify an existing *Cosimulation State File Name* to restart the emulator using the settings of a previous design run. Otherwise, create a new file to save the cosimulation state for later use.

   This group box also has two check boxes, *Emulation State* and *Debug Env*. Selecting these check boxes indicates that the emulation state and/or debug environment needs to be saved or restarted. In the *Path* field, you can specify the path to save the simulation snapshot when running the designs in SA mode. By default, the snapshot is saved at the run directory.

7. For running the design, access the *Control* tab. For further details, refer to Control Tab.

Setup Tab in SA Mode for Offline Debugging

Selecting the *Offline* radio button on the *Setup* tab allows you to debug a design in the offline session, using only the database files and the waveform viewer; thus, freeing the emulator for other users. Debug utilities available in offline session include adding and displaying uploaded physical probes, calculating virtual probes, and bringing up waveform and schematic browsers.

Note: In an offline session in SA mode, you do not have the hot swap options to switch from the simulator to emulator or vice versa.

In offline session, the layout of the *Setup* tab is as illustrated in Figure 42-34 on page 1918. Follow the steps given below for debugging in an offline session:

1. In the *Offline Debugging* group box, specify the *Phy Data File* to use for debugging. You can use the directory button adjacent to the *Phy Data File* to select the required file with `.phy` extension.

2. Optionally, specify the library definition in the *cds.lib* field. By default, the file is located in `INCA_libs/worklib` directory.

3. Specify the design *Snapshot* to provide the connectivity-related information.

4. Click *Open Offline* to start the offline session.

© 2021 All Rights Reserved.

a. If you do not want to continue with the offline session at any time, click *Release* to release the offline host.

b. Clicking *Close Design* enables you to close the current design and select a new design to debug on the same offline host.

Note: The *SST2 (SHM) Wave Database* field gets enabled after you click *Open Offline*. Specify the name of the waveform database you want to open.

Figure 42-34 Setup Tab in Offline Debug Mode (SA Mode)

Using Setup Tab in ICE Mode

After opening the design in ICE mode, the *Setup* tab (as shown in Figure 42-35 on page 1919) allows you to:

- Specify whether you will start the debug session in *Online* or *Offline* session
- Define the *Configuration*s, such as an emulator.

The steps for defining the type of debug session and configuration are similar to what have been already discussed in Using Setup Tab in SA Mode. Therefore, only the steps that are different in ICE mode are discussed below.

Note: If you select to use the emulator in the *Offline* session, the layout of the *Setup* tab is different and steps discussed below are not required; instead, refer to Setup Tab in ICE Mode for Offline Debugging.

© 2021 All Rights Reserved.

1. Optionally, specify the library definition in the *cds.lib* field. By default, the file is located in `INCA_libs/worklib` directory.
2. Select the *Online* radio button to start an online session.

   You must initially download the database to an online session and create a trace file, before you can specify an offline session. If you have already saved a trace file from a previous online session, you can specify the trace file and open an Offline session.

   Note: To use the *InfiniTrace* tab, you must start debugging in an online session.
3. Specify the design *Snapshot* to provide the connectivity-related information.
4. In the *Configuration* group box, select the *Debug Mode*, that is, *STB* or *LA*.
5. Click the *I/O Assignments* button to open the window for assigning target locations.
6. Click *Host* to select the specified emulator.
7. Click *Download* to start the debug session.

   When the download is complete, the other tabs become accessible based on the design elements.

Figure 42-35 Setup Tab after Opening the Design in ICE Mode

After the design is downloaded to the specified emulator, several other fields get enabled.The *Release* and *Suspend* buttons get enabled on the *Setup* tab after design download. The *SST2 (SHM) Wave Database* field and the *Save/Restart Emulation State* group box has the function similar to the *Save/Restart Cosimulation State* group box on the *Setup* tab displayed in SA mode. For details, refer to steps 4 and 5 in Using Setup Tab in SA Mode. You

© 2021 All Rights Reserved.

can specify the compression levels to compress the saved design snapshot using the following options: `None` (that is, no compression), `Low`, `Medium`, or `High`. This is equivalent to issuing the following command: `save -compress [{Low | Medium | High | None}]`. For more information on compressing the saved database, refer to the Compressing Saved Design Snapshots in ICE Mode section.

Suspending and Resuming the Design Run Using the Setup Tab

The emulation session can be suspended at any time after downloading the design by clicking the *Suspend* button available on the *Setup* tab. After the design has been suspended from the emulator, you see a *Resume* button instead of the *Suspend* button. Click this button to restore the design state to the emulator.

In the xeDebug GUI, there is no need to set the `suspendEnable` run-time parameter. The session can be suspended anytime during emulation. You need not be running clocks to suspend the emulation.

For detailed information on the processing of the suspend and resume operations, refer to Using Suspend and Resume to Manage Emulator Resources section.

Setup Tab in ICE Mode for Offline Debugging

If you select the *Offline* radio button on the *Setup* tab, the layout of the tab is as illustrated in Figure 42-36 on page 1921. The steps performed while debugging in an offline session in ICE mode are same as in SA mode.

For details about the fields available on *Setup* tab in the offline session, refer to Setup Tab in SA Mode for Offline Debugging.

© 2021 All Rights Reserved.

Figure 42-36 Setup Tab in Offline Debug Mode (ICE Mode)

Running the Design Using the xeDebug Tabs

After the design has been downloaded, tabs other than the *Setup* tab are enabled as per their relevance in the design being run. For detailed information on the *Setup* tab, refer to "Starting a Debug Session" on page 1912.

The following sub-sections discuss the remaining tabs in detail:

- **Tabs available in both SA and ICE modes**
  - Control Tab

    The *Control* tab controls the running of a design.

    In ICE mode, it contains functions for InfiniTrace capture, forcing nets, triggering, capturing trace data, and monitoring status. Three versions of the *Control* tab in ICE mode are available: Vector Debug (Figure 42-41), Logic Analyzer, STB FullVision (Figure 42-37).

    In SA mode, it contains only the run controls and ability to specify the waveform database. Refer to Figure 42-38 (without uncontrolled clocks) and Figure 42-38 (with uncontrolled clocks).

  - Bookmark Tab

© 2021 All Rights Reserved.

The *Bookmark* tab enables you to define bookmarks for specific time points. You can use bookmarks as parameters to other commands. Bookmarks can be created either through the command line or using the *Bookmark* tab in xeDebug or vvmDebug. For details on how to create bookmarks, see Creating Bookmarks.

- SDL Tab

  Allows you to access the SDL event editor and file editor.

  - Using Advanced SDL Controls

- Probe Tab

  This tab lists all probes added at run time. Use this tab to add probes by net, instance, driver, or add physical probes. You can import probes from a schematic, save probes to a file, import a probe file, or remove probes.

- Memory Tab

  Controls memory read/write (that is, dump and load) and initialization operations.

- InfiniTrace Tab

  This tab allows you to save and restore the emulation status for a user-specified period of time. You can observe the emulation status, dump out memory content, get the value of any net in the design, and add probes to upload the waveform of any net.

  Note: In SA mode, this tab is enabled only when the design is swapped to the emulator. Also, InfiniTrace cannot be used in LA mode.

- Force Tab

  This tab allows you to force, set, and get values for individual nets or symbols. A symbol is one or more probes (signal) represented by a single name.

- Assertions

  This tab allows you to access the assertion functionality through the menu interface. The tab is divided into three sub-tabs: Config & Settings, Assertion Control, and Reports. For details, refer to Running and Debugging Designs with Assertions of the Chapter 27, “Compiling and Running Designs with Assertions.”

- RTL/Gate Tab

  Controls the RTL files and gate-level netlist debug functions.

- **Tabs available in only SA mode**

  - Deposit Tab

© 2021 All Rights Reserved.

- ❑ Break Tab

■ **Tabs available in only ICE mode**

- ❑ Clock Tab

  If the database is compiled using CAKE or SCM technology, the *Clock* tab lists all the clocks that can be changed at run time. You can use this tab to change the frequency or delay for a specified clock, or to change the oversampling ratio. The clock file can be saved for later import.

Control Tab

Once the debug session has been opened, the *Control* tab is displayed. The *Control* tab in STB mode is illustrated in Figure 42-37. The *Control* tab in SA mode is illustrated in Figure 42-38. When you have finished configuring the design in the other tabs, click the *Control* tab. You must set up the testing parameters using the other debug forms, before starting the design runs.

Note: In Simulation Acceleration, use `Ctrl+C` to stop an emulation run. This command will not stop a run initiated from the *Control* tab in ICE mode.

Figure 42-37 Control Tab (STB Mode)

© 2021 All Rights Reserved.

Figure 42-38 Control Tab (SA Mode)

The *Control* tab is composed of the following functional areas:

1. *InfiniTrace Prepare* group box is used to prepare for an InfiniTrace session. After an InfiniTrace session has been created, the InfiniTrace tab will be enabled.

   a. Specify the InfiniTrace *Session Name* and click the ✔ button to confirm the selection.

   b. Specify a *Bookmark* name, if required. For details, refer to Adding Bookmarks when Preparing InfiniTrace Session.

   c. The *Status* field displays whether the current session is *Opened* or *Closed*.

   d. Use *InfiniTrace Capture* radio buttons to turn *On* and *Off* InfiniTrace data capture.

   Note: For those debug modes where InfiniTrace Prepare is not available, the corresponding InfiniTrace tab is not functional.

2. *Force Nets* group box allows you to define two user nets.

   Note: This group box is enabled only in ICE mode. In SA mode, it is disabled even when the design is run on hardware.

© 2021 All Rights Reserved.

a. The nets that you have defined can be selected using the two buttons, *UserNet1* and *UserNet2*, which when clicked display the *Select User Defined Nets* dialog box as shown in Figure 42-39.

Figure 42-39 User-Defined Net Pop-Up

The toggle button adjacent to each user net definition button provides the following options – *R* to release the forced value. *0* to force the signal to a value of `0`, and *1* to force the signal to a value of `1`.

b. Specify in the *Net* field the signal and its accompanying value. The adjacent button specifies the format (binary, hex, octal, or decimal).

c. Clicking the *Force* button sets the state of the selected net to the specified value until explicitly released by clicking *Release*.

d. Clicking the *Release* button releases a previously-forced signal.

e. Clicking the *Get Value* button retrieves the current state of the specified signal.

3. *DUV Capture & Break Control* group box allows you to select trigger modes for SDL instances. If no trigger mode is defined, the design runs without any trigger conditions set.

Note: In SA mode, this group box is enabled only when the design is run on hardware. In this case, in addition to the fields explained below, an extra button, *Import Sim breakpoints*, is displayed (as shown in Figure 42-40.) For corresponding details, refer to Importing Breakpoints from the Xcelium Simulator into SDL Using GUI.

Figure 42-40 DUV Capture & Break Control Group Box in SA Mode (Hardware Run)

© 2021 All Rights Reserved.

a. Select either *Enable* or *Disable* radio button to control the *SDL Program*s that have not been specifically enabled or disabled using the *Advanced SDL Control* window.

b. Select either *With Delay* or *Without Delay* radio button to define whether delay needs to be added after a trigger occurs.

c. In the *Trigger Expression* field specify the event or logic expression for triggering by expression. The syntax and semantics of the expressions are the same as SDL expressions.

In STB mode, when this switch is selected, all the components of the SDL program defined in the SDL form are disabled without exception, even those that were separately enabled using the following XEL command:

```
sdl -enable <instance_names_list>
```

d. In the *Trigger Position (%)* field, specify a percentage to calculate the number of samples to capture after a trigger occurs. By default, it is set to 100%.

Note: This is a toggle field that allows you to select either of the following options: *Trigger Position (%)*, *Post Trigger Percent (%)*, and *Post Trigger Samples*. Use *Post Trigger Samples* to specify a value in terms of samples. Use either *Trigger Position (%)* or *Post Trigger Percent (%)* to specify a percentage of trace memory.

4. *Run Control* group box has different fields based on the mode in which the design is running.

- STB mode

a. Select the *Continue Run* check box to run the design continuously without resetting the SDL state from the previous run.

b. Select the *Break After* check box for the run to break after the specified time elapse. The time unit is displayed adjacent to the text field.

c. There are only two buttons to control the design – one to start the design run and other to stop it.

© 2021 All Rights Reserved.

- LA mode

a. The *Run Control* group box has only three buttons – start design run, stop design run, and start emulator clock.

- SA mode

Note: The *Stop Mode* field is visible only when a design has both controlled and uncontrolled clocks.

a. Select the *Continue Run* check box to run the design continuously without resetting the SDL state on execution of the `run` command.

b. Select the *Non-blocking Run* check box for the design run to be non-blocking.

c. Select the *Break After* check box for the run to break after the specified time elapse.

d. Select the *Controlled*, *Uncontrolled*, or *Capture/SDL* check box adjacent to the *Stop Mode* field.

This is equivalent to issuing the following command:

```
clockConfig -stopMode
        [controlled | *controlled | uncontrolled | both | none | *none]
```


© 2021 All Rights Reserved.

e. The *Run Control* group box has six buttons – start design run, stop design run, reset simulation, run one statement stepping into sub-program calls, run one statement stepping over any sub-program calls, and show current execution point.

5. *Trace Data Capture* group box provides fields to select how captured trace data is to be dealt with.

a. Select the *Continuous Upload* check box for continuous upload of the waveform database when you are going to open a database. The check box remains disabled when a database has already been opened.

You can also select the *Continuous Upload* check box on the Setup tab.

This is equivalent to issuing the following command:

```
database -open -continuousupload <database_name>
```

The *Auto Upload*, *Append Trace*, *Trace Memory Size*, *Prepare Offline*, and *Upload* options/buttons are disabled after starting continuous upload.

In SA and STB modes, continuous upload is supported. However, in SA mode, continuous upload is not supported for a design with uncontrolled clocks. In VD and LA modes, continuous upload is not supported.

Note:

- After continuous upload has started, you cannot execute the `database -prepareoffline` and `database -upload` commands manually.

  Also, you cannot execute the `configPM`, `reset`, `restart`, `clockConfig -stopMode`, and `infiniTrace` commands.

- To see probes on a waveform, you must add the probes before using a `run` command.

- By default, the *Upload* button on the Probe Tab, RTL/Gate Tab, and DUV Design Browser will bring up SimVision. For FSDB database, the *Upload* button will bring up the Novas waveform viewer, Verdi.

For details, refer to Generating Continuous Waveform in SA Mode.

b. Specify the name of the trace file in the *Wave DB Name* field.

Note: In SA mode, when the design is running on the software side, this is the only active field in the *Trace Data Capture* group box. The remaining fields discussed below are non-editable.

c. Select the *Probe SDL Nets* check box if the SDL signals need to be probed.

© 2021 All Rights Reserved.

d. Select the *Upload SDL Activity* check box to upload the SDL activities with the waveforms.

e. Select *Controlled*, *Uncontrolled*, or *Controlled With Event* from the *Capture Mode* box to control the waveform display for a design with uncontrolled clocks.

This is equivalent to issuing the following command:

```
database -open <name> -captureMode [controlled | uncontrolled] [-event]
```

f. Select the *Append Trace* check box to show trace data captured during the first run and add to it the data for the current run.

g. Select the *Auto Resume* check box to restart the run after data capture.

h. Turn on the *Auto Upload* option to automatically trace data, upload it and display the waveforms after each run. *Auto Upload* is set to *off* by default.

i. Select the *Auto Wave* check box to automatically display data in Waveform Viewer.

j. Specify the *DPA Sparse Sampling Ratio* to use sparse sampling mechanism while analyzing the power consumed by the design and click the button. Specify the sample ratio before running the emulation so that when the trace memory is uploaded, the data saved in the `*.phy` file is already sparse.

For detailed information about this mechanism, refer to Using Sparse Sampling with DPA in Chapter 32, “Analyzing Power Consumption of the Designs.”

Note:

- This field is valid for both ICE and SA modes. Selecting this field is equivalent to issuing the `dpa -sparse_sampling <N>` command.
- Sparse sampling can only be used in online sessions.
- Specifying the ratio as `0` or `1` disables sparse sampling.

© 2021 All Rights Reserved.

- The *InfiniTrace tab* also has the *DPA Sparse Sampling Ratio* field. The field on both the tabs shows the same value. Therefore, changing the ratio on any one tab causes the change on the other tab as well.

k. Specify the *Trace Memory Size*. You can set the value to be interpreted in terms of the specify unit or percentage.

l. Click *Upload* to manually upload the trace data to the Waveform Viewer.

m. The *Prepare Offline* button is enabled only in FV mode. Clicking this button creates a file for later use in offline session.

n. Click *Clear Trace* to delete the trace data from the current trace file.

6. The fields to display current status of the test run are different based on the mode in which the design is running. In ICE mode, the *Emulator Status* group box displays current status of the test run. In SA mode, the *System Status* group box displays current status of the test run.

The fields displayed in ICE mode are listed in Table 42-2. The fields displayed in SA mode are listed in Table 42-3.

Table 42-2 Emulator Status in ICE Mode

Emulator Status	Status Indicated
Op Status	The operation status of the emulator, such as Ready, Running, or Done. **Note:** This status is not visible when the design is running on the software side in SA mode.
Emulator Clk	The status of the trace clock, such as Running or Stopped. **Note:** This status is not visible when the design is running on the software side in SA mode.
Current Time	The current cycle time of design run.
Current Cycle	The current cycle number.
Trigger Pos	The cycle where the trigger occurred.
Cycles Run	Number of cycles since the last stop.

© 2021 All Rights Reserved.

Emulator Status	Status Indicated
SDL Status field	The SDL status field at the bottom of the *System Status* group box provides a limited view of the contents, including whether the instance was triggered, current state, counters, and various information about the trigger, for the instances encountered. Double click on the window or click the *SDL Status Viewer* button below the field to bring up the *SDL Status* pop-up window containing the complete status.

Table 42-3 System Status in SA Mode

System Status	Status Indicated
Simulator Status	The status of the controlled clocks of the simulator, such as Running or Stopped.
Simulator Time	The cycle time of the controlled clocks of the simulator.
Uncontrolled Clocks	The status of the uncontrolled or free-running clocks, such as Running or Stopped. **Note:** This status is only visible when the design has uncontrolled clocks.
Uncontrolled Clock Time	The cycle time of the uncontrolled or free-running clocks. **Note:** This status is only visible when the design has uncontrolled clocks.
SDL/Capture Status	The status of the SDL and capture clocks, such as Running or Stopped.
Trigger Pos	The cycle where the trigger occurred.

The *SDL Status* window contains information on the full list of trigger instances, and can be expanded as necessary. The text displayed below shows a sample of output you can see in this window.

```
# SDL Instances = 3
# Triggered Instances = 2
SDL Instance: I1 <Triggered>
     Current State: A1
     Counter1: 200
     Counter2: 300
     SDL Filename: trigger.tdf
     Trigger Line: 9
     Trigger Label:
```


© 2021 All Rights Reserved.

```
SDL Instance: I15 <Triggered>
Current State: NONE
     Counter1: 0
     Counter2: 0
     SDL Filename: trigger.tdf
     Trigger Line: 39
     Trigger Label: label1
SDL Instance: I6
     Current State: NONE
     Counter1: 0
     Counter2: 0
```

Control Tab in Vector Debug (VD) Mode

Vector Debug (VD) applies test stimulus to the design from a vector file and obtains response vectors. The vector file contains a list of vectors (one per cycle) where each vector defines the stimulus and expected results for that particular cycle. This mode is most suited for regression tests created from ATPG (Automatic Test Pattern Generation) software or for comparison with software simulation results. The *Control* tab for VD mode is displayed in Figure 42-41 on page 1932.

Figure 42-41 Control Tab (VD Mode)

© 2021 All Rights Reserved.

You can either call PTRAN as an independent executable to translate from a variety of stimulus file formats into Cadence's standard VBF (Vector Binary Format) file, which Vector Debug mode uses directly, or use the GUI (or the `vector` command in command-line mode) to call PTRAN to perform the translation.

Vector Debug mode cannot use a target connection.

In addition to specifying the number of vectors to run, Vector Debug mode allows you to use SDL as a breakpoint mechanism.

DRTL Tab

The DRTL tab in the xeDebug GUI enables you to define the DRTL modules and DRTL instances using Verilog, SystemVerilog, and VHDL files.

© 2021
All Rights Reserved.

Defining DRTL Modules

1. Open one or more source files. You can add more source files, if required. Multiple source files are separated by space.
2. In *DRTL Name*, specify the name of the DRTL module.
3. In *Top Module*, specify the name of the top module. This is equivalent to issuing the following command:

   `drtl -definemodule <DRTL_module_name> [-verilog | -vhdl | -sv] <file_list> [-topmodule <topmodule_name>]`

4. Click to define a DRTL module. The DRTL module will be added to the grid below the *Define DRTL* groupbox.

The following options are optional. Use these options according to your requirements:

- In the *Script* field, specify the compile script name for compiling the DRTL module.
- In the *Parameter* field, specify the parameter that needs to be passed to the DRTL module.
- Select the *Capacity* option to find the capacity of the DRTL module, if required.

Opening DRTL Source Files

1. Double-click *File* and select the source file to be opened.
2. Double-click the source file.

© 2021 All Rights Reserved.

Adding DRTL Instances

1. In *DRTL Name*, specify the DRTL module name.
2. In *DRTL Instance*, specify a name for the DRTL instance and click . The DRTL instance will be added to the grid.

   Note: You also need to connect the signals with the input ports of the DRTL instances.
3. Select the port name under the instance and enter the signal name in the textbox labelled with port name. This is equivalent to issuing the following command:

   ```
   drtl -addinst [-overwrite] <DRTL_module_name> <DRTL_instance_name>
   (.<port_name1>(<signal_name1>), .<port_name2>(<signal_name2>),...)
   ```
4. Similarly, add all signals for input ports.
5. Click *Apply* to add the DRTL instance.

Removing DRTL Modules and Instances

- Select the DRTL module to be deleted from the grid and click *Remove Module* to remove the module.
- Select the DRTL instance to be deleted from the grid and click *Remove Instance* to remove the instance. This is equivalent to issuing the following command:

  ```
  drtl -rminst <DRTL_instance_name_pattern> [-modulename <DRTL_module_name>]
  ```

  Note: You can also select and remove multiple modules and instances by pressing the `crtl` key.
- *Remove All*: Removes all DRTL modules.
- *Remove Inst*: Removes the selected DRTL instances.

© 2021 All Rights Reserved.

- *Hide All*: Hides the expanded list of DRTL modules.

Saving and Loading DRTL Modules

You can save the defined DRTL modules and use them later without recompiling the modules.

To save modules to a file:

1. In *File Name*, specify the file into which the DRTL module needs to be saved.
2. In *DRTL Name*, specify the name of the DRTL module.

   If a file with the same name already exists, the system will prompt you to confirm overwriting the existing file.
3. Click *Save* to save the DRTL module to the specified file.

   You can also specify a pattern to save the matching DRTL module names to a file. The equivalent command is as follows:

   ```
   drtl -save [-overwrite] <file_name> [DRTL_module_name_pattern_list]
   ```

   For example, you have three DRTL modules: `drtl1`, `drtl2`, and `drtl3`.

   The `drtl -save file1 drtl*` command will save `drtl1`, `drtl2`, and `drtl3` to file1.

To load modules from a file:

1. In *File Name*, specify the file from which the DRTL module needs to be loaded.
2. In *DRTL Name*, specify the name of the DRTL module to be loaded.
3. Click *Load* to load the DRTL module.

   The above steps are equivalent to issuing the following command:

   ```
   drtl -load [-overwrite | -ignore] <file_name>
   [DRTL_module_name_pattern_list]
   ```

Controlling System Tasks

By default, the system tasks are disabled for all instances. You can change the default settings to enable the system tasks from the *System Tasks* grid.

1. In *Instance Name*, specify the name of the instance.
2. Select *Enable* to enable the system tasks for the specified instance.

© 2021 All Rights Reserved.

SDL Tab

The *SDL* tab, shown in Figure 42-42 on page 1937, contains the *SDL Editor* group box that can be used to create and edit SDL (State Description Language) programs. The SDL programs can be saved to and restored from files by using the *Save* and *Load* buttons.

The *SDL Editor* group box provides drop-down menus, such as *File*, *Control*, and *Format*, that help you to build your SDL programs.

Note: You can use the right mouse button to drag and drop symbols/events from various forms, such as the *Symbol Editor*. Also, during the run, the current state of the loaded SDL file is highlighted.

For information on how to create SDL programs, refer to Chapter 43, "Controlling the Run with SDL."

Figure 42-42 SDL Tab

© 2021
All Rights Reserved.

Using SDL Editor Group Box

The following table lists the drop-down menus available within the *SDL Editor* group box:

Menu Item	Function
File	Provides you the following sub-menu items: ■ *New*: To create a new SDL program ■ *Open*: To open an existing SDL program ■ *Undo*: To revert any edits done to the SDL program. ■ *Redo*: To re-apply the edits done to the SDL program. ■ *Save*: To save the new/edited SDL program ■ *Save as*: To save the SDL program as a file with extension `.tdf` or `.sdl`.
Control	Allows you to specify SDL controls by selecting the required control from the following sub-menu items: *Instance*, *Include*, *State*, *If*, *Else If*, and *Else*
Expression	Allows you to select various keywords and operators for creating boolean conditions, such as {}, >, <, and &, for use inside IF statements.
Action	Allows you to select an action from the following sub-menu items: Goto, Load, Increment, Decrement, Start, Stop, Trigger, No_Acquire, Acquire, DISPLAY, TX, and EXEC
Global	Provides options used in creation of global definitions for TRIGGER, DISPLAY, TX, EXEC, conditional acquisition, and initial load values or count enable expressions for the two 40 bit counters.
Format	Allows you to change the indentation of SDL statements.

Using Advanced SDL Controls

Clicking the *Advanced SDL Control* button on the *SDL* tab displays the Advanced SDL Control window as shown in Figure 42-43 on page 1939.

© 2021 All Rights Reserved.

Figure 42-43 Advanced SDL Control Window (ICE Mode)

This window enables you to control individual SDL instances and SDL activities like probes, trace, and upload through the GUI from the following group boxes:

- *Individual SDL Instances of Current SDL Program* to enable or disable specific SDL instances

© 2021 All Rights Reserved.

- *SDL Trigger Labels* to enable or disable specific trigger labels
- *SDL Probe Control* for probing SDL nets
- *SDL Activity - Trace Control* to identify:
  - Conditions for trace, such as *Use conditional acquisition filtering*, *Trace during post triggers*, and *Trace every cycle*
  - Actions that cause SDL tracing, such as *Load*, *Trigger*, *Exec*, *Tx*, *Display*
- *SDL Activity - Upload Control* to identify when to *Clear SDL Trace*, such *OnRun* or *OnUpload*
- *Generate SDL Activity Trace*
- *Sim BreakPoints* to import Xcelium breakpoints to SDL during swap-in

  Note: This group box is displayed only when you are running the design in SA mode.
- *EXEC Control* to specify control conditions
- *DISPLAY Control* to specify control conditions
- *Halt On Trigger Control* to specify control conditions

Importing Breakpoints from the Xcelium Simulator into SDL Using GUI

In SA mode, to import the breakpoints from the Xcelium simulator into SDL, perform the following steps:

1. Access the *Setup* tab after starting a debug session.
2. Click the --> *Show Hot Swap Options* button from the *Configuration* group box to view the available options (refer to Figure 42-33 on page 1916). You have the following two check boxes that support the corresponding run-time commands for importing the Xcelium breakpoints:
   - *Error Ok* (for `sdl -simBreakpoints errorok 0|1` command)
   - *Auto import breakpoints* (for `xeset sdlAutoSimBreakpoints 0|1` command)

   Note: These options are also available in the *Sim BreakPoints* group box of the *Advanced SDL Control* window accessible from the *SDL* tab (refer to Figure 42-43 on page 1939).

© 2021 All Rights Reserved.

3. When running the design on hardware, click the *Import Sim breakpoints* button from the *DUV Capture & Break Control* group box to execute the `sdl -simBreakpoints -import` command (refer to Figure 42-40 on page 1925).

For detailed information on the importing of Xcelium breakpoints, refer to Importing Breakpoints from the Xcelium Simulator into SDL in Chapter 43, "Controlling the Run with SDL."

Probe Tab

The *Probe* tab provides the ability to create a probe on either a specific signal (or assertion), a specific scope (that is, all objects under the specified design hierarchy), the input cone of a net, or the nets along the logic path between two given nets, although the last two options are available for objects emulated only on the hardware. The term **probe** refers to the collection of probed objects grouped under a single probe name.

The *Upload* and *Display* buttons can be used to upload the data into the waveform database and display the probes in the specified waveform browser. If required, probe lists can be saved and imported using the *Save* and *Import* buttons.

Use the *Edit* button to modify a probe selected in the *Probes Display* group box. The probed objects grouped under the selected probe are displayed in the *Add Probes* group box. After modifying the probe, click the *Apply* button to submit the changes. If a probe name already exists, the old probe is first deleted before the new probe is added.

When you add a probe in SA mode, and it does not appear in the waveform window, check the XEL console section for `xmsim: *E, DBOBBD..` error associated with that probe. This error means that SimVision is unable to probe the specified object. If the following error message is generated, the specified probe will not be added and the corresponding signal will not be displayed in the waveform browser:

```
xmsim: *E,DBOBBD: cannot create SHM probe for : <signal_name>
```

For further help regarding this error, refer to the Xcelium documentation available on the Cadence Online Support website.

If you have probed a scope, you can enable *Include sub-scopes* and choose which sub-scopes you want to probe. As you make your selections, the *Probe* tab changes the scope view to indicate your selection. For example, if you choose `all`, all the hierarchical levels below the selected scope will be probed.

© 2021 All Rights Reserved.

Figure 42-44 Regular Probes

In SA mode, you can also add the probe objects that are simulated by the software and are not nets: Select the *Include Tasks* check box to include tasks in the probe; select the *Include Functions* check box to include functions. You do not need to select these check boxes if the selected scope is a task or function.

Creating Streaming Probes

You can create streaming probes to stream the waveform data while the data is being captured. Streaming probes use regular memories instead of DCC.

© 2021 All Rights Reserved.

Figure 42-45 Streaming Probes

The usage to create and upload the streaming probes is similar to that of the regular probes. Specify the streaming probe and the streaming waveform database name in the *Probe* and *Wave Stream DB* field respectively. The default waveform streaming database name is `sttrace`.

Note: Streaming probes only support signal names. Cone, path, and SDL file name are not supported.

The following options are available to control waveform streaming:

- *Duration*: Specifies the streaming duration. The timer is started at the beginning of the run and stopped when the time is up or clock is stopped. The *Duration* field is optional. You can specify the duration either in seconds or minutes. Select the time unit accordingly from the adjacent box.
- *Overflow*: Specifies whether or not to stop an ongoing run, or generate a message when an overflow condition occurs on streaming. An overflow condition occurs if data from the streaming memories is overwritten before it has been flushed to the waveform output file The default overflow mode is `error`.
- *Enable Stream*: Enables data capturing in streaming probe memories.
- *Disable Stream*: Disables data capturing in streaming probe memories.
- *Start Stream*: Turns on waveform streaming wherein waveform data is streamed using memories, and uploaded continuously during the run.
- *Stop Stream*: Turns off streaming. Waveform data is continuously captured into the streaming memories, and might be manually uploaded using the `database -stream -upload` command. This is referred to as the buffered mode.

© 2021 All Rights Reserved.

- *Waveform*: Adds the specified probes to the waveform immediately on clicking *Apply*.

The *Remaining* field shows the time remaining for waveform streaming in seconds. The time is updated every second. The *Available/Used/Total* fields show the number of available, used, and total streaming probes. The default total number is 1024, however, you can use the following compiler option command to specify the depth and size of streaming probes before compilation:

```
compilerOption -add {waveformStreamDepth <depth>}
compilerOption -add {waveformStreamWidth <width>}
```

For information about the `waveStreamDepth` and `waveformStreamWidth` options, refer to the description of the `compilerOption` command in the *User Data Commands* chapter of the *VXE Command Reference Manual*.

For more information on waveform streaming, refer to the following sections:

- *Streaming Waveform Data* section in the *Debugging Designs on Palladium Z1* chapter of the *VXE User Guide*.
- Description of the following commands in the *VXE Command Reference Manual:*
  - `compilerOption -add {waveformStreamDepth <depth>}` and `compilerOption {waveformStreamWidth <width>}` command in the *User Data Commands*.
  - `database -stream` and `probe -stream` command in the *Run-Time Commands* chapter.

Creating Probe Groups

You can define a probe group at run time through the xeDebug GUI. A probe group represents a group of signals that can be probed together. You can then request to probe a specific group by selecting its name from the *Probe Group* tab in the GUI. Probe groups can be defined for both normal and streaming probes.

You can also specify the signals in a text file and then upload the file to import the probe groups. If any changes are required in the text file, edit the file and then re-import the content into the GUI to overwrite the previously imported content.

Following is the supported format for the text file:

```
@ <group-name>
<object>
<object>
...
@ <group-name>
<object>
```


© 2021 All Rights Reserved.

```
<object>
...
```

The first line for each group definition represents the group name followed by the objects to be probed, one per line. The definition of the group name uses the @ character on the first column, followed by one or more spaces and the name of the group (denoted as <*group-name*>). The group name definition is followed by any number of lines each representing an object to be probed (denoted as <*object*>).

<*object*> can be any one of the following:

1. Full hierarchical name of a single net.
2. Full hierarchical name of an instance — in such cases, all the nets in the instance are probed.
3. An object can also be specified using the following syntax: `-depth` <*n*> <*instance*>

   <*instance*> is a full hierarchical name of an instance and <*n*> is a non negative integer. All the nets inside the specified instance will be probed for all nested instances up to the specified nesting level <*n*>. The syntax has the same semantics as the `probe -create -depth` <*n*> <*instance*> command.

   Note: For streaming probes, the `probe` command does not support using instance names as probed objects. Therefore, #2 and #3 for the object syntax are supported only with normal probes and not with the streaming probes. To add probe groups for streaming probes, you must specify the full hierarchical name of the nets in the probe group.

The following code shows an example of the text file content:

```
@  REGS
top.cpu.alu.regfile.ra[31:0]
top.cpu.alu.regfile.rb[31:0]
top.cpu.alu.regfile.rc[31:0]
@  CLKS
top.controller.video.clk1
top.controller.video.clk2
top.controller.video.clk3
top.controller.rsx.clk1
top.controller.rsx.clk2
@  MIX
top.cpu.alu.regfile.mclk
-depth 0 top.controller.video
-depth 5 top.controller.comm
```

Adding Probe Groups

1. In the *Group Name* list box, specify a group name that represents a set of signals to be probed.

© 2021 All Rights Reserved.

2. In the *Signals* list, specify the signal names and click to add the signal to the selected probe group. You can also drag and drop the signals from the *DUV Design Browser* or remove the added signals from the group.

After a group is added successfully, it is displayed in the *Display Groups* grid.

Figure 42-46 Probe Groups

Removing Probe Groups or Signals

You can also remove the probe groups or signals as required:

1. Select the groups or signals from the *Display Groups* grid.
2. Click *Clear* or *Clear All* as required.
3. Click *Apply* to apply the changes.

A warning message is always issued to confirm about overwriting the Probe group.

In the *Probe* list box, specify a name to add the selected group name for Normal or Streaming probes. This field is optional. This is equivalent to issuing the `-name` option to the `probe` and `probe -stream` commands for normal and streaming probes respectively. You can also use the group name as the probe name. If a probe name is not specified, xeDebug assigns a default probe name.

- To add signals from the selected group as normal probes, click *Normal*.
- To add signals from the selected group as streaming probes, click *Streaming*.

© 2021 All Rights Reserved.

- Select the *waveform* option to add the selected probe groups to the waveform. The objects in the probe groups will be added to the waveform display immediately after clicking *Normal* or *Streaming*.

  Note: If, however, the `waveform` option is left unchecked, SimVision will not be opened immediately, and the `probe -create` <*net_name*> <*net_name*>... and `probe -stream -create` <*net_name*> <*net_name*>... commands will be executed for normal and streaming probes respectively.

Saving and Importing Probe Groups

- In the *File* list box, specify the file name that defines the set of signals to be probed.
- Click *Save* to save the contents of the file or click *Import* to import the file content into the GUI.

Memory Tab

The *Memory* tab, as shown in Figure 42-47 on page 1948, controls memory read, write, and initialization operations.

Memory Tab in SA Mode

In SA mode, the *Memory* tab (Figure 42-47) enables the following functions:

1. *Memory Action Edit* group box enables you set the parameters for editing a memory

   a. In the *Memory Location* field, select whether the edit operation should apply to *all* memories, *tb* (testbench) memories only, or *dut* memories only.

   b. Specify the *Memory Instance* to be operated upon.

   c. Specify in the *File* field a memory content file for a specific memory instance.

   d. Specify a memory *Format*. This field is a listbox. Click the down-arrow on the right of the field to view and select an available format.

   e. Specify in the *Start Address* and *To* fields the range of memories to be read. Enter hexadecimal values. If these fields are left blank, the entire memory is read. Specify auto, hexadecimal, binary, octal, or decimal format in the *Format* field.

   f. Select from the *Memory Action* drop-down the action that you want to be performed on the selected memories. Use the button to commit the memory contents file to the selected memory instance.

© 2021 All Rights Reserved.

Note: The actions listed in the *Memory Action* drop-down are the same as the radio buttons adjacent to the *Display* field of the *Memory Action Control* group box.

2. *Memory Action Control* group box allows you to set the display and executes the selected action.

   a. The selected *Display* option determines the content of the display fields in the table: *All*, *Dump* (memories to be read), *Load* (memories to write), *Reset*, *Set*, *SetValue*, and *SetRandomValue*.

   b. Select the *AutoView* check box to bring up the *Memory Content* window (as shown in Figure 42-48 on page 1949) when *Apply* is clicked. This window allows you to directly edit the memory contents, and either write them to memory or save the file for later use.

   Note: *AutoView* is a FV option for dumping the memory contents list to a file.

   c. Click *Apply* to enforce the selected memory control action.

Figure 42-47 Memory Tab (SA Mode)

   d. Click *Save* to bring up the dialog box that enables you to save the memory list to a file.

   e. Click *Import* to bring up the dialog box that enables you to load memory data from a previously-saved file.

© 2021 All Rights Reserved.

f. Click *Clear* or *Clear All* to clear either the select memory or the entire list of memories, respectively.

Figure 42-48 Memory Content Window

Memory Tab in Vector Debug Mode

In Vector Debug mode, the *Memory* tab (Figure 42-49 on page 1950) is slightly different from the one accessible in SA mode. This section lists only the differences. For details about the common fields, refer to Memory Tab in SA Mode.

1. The *Memory Action Edit* group box does not have the *Memory Location* field.
2. The *Memory Action Control* group box has the following additional fields:
   - **a.** The *Display* field provides the following radio buttons for selection: *All*, *Dump* (memories to be read), *Load* (memories to write), *Reset*, *Set*, *SetValue*, *SetRandomValue*, and *Init*.
   - **b.** Select the *Skip Zero* check box to not include the zero values in the memory dump.
   - **c.** Select the *Progress bar* check box to disable the progress bar.

© 2021 All Rights Reserved.

Figure 42-49 Memory Tab (Vector Debug Mode)

Memory Tab in STB and LA Modes

In STB and LA modes, the layout and fields of the *Memory* tab (Figure 42-50 on page 1951) are similar to Vector Debug mode, except for the *Init* radio button adjacent to the *Display* field in the *Memory Action Control* group box. This radio button is not available in the STB and LA modes.

For details about the common fields, refer to Memory Tab in Vector Debug Mode.

© 2021 All Rights Reserved.

Figure 42-50 Memory Tab (STB Mode)

InfiniTrace Tab

First, you need to name and prepare a session through the *Control* tab. After the session has been prepared, switch to the *InfiniTrace* tab that allows you to save and restore the emulation status for a specified time.

Prepare Session Mode

1. You must be online to prepare a session.
2. Switch to the *Control* tab.
3. Specify a *Session Name* for your InfiniTrace prepare session. If an existing session is specified, an error is returned. All InfiniTrace data will be stored in this session name. Use the ✔ button to open a session.

© 2021 All Rights Reserved.

Initially, the *Status* for *InfiniTrace Prepare* group box shows as *Closed*. It changes to *Opened* after the session is opened. You cannot delete and open a session. Close the session before deleting it.

Figure 42-51 Preparing InfiniTrace Session on Control Tab

4. Select the *InfiniTrace Capture: On* or *Off* option to explicitly enable or disable data capture. After the *Prepare* session is opened, data capture is enabled by default.

   Note: The *On* and *Off* radio buttons are enabled only after the InfiniTrace is opened.

5. To observe the InfiniTrace data captured during the *Prepare* session, switch to the *InfiniTrace* tab.

6. Use the InfiniTrace Display functions to upload trace data to the waveform viewer.

Note: If the *InfiniTrace Capture* radio button is set to off, you will not be able to see data for that session.

Running the design is disabled while you are in an InfiniTrace *Observe* session, which is controlled through the options in the InfiniTrace tab.

© 2021 All Rights Reserved.

Adding Bookmarks when Preparing InfiniTrace Session

The following actions can be performed while bookmarking the InfiniTrace session:

- Specify the name of the bookmark in the *Bookmarks* field within the *InfiniTrace Prepare* group box on the *Control* tab and click the ✔ button.
- To remove a bookmark, specify its name in the *Bookmarks* field and click the button with a **X** cross icon.

Observe Session Mode

1. Click the *InfiniTrace* tab to access the InfiniTrace *Observe* session, after you have run a *Prepare* session.

   Note: The *InfiniTrace* tab is disabled when you attempt to download the design to host after release of the host, or in offline session. After exiting InfiniTrace *Observe* session, you can run the design in the normal mode.

Figure 42-52 Observing InfiniTrace Session

2. Specify the name of the required *Observe* session and click the *Open* button.

© 2021 All Rights Reserved.

3. Go to a specified time in the *Observe* session.
4. Use the *Session Status* group box to observe the time and trigger data.
5. In the *InfiniTrace Run Control* group box, you can go to an existing bookmark by specifying the name in the *Bookmarks* field, or by selecting a bookmark from the drop-down list if you had added bookmarks while preparing for the InfiniTrace session.

Force Tab

The *Force* tab allows you to force, set, and get values for individual signals or nets. These values are used in design runs to control signal values.

Set holds the specified state for the specified symbol or net for one clock cycle only. This value is overwritten during the design run.

Force sets the state of symbol or net to the given binary value until explicitly released by a signal *Release* command.

In SA mode, the *Force* tab illustrated in Figure 42-53 on page 1955 is displayed. Whereas, in ICE mode, the *Force* tab illustrated in Figure 42-54 on page 1955 is displayed.

The *Force* tab in both the SA and ICE modes has the following buttons:

a. Click *Save* to bring up the dialog box that enables you to save the forced values of the listed objects to a file.
b. Click *Import* to bring up the dialog box that enables you to load forced data from a previously-saved file.
c. Click *Sort* to arrange the displayed data in a specific order.
d. Click *Select All* to select all forced values displayed in the *Display Objects* group box.
e. Click *Release Sel* to release the forced value of the selected object.
f. Click *Delete Sel* to delete the selected object.

© 2021 All Rights Reserved.

Figure 42-53 Force Tab (SA Mode)

Figure 42-54 Force Tab (ICE Mode)

© 2021 All Rights Reserved.

RTL/Gate Tab

The *RTL/Gate* tab allows you to debug Verilog and VHDL-based RTL files and gate-level netlists. It can be used in conjunction with the RTL Design Viewer for working with RTL files, or the Schematic Viewer for working with gate-level netlists.

This tab controls the import of gate-level netlists, or RTL files. Imported files can then be displayed in the Schematic Viewer or RTL Viewer.

Note: The *RTL/Gate* tab is enabled in SA mode only when FSDB database is selected, and not with SST2 database.

The *RTL Schematic Viewer* group box enables you to specify the *cds.lib* file and name of the *Snapshot* in the designated fields. Clicking the *Open RTL Viewer* button enables you to view the RTL source code or schematic display of the specified snapshot in the RTL Viewer.

The *RTL Schematic Viewer* group box is useful for RTL-level debug only. The schematic viewer supports mixed Verilog/VHDL. To view a mixed-language module, the Verilog and VHDL databases must be separately generated. The design is then viewed by providing the schematic database and top design name.

Figure 42-55 RTL/Gate Tab with RTL-Level Schematic Group Box Enabled

© 2021 All Rights Reserved.

When you select the *Generate Snapshot* radio button in the *Gate Level Schematic Viewer* group box, the remaining fields in the group box get enabled as shown in Figure 42-56 on page 1957.

Figure 42-56 RTL/Gate Tab with Gate-Level Schematic Group Box Enabled

Deposit Tab

The *Deposit* tab is available only in SA mode. You can use the options available on this tab (Figure 42-57 on page 1958) to deposit values for individual signals or nets. This tab also enables you to deposit values for testbench objects. When using the xeDebug GUI in ICE mode, the deposit operation is accessible through the *Force* tab illustrated in Figure 42-54 on page 1955.

Using the options on the *Deposit* tab, you can perform the following tasks:

- Choose whether the deposit should occur at relative time intervals or at absolute time points.
- To deposit values at regular time intervals, enable the *Repeat every* check box and specify the time interval.
- To stop depositing values after a specified time, enable the *Cancel after* check box and specify the time interval.

© 2021 All Rights Reserved.

- Enable the *Release any forces prior to depositing values* check box to release any forces before depositing the value.
- Choose the type of delay that you want to occur before depositing the value—transport delay, inertial delay, or no delay.

Figure 42-57 Deposit Tab (SA Mode Only)

Break Tab

The *Break* tab is available only in SA mode. You can use the options available on this tab (Figure 42-58 on page 1959) to define and control break points.

© 2021 All Rights Reserved.

Figure 42-58 Break Tab (SA Mode Only)

1. *Set Break Point* group box allows you to define parameters for break points.

   a. Specify a *Break Point Name*. This is an Optional field.

   b. There are the following four tabs to define the different parameters for the break point:

   - On the *Time* tab, you can specify a *Stop At Exact Time* or *Stop At Regular Intervals Of* value. You can also specify the unit of time using the drop-down button adjacent to the first field on the tab.

     When you specify the *Stop At Regular Intervals Of* value, the *Starting Now* drop-down list is enabled. Clicking this drop-down will display to you two more options – *Starting At Time* and *Starting Now and Including Time*. Specify the required value and select the unit of time using the drop-down list adjacent to the second field.

© 2021 All Rights Reserved.

Figure 42-59 Break Tab with Time Tab

- On the *Line* tab, you can specify a *Line* number and *File* name. In addition, you can select whether to *Stop In One Instance At Scope* or *Stop In All Instance Of Module/Unit*.

Figure 42-60 Break Tab with Line Tab

- On the *Signal* tab, you can specify the *Signal* name on which the break point should occur.

© 2021 All Rights Reserved.

Figure 42-61 Break Tab with Signal Tab

- On the *Condition* tab, you can specify the *Condition* on which the breakpoint should occur.

Figure 42-62 Break Tab with Condition Tab

c. Select the *Stop only if this condition also is true* check box if you have an additional condition. Use the field adjacent to this check box to specify the condition.

d. Select the *Execute the following commands at each break* check box if you have such commands. Use the field adjacent to this check box to specify the commands.

e. Select any one of the following conditions using the drop-down list:

© 2021 All Rights Reserved.

- ❍ Always Stop At Breakpoint
- ❍ Skip The First
- ❍ Continue Through Breakpoint

f. Select any one of the following conditions using the drop-down list:

- ❍ Keep Breakpoint
- ❍ Delete Breakpoint After

2. *Break Points Control* group box allows you to define parameters for break points.

a. Click *Clear* to delete the existing breakpoint that has been selected.

a. Click Clear All to delete all the existing breakpoint that has been selected.

Clock Tab

The *Clock* tab (as shown in Figure 42-63 on page 1963) is available only in ICE mode. It enables you to change the frequency or delay for a specified clock, or to change the oversampling ratio. The initial values result from the values entered into the fields of the *Clock* tab at compile time. By using *Control* tab, you can substitute the new values at run time.

© 2021 All Rights Reserved.

Figure 42-63 Clock Tab (ICE Mode Only)

1. *Clock Configuration* group box initially takes the values from the clocks specified at compile time. By entering new values, you can change the clock settings at run time, without having to recompile.

   a. In the Clock Name field specify the name to be assigned to the clock signal.

   b. Specify the clock *Frequency* in MHz and click the button to apply entered value.

2. *Clock Display* group box shows the frequency and delay values of the current clocks in the design, as well as the *Current Clocks Oversampling Ratio*.

   The *Actual Frequency* column displays the design frequencies based on the maximum emulator operating speed reported in the `xe.msg` log file.

3. Click *Save* to save the values you specified at run time.

4. Click *Import* to bring up the *Import* dialog box that allows you to import clock settings from a previously-saved file.

PartialFV Tab

Enables you to add instances for isolation for partial FV computation.

© 2021 All Rights Reserved.

Figure 42-64 Isolating Instances

1. Specify the instance to be isolated in the *Instances* field.
2. Click to add the instance.

   Note: You can also specify a partition to exclude from computation. To do this, select the *Exclude* option.

3. Click *Commit* to create isolation for the added instance.

Removing and Filtering Instances

You can remove the isolated instances or specify a filter criteria to view the instances in the grid.

- Select the instance(s) from the grid, and click *Remove* or *Remove All* as required to remove the isolated instance(s).
- Specify the filter criteria using the *Filter* field to view the instances in the grid. Wildcards are also supported.

© 2021 All Rights Reserved.

Sorting and Viewing Instances

Various display options are provided that enable you to sort the instances, and view other information such as the "*exclude*" status and percentage of boundary nets. Use these options according to your requirements.

Table 42-4 Isolated Instances and Boundary Nets

Option	Description
Instance	Displays the isolated instances in the design. Click this button to sort the instance names in ascending or descending order.
NumNets	Displays the number of boundary nets. Click this button to sort the boundary nets in ascending or descending order.
Exclude	Indicates whether an instance is excluded for partial FV computation.
Percentage	Shows the percentage of boundary nets. The percentage is equal to *NumNets*/*Total Nets*.

For more information, refer to the following sections:

- Section Reducing FullVision Overhead using Run-Time Partial FullVision.
- `database -partialfv -add <instance_name>` command in the *Run Time Commands* chapter of the *VXE Command Reference Manual*.

© 2021 All Rights Reserved.

Saving/Restoring Debug Settings

Most of the tabs in the xeDebug GUI allow you to save and reload their content. This feature enables you to keep multiple versions of the settings that can be reloaded while debugging.

The following tabs/windows allow you to save and reload data:

- Symbols and Values
- Probe
- Clock
- Force
- SDL
- RTL/Gate
- Memory Initialization
- Memory Write
- Memory Read
- Memory Contents

Using Expressions in the Filter Field

The filter field on any tab/window allows use of glob- or regexp-type expressions to show data matching the specified filter string. The selection of glob- and regexp-type expressions is made based on the *Filter* tab of the *Preference* window.

When using glob matching, a wildcard (*) selects all related data.

When using regexp matching, period wildcard (.*) selects all related data.

Saving Data Entered on xeDebug GUI Tabs

Clicking the *Save* button on a particular tab displays the *Save File* dialog box illustrated in Figure 42-65.

© 2021 All Rights Reserved.

Figure 42-65 Save File Dialog Box

Use the fields and buttons on this dialog box to save the data into a file. When saving, use the recommended suffixes listed in Table 42-5 to easily identify the file types.

Table 42-5 Recommended Filename Suffixes

Form	File Type	Recommend Filename Suffix
Symbols and Values	XEL Script	`.sym`
Probe	XEL Script	`.upld`
Clock	XEL Script	`.clk`
RTL	XEL Script	`.rtlf`
Trigger State Editor	SDL program	`.tdf`
Emulation Control	Vector file	depends on vector format: `.in, .cbc`
Memory Initialization	Memory list file	`.mem`
Memory Write	Memory list file	`.mem`
Memory Read	Memory list file	`.mem`
Memory Contents	Memory list file	`.cont`

© 2021 All Rights Reserved.

The saved data can be restored or reloaded later by clicking either the *Import* or *Load* button on the tab. Clicking either of these buttons displays the *Import File* dialog box illustrated in Figure 42-66.

Figure 42-66 Restoring Form Data Example

1. (Optional) Enter a regular expression in the Filter field.
2. Click *Filter*.

Table 42-5 lists the recommended filename suffixes to use in regular expressions. By filtering based on the filename suffix, find the file you want in the *Files* list.

For example, the regular expression in Figure 42-66 is `*.mem`. After specifying the relevant expression, perform the following steps:

1. Select file to load.
2. (Optional) Enter a filename in the *Selection* field.
3. Click *OK*.

© 2021 All Rights Reserved.

Customizing the xeDebug GUI

The xeDebug GUI allows you to add customized elements, such as widgets, menu items, and tabs, and personalize the interface to meet your requirements. These customization are portable across Palladium Z1 releases; therefore, your customization remain unaltered even after tool upgrade.

The GUI customization made by you are maintained in predefined spaces because access to the Tcl and Tk environment of the xeDebug tool is restricted. Your code can access only the main Tcl interpreter.

The GUI customization can be made using either of the following methods:

- Using xegui XEL Command
- Loading libtk.so Tk Library (for Advanced Users)

To automatically load the customization when xeDebug GUI is launched at run time, you can create a script file named `.xeguirc` that can contain the `xegui` commands, or commands to load the `libtk.so` Tk library with native Tk commands. The `.xeguirc` file can be placed at any of the following locations for automatic upload:

- Palladium installation directory
- Home directory
- Design directory

Using xegui XEL Command

The `xegui` XEL command enables you manage the following aspects of xeDebug GUI customization:

- Creating the Custom GUI Elements
- Configuring the Custom GUI Elements
- Saving the Custom GUI Elements
- Invoking the Widget Commands
- Destroying the Custom GUI Elements

© 2021 All Rights Reserved.

Creating the Custom GUI Elements

You can create the following types of GUI elements:

- Widgets, such as buttons, check boxes, entry fields, frames (that is, group boxes), images, labels, list boxes, radio buttons, scales, spin boxes, and scroll bars.

  For a complete list of the supported types of widgets, refer to the description of the `xegui -create` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

- Menu items (Refer to Creating Custom Menu)
- Tabs (Refer to Creating Custom Tab)

Use the following `xegui` command and its options to create GUI elements customized according to your requirements:

```
xegui -create <name> [-type <type>] [-location <location>] [-pack <options>] \
      [<Tk widget_options>]
```

For more information on the `xegui` command and its options, refer to the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

The `xegui` command also accepts abbreviated commands, like `-cre` for `-create`, and `-com` for `-command`.

For example:

```
xegui -cre b1 -com "run"
```

The default location of each GUI element is at the top and the default type is button.

The Tcl interpreter where the commands associated with the widgets are executed is different from the main Tcl interpreter. The `xegui` command translates the `-variable`, `-textvariable`, and `-command` Tk widget options to hide the difference, but arguments escaped by curly brackets, { }, are substituted in another Tcl interpreter.

If the first character of a command is `@`, the command is executed in the console without echoing.

For example:

```
xegui -create btn -command {@ puts "no echo in console"}
```

Creating Custom Menu

You can create custom menu items of four types – command, separator, check box, and radio button.

© 2021 All Rights Reserved.

Creation of custom menu items as illustrated in Figure 42-67 on page 1972 require you to specify the name of the menu with the `-create` option of the `xegui` command.

To create a top-level menu displayed on the menu bar, you also need to specify the `-type` option as `menu` with the `-create` option. The following example creates a top-level menu, *Custom*:

```
xegui -create Custom -type menu
```

To create a submenu for a top-level menu item, ensure that the name of the menu contains the menu hierarchy labels separated by the greater than (>) sign. The `xegui` command automatically creates all hierarchy levels till the last. For example, use the following command to create a pull-down menu *Custom*, with submenus *Simulation* and *Start*:

```
xegui -create "Custom>Simulation>Start" -command {run}
```

Note: Creating top-level menu items is useful at the time of saving and destroying custom menu. To save or destroy an entire menu hierarchy, you just need to specify the name of the top-level menu item.

The check box and radio button menu items are recognized by the value of the `-type` option, which can be followed by the `-command` option used to associate an action with the menu item. The commands and separators do not require specification of the `-type` option. In other words, this translates the requirements for creating the supported types of menu items as follows:

Type of Menu Item	Defined By

Command	Specifying the `-command` option with the `-create` option.
	For example:
	`xegui -create "Custom>Simulation>Start" \`
	`  -command "run" -var mycheck`

Separator	Specifying only the `-create` option with the hierarchy labels separated by the greater than (>) sign. For example: `xegui -create Custom>Simulation>s1`
Check box	Specifying the `-type` option as `checkbutton` that can be followed by the `-command` option. For example: `xegui -create Custom>Simulation>Start -type checkbutton -command "deposit"`

© 2021 All Rights Reserved.

Type of Menu Item	Defined By
Radio button	Specifying the `-type` option as `radiobutton` that can be followed by the `-command` option. For example: `xegui -create Custom>Simulation>Start -type radiobutton -command "force"`

Figure 42-67 Creating Custom Menu

Creating Custom Tab

To add a custom tab, with the `xegui -create <tab_name>` command, specify the value of the `-type` option as `tab`.

Define all nested widgets that need to be placed on the tab using the `-pack` option with the `-in` sub-option. In addition, note that the nested widgets must refer to the tab by its name.

For example, the following set of command creates a tab named *My Tab # 1*:

```
xegui -create tab1 -type tab -label {My Tab}
```

Then, to add a group box labeled *Database and Work Directory* on *My Tab # 1*, use the following command:

```
set topf [xegui -create tab1.top -type ttk::labelframe \
         -text {Database and Work Directory} \
         -height 40 -pack {-in tab1 -fill x -padx 6}]
```

Refer to Figure 42-68 on page 1973 to see the tab and nested widget that gets created from the above commands.

© 2021 All Rights Reserved.

Figure 42-68 Creating a Custom Tab

Adding Vertical and Horizontal Scroll Bars

To add a vertical and horizontal scroll bar to the widget (as shown in Figure 42-69 on page 1974), with the `xegui -create <name>` command, specify the value of the `-type` option as `scrollbar` and use the `-apply` option as illustrated in the following example:

```
xegui -create tab1 -type tab -label {My Tab}
xegui -create frame1 -type ttk::labelframe -text {Edit} -pack {-in tab1 \
              -fill both -padx 6}
xegui -create scroll1 -type scrollbar -orient vertical -pack {-in frame1 \
              -side right -fill y} -command {xegui -apply txt1 yview}
xegui -create scroll2 -type scrollbar -orient horizontal -pack {-in frame1 \
              -side bottom -fill x} -command {xegui -apply txt1 xview}
xegui -create txt1 -type text -height 30 -pack {-in frame1 -fill both -side top}\

              -yscrollcommand {xegui -apply scroll1 set}
```


© 2021 All Rights Reserved.

Figure 42-69 Adding Horizontal and Vertical Scrollbars

Configuring the Custom GUI Elements

Use the following command to modify the options of an existing custom widget:

```
xegui -configure <name> <options>
```

All Tk options of the associated widget are supported.

For example, to configure a submenu named *Run* to run the design for 99 cycles when clicked, use the following commands:

```
xegui -config "My Menu>Run" -command "run 99"
xegui -config "My Menu>Run" -state disabled
```

Saving the Custom GUI Elements

You can also save the current state of a custom GUI element using the following command:

```
xegui -save <filename>
```


© 2021 All Rights Reserved.

Cadence recommends that you use the `xegui` command in interactive mode, then save the result, and later use it with minor modifications as the `.xeguirc` automatic initialization script.

Note: When you specify the `<filename>` as an empty string, such as `xegui -save ""`, the command returns a Tcl script containing the custom GUI elements created in the current session.

Invoking the Widget Commands

As the xeDebug GUI interface does not allow you to access Tk widgets directly, the `xegui` command provides the following option to invoke any widget's command:

```
xegui -apply <name> <widget_command>
```

The `-apply` option does not check `<widget_command>` specified after `<name>`. It passes the entire content in the following format to the Tk's widget directly:

```
<Tk path> <widget_command>
```

For example:

```
xegui -apply reset1 invoke
set txt [xegui -apply text1 get 0.0 end] ;# get all content from text widget
```

Destroying the Custom GUI Elements

Use the following command to destroy/delete the specified custom GUI element:

```
xegui -destroy <name>
```

For example:

```
xegui -destroy check1
```

To destroy all custom widgets, use the special name `all`.

Loading libtk.so Tk Library (for Advanced Users)

To build a GUI-rich environment, load the `libtk.so` Tk library directly to the XEL's Tcl interpreter either in script mode or interactive mode. Then, create your own Tk windows and widgets using the native Tk commands and having direct access to the XEL commands.

Note: All the Tk commands are supported. For more information on Tk widgets and commands, refer to the Tcl8.5.9 / Tk8.5.9 Documentation.

Following is an example of loading the Tk library from the xeDebug command prompt:

© 2021 All Rights Reserved.

```
load libtk.so
wm withdraw .
toplevel .mywin
```

You can also create a slave interpreter and load the Tk library there. However, for this, you will have to create aliases for all XEL required commands.

© 2021 All Rights Reserved.

43

Controlling the Run with SDL

This chapter describes the State Description Language (SDL). SDL is a run-time debugging feature that enables, based on design signal values, to control several aspects of the run such as when to stop, or when to capture probed data (tracing). Certain run modes enable to schedule execution of Tcl or XEL commands during a run using the EXEC facility.

Topics discussed in this chapter include:

- SDL Concepts
- SDL Preparation, Compilation and Execution
- Formatting and General Syntax Rules
- SDL Language Constructs
- Examples: SDL Program
- Importing Breakpoints from the Xcelium Simulator into SDL
- Creating Time Markers and Bookmarks
- Additional Tips

SDL Concepts

SDL is the trace control and break mechanism used by VXE. It can control data capture in In-Circuit Emulation (ICE) modes (both STB and LA modes) and Simulation Acceleration (SA) mode. It can stop the running design in Vector Debug (VD), STB and SA modes, and it can schedule execution of XEL commands during a run in STB and SA modes. SDL is specified as a program in human-readable text.

In VD, STB and SA modes, SDL will stop the run when executing a TRIGGER statement.

In Dynamic Target and LA modes, executing a TRIGGER statement typically stops only data tracing (probed data capturing). It is possible to override the default behavior by using XEL commands, and make the trigger stop the design clocks as well.

© 2021 All Rights Reserved.

Additionally, if using a run mode that supports *Conditional Acquisition*, SDL can control when to capture or not capture probed data, using `ACQUIRE` or `NO_ACQUIRE` statements.

In most run modes, an EXEC statement can be used to execute XEL commands while the run is in progress.

Essentially, SDL controls the functions of the emulator:

- When running in STB or VD mode, a trigger condition stops the design clocks and data tracing.
- When running in LA or Dynamic Target mode under IXCOM, a trigger condition stops data tracing by default, The default behavior can be modified to stop both design clocks and data tracing.
- When running in SA mode with a design that does not have uncontrolled clocks, a trigger condition stops the design clocks and data tracing.
- When running in SA mode with a design that has uncontrolled clocks, a trigger condition might either stop the controlled clocks, uncontrolled clocks, data tracing, or a combination of the above, depending on the settings specified with the `clockConfig` command.
- SDL can conditionally filter out traced data (Conditional Acquisition feature). The availability of this feature depends on the run mode.

SDL can print out formatted messages into a file and/or the console using the DISPLAY statement, or write transaction data into a SST2 database. Although `EXEC` can also print out messages by scheduling the Tcl command `puts`, the `DISPLAY` statement has advantages over using `puts` inside `EXEC`, which are discussed in Printing Messages from SDL with the DISPLAY Facility.

SDL provides a way to specify complex conditions, based on signal value patterns along the time axis. Unlike basic breakpoints (which are common in many simulators for stopping the run and are based on a single pattern of signal values) SDL can detect complex patterns along the time axis with relative ease. An example can be: trigger when signal X makes 1000 value transitions without staying at one value more than 2 consecutive cycles.

SDL programs can be quite complex by themselves. So, SDL comes with its own debugging facility called *SDL Trace Dump,* which enables to troubleshoot SDL programs using either the waveform display or text format. See Debugging SDL Operation with the SDL Trace Dump Facility for more details.

If you are running in SA mode (under IXCOM) and swapped the DUT from the software simulator into the Palladium hardware, breakpoints defined for the software simulator using the `stop` command will not function correctly after the swap if they use design objects from

© 2021 All Rights Reserved.

the DUT. With some limitations, you can import these breakpoints into an SDL program and use it on Palladium, provided that the breakpoints are of type `object` or `condition`, and use design objects only from the DUT. For more information, see Importing Breakpoints from the Xcelium Simulator into SDL.

SDL can leverage the programming power of VHDL and/or Verilog languages by connecting to the dynamic RTL (DRTL) instances.

Using DRTL Instances in SDL Expressions

SDL can use the values of the output ports of DRTL instances in expressions, anywhere where a design signal can be used. The DRTL instances are defined from the Verilog or VHDL source files, and compiled and downloaded into the emulator at run time. The inputs to DRTL are the design signals. This enables SDL access to complex operations that are available in Verilog or VHDL.

DRTL also has some built in system tasks that mirror SDL actions such as `acquire`, `exec`, `trigger` and `display`.

For more information on DRTL, refer to Controlling the Run with Dynamic RTL.

Multiple SDL Instances

SDL can independently track up to 64 different patterns along the time axis. Each pattern is tracked by a state machine (called an SDL instance), which has internal resources such as counters and expression evaluators. Each of the SDL instances operates independent of the others, and can independently stop emulation, control data capturing or invoke XEL commands through `EXEC` statements.

In simple cases, a single SDL instance is enough. A typical application of using multiple SDL instances is to trigger when any one of several possible sequences of events happen. Each SDL instance can be used to identify one of these sequences. Using this example (trigger when a signal makes 1000 value transitions without staying at one value more than 2 consecutive cycles), a set of three SDL instances can be used to detect this pattern independently on three different signals.

While each SDL instance can contain only one state machine (or none at all), it can also contain multiple number of global `EXEC`, `TRIGGER`, `DISPLAY`, `TX`, or `NO_ACQUIRE` statements in the Instance Definition Section (See State Machine versus Global trigger on page 1984).

© 2021 All Rights Reserved.

The number of different SDL instances available during run time must be declared to the compiler using the `sdlInstances` compiler option. If not specified, the compiler will allocate the resources for a single SDL instance.

Enabling and Disabling Individual SDL Instances

Each SDL instance can be enabled or disabled at run time (even while the emulator is running) independent of other SDL instances through the `sdl -enable` and `sdl -disable` XEL commands.

The software maintains two lists of SDL instance names, which are referred to as the ***Enabled Instances List*** and the ***Disabled Instances List***.

The Enabled Instances List is a list of all the SDL instances that are always enabled. An instance name can be appended to the list through the XEL command:

```
sdl -enable <SDL_instance_name>
```

The Disabled Instances List is a list of all the SDL instances that are always disabled. An instance name can be appended to it in a similar way using the `-disable` option.

We will refer to all the other SDL instances that do not appear in either the Enabled or Disabled Instances Lists as the ***Uncommitted Instances List***.

The Uncommitted Instances List, as well as all the `TRIGGER/DISPLAY/EXEC/TX/ACQUIRE/NO_ACQUIRE` constructs in Global Definition Section are enabled or disabled as a whole using one of the XEL commands:

```
sdl -enable
sdl -disable
```

For further details, refer to the `-enable` and `-disable` options of the `sdl` command in the *Run-Time Command*s chapter of the *VXE Command Reference Manual*.

When running under GUI, selecting *Enable* or *Disable* for the *SDL program* in the *Control* tab affects only the Uncommitted Instances List and the constructs in the Global Definition section, but does not affect instances in either the Enabled Instances List or the Disabled Instances List.

This mechanism was originally created to enable uninterrupted use of one SDL instance as a “system service” (for example, to unload a streaming memory at given intervals), while using another SDL instance for “regular debugging” (for example, to stop the emulator when a specific condition happens).

© 2021 All Rights Reserved.

Enabling and Disabling Individual Trigger, Display, TX, or Exec Statements

In addition to enabling or disabling complete SDL Instances, it is also possible to disable individual `TRIGGER`, `DISPLAY`, `TX`, or `EXEC` statements in any part of the SDL program, provided that they have a label. The system only maintains a single list of disabled labels, and any statement with a label that exists in that list is disabled. Labels are added or deleted from the list through XEL.

For more details and other command-line options related to this feature, refer to the `-disable -label` and `-enable -label` options of the `sdl` command in the *VXE Command Reference Manual*.

Conditional Acquisition Using DCC

SDL syntax provides two methods to define conditional acquisition, one using `ACQUIRE` statements, the other using `NO_ACQUIRE` statements. The two methods cannot be mixed in the same SDL program.

In the first method, probed data is acquired into the trace buffer in every cycle in which any `ACQUIRE` statement is active (which can be either an ACQUIRE action or a Global ACQUIRE).

In the second method, probed data is acquired into the trace buffer in every cycle, except cycles in which any `NO_ACQUIRE` statement is active (which can be either a NO_ACQUIRE action or a Global NO_ACQUIRE).

It is possible that one or two extra probe sample might be added to the end of each run, even if that time segment was not supposed to be captured due to conditional acquisition.

If the SDL program does not specify any `ACQUIRE` or `NO_ACQUIRE` in any SDL instance, the probed data is acquired every cycle.

Some operation modes, such as Vector Debug and infiniTrace, do not support conditional acquisition.

If the design was compiled for FullVision mode, probed data is acquired in whole frames, where a *frame* is 32 consecutive samples or less. A whole Frame is always acquired if the SDL program requests to acquire any sample within that frame. On the waveform display this might show up as extra samples that were originally requested to be filtered out.

© 2021 All Rights Reserved.

Conditional Acquisition for Streaming Probes

SDL provides conditional acquisition actions to probe with the SDL actions `ACQUIRE` and `NO_ACQUIRE`. You can also start or stop the acquisition of streaming probes independently of the DCC conditional acquisition using SDL.

To unlink SDL's acquisition of streaming probes from acquisition of DCCs, do the following:

In the SDL global definition section (above any `State` or `Instance` statements), add the following statements to control the streaming probes acquisition:

```
if(<expression1>) acquire stream stop;
if(<expression2>) acquire stream start;
if(<expression3>) acquire stream;
```

<*expression1*> <*expression2*> <*expression3*> must be legal SDL expressions.

These statements must be specified only in the SDL global definition section. An error message will be displayed if these statements are used inside an instance or inside a state.

By default, streaming probes start acquiring data from the first cycle of each run. If SDL is enabled, streaming probes stop data acquisition when <*expression1*> becomes true. From that point, streaming probes acquisition remains stopped. The acquisition resumes the data capture if the run is restarted (with a new `run` command), or if <*expression2*> becomes true. It will also capture in specific cycles in which <*expression3*> is true.

The SDL program might contain any number of the above statements.

Note: The `acquire stream` or `acquire stream start` actions have priority over `acquire stream stop` action. For example, if <*expression1*> and <*expression3*> are executed in the same cycle, then acquisition will go into a stopped state, however, a probe sample will still be taken in that cycle.

Example:

Suppose that the `signal cnt[15:0]` represents the output of a binary up counter. The following is an example of an SDL program using that signal:

```
if(cnt[15:0] == 100) acquire stream stop;
if(cnt[15:0] == 200) acquire stream start;
if(cnt[15:0] == 300) acquire stream stop;
if(cnt[15:0] >= 1000 && cnt[15:0] <= 1100) acquire stream;
if(cnt[15:0] >= 1300 && cnt[15:0] <= 1400) acquire stream;
```

In this example, assuming `cnt[15:0]` starts with value `0` at the beginning of the run, streaming probes will acquire data while the value of `cnt[15:0]` is within any of the following intervals:

© 2021 All Rights Reserved.

```
0 <= cnt[15:0] < 100
200 <= cnt[15:0] < 300
1000 <= cnt[15:0] <= 1100
1300 <= cnt[15:0] <= 1400
```

You can also select whether to use the `acquire stream, acquire stream start,` and `acquire stream stop` actions, or to ignore them, and instead link the conditional acquisition of streaming probes with that of the DCC probes. To do this, use the following XEL command:

```
xeset streamAcquireControl [linked | unlinked | auto]
```

For more information on the `xeset streamAcquireControl` command and its values, refer to the documentation of this command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

- `linked`: Streaming probes use the same conditional acquisition as DCC probes — that is, SDL actions `ACQUIRE` and `NO_ACQUIRE`, and the DRTL system tasks: `$qel("acquire")`, `$qel("acquire start")`, and `$qel("acquire stop")` are used. If the SDL program contains `acquire stream start` or `acquire stream stop` actions, then they are ignored.
- `unlinked`: Conditional acquisition for streaming probes are unlinked from DCC probing and only the following SDL actions are used to control the acquisition: `acquire stream`, `acquire stream start` and `acquire stream stop`.
- `auto`: The system automatically selects between `linked` and `unlinked` values depending on the SDL program. If the SDL program contains any of the actions: `acquire stream`, `acquire stream start` or `acquire stream stop`, then conditional acquisition behaves according to the `unlinked` value, otherwise, conditional acquisition behaves for the `linked` value. `Auto` is the default value.

If the `xeset streamAcquireControl` command is executed without any value, the command returns the current value, that is, `linked`, `unlinked`, or `auto`.

Example: Sample Script: Using linked, unlinked, and auto

Contents of run-time script:

```
run -swap
xeset streamAcquireControl unlinked
sdl -setfile sdl.tdf
sdl -enable
sdl -verify
database -stream -open open_st
database -open open_dcc
…
getTime
xeset streamAcquireControl linked
run 50
```


© 2021 All Rights Reserved.

```
xeset streamAcquireControl auto
run 100
```

In the above example, initially the streaming probes are unlinked from DCC. After the `xeset streamAcquireControl linked` command, the streaming probes are linked to DCC. When the `auto` option is used, the conditional acquisition is determined on the basis of the statements used in the SDL program — that is, if the SDL program contains any of the actions: `acquire stream` or `acquire stream start` or `acquire stream stop`, then conditional acquisition behaves according to the `unlinked` value, otherwise, conditional acquisition behaves for the `linked` value.

State Machine versus Global trigger

`TRIGGER`, `DISPLAY`, `EXEC`, `TX`, `ACQUIRE`, and `NO_ACQUIRE` statements can be specified in one of two methods. The two methods can be used together in the same SDL program.

The first method, called Global TRIGGER (or Global EXEC or Global ACQUIRE/Global NO_ACQUIRE) uses a basic boolean expression based on design signal values (or assertions) evaluated repeatedly every cycle. Such global expressions can appear either at the very top of the SDL program (Global Definition Section), or at the top of each SDL instance (Instance Definition Section).

The second method uses a state machine that can track complex patterns along the time axis. Each SDL Instance can contain at most one state machine. `Trigger` or other Actions are embedded inside the definition of the state machine. The state machine has two internal counters that can be used to count long sequences (up to 2^{40}-1). Other statements within the state machine can be used to control the state machine, for example, decide when to switch from one state to another, or when to load or decrement one of the two counters. Any Action can be executed based on boolean conditions made from design signal values or the outputs of the counters.

SDL General Purpose Counters

Each state machine has two 40-bit general purpose counters (named `COUNTER1` and `COUNTER2`), which can be set to auto-decrement every clock cycle, or can be incremented or decremented only in selected cycles. Each counter can be independently controlled and its output can be tested to compare the counter's value against any positive integer value between zero and (2^{40} -1).

The scope of each of these counters is the entire state machine. While they are shared among different states in a state machine, they are not shared between different SDL instances.

© 2021 All Rights Reserved.

Each counter can be loaded with a new value, which can be different from one LOAD command to another.

Note: Values for SDL counters can also be loaded dynamically from the command line by using the following command:

```
sdl [-ldcnt1 | -ldcnt2] [-instance <instance>] <value>
```

For more information, refer to the `sdl` command description and syntax in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Each of the counters decrements or increments by 1 each time when the statement DECREMENT or INCREMENT is executed. In addition, the counter can also be put in auto-decrement mode through the use of the START statement. When in auto-decrement mode, the counter will decrement by 1 each time when its Count Enable is high. The Count Enable is a boolean expression made of design signals.

Although the `DECREMENT` statement is honored also while in auto-decrement mode, the counter decrements at most by 1 each cycle.

Since the counters are state machine resources, they can only be used inside STATE Statements. This is true for both counter actions, such as DECREMENT, and Counter Conditions.

COUNT() Operator

SDL also supports an unlimited number of counters, each private to the expression that contains it. The output of these counters becomes true after counting a specified number of times in which a given expression is true. For example, the following will trigger after counting 35 times in which `net1` has the value `1`:

```
if(count(net1,35)) trigger;
```

The following will trigger after counting 50 clock cycles:

```
if(count(1,50)) trigger;
```

There is no limit to the number of these counters in an SDL program. The conditions under which these counters are reset are configurable.

Statement Usage and Execution Semantics of an SDL State Machine

A basic state machine construct is the STATE Statement, which defines one state of the state machine. Within each state, input conditions (based on values of design signals and/or the internal SDL counters) can be tested by one or more IF - ELSE statement, and various actions can be taken, based on the results of these tests.

© 2021 All Rights Reserved.

In each evaluation cycle, first all the conditions are evaluated, and then all the actions are executed concurrently. This means, for example, that the result of an action to decrement a counter in cycle N can be tested by a condition only in cycle N+1.

Some types of actions cannot be executed together. For example, two GOTO actions that both request to transfer control to two different states are conflicting, and cannot be executed together. Conflicts happen when any of the following types of actions are requested to be executed in the same cycle (within the same SDL instance):

- Two GOTO actions with different destinations.
- Any two different actions for the same counter, except the following two combinations. These two combinations are allowed and will be executed concurrently if encountered in the same cycle.
    - LOAD and START
    - LOAD and STOP

In case of a conflict between two actions, SDL picks the one that appears first in the SDL program (when scanning the text from top to bottom). The SDL compiler prints out warning messages about potential conflicts before execution begins.

If enabled, the SDL state machine continues to execute as long as the design in the emulator continues to run (or until the requested amount of trace buffer is filled, when running in LA mode).

SDL Clocking

When running in ICE mode, each SDL execution cycle is typically a single Fast Clock (FCLK) cycle. Design signals can toggle faster or slower than FCLK depending on the sampling ratio. This means that when running with sampling ratio of 1X or 0.5X, SDL is not guaranteed to detect all signal values. For more details, refer to Limitations when Running with 1x Sampling Ratio.

When running in SA mode, SDL evaluation cycles run every FCLK cycle in which any design signal makes a transition. While this guarantees that SDL will detect all signal transitions, there is no guaranteed correlation between SDL evaluation cycles and simulation time. Also, SDL might or might not detect transitional values on combinatorial nets, depending on the way the design was compiled. Additional limitations exist when running in SA mode with 1X clocking. See Limitations when Running with 1x Sampling Ratio for more information.

When a design is compiled in SA mode using IXCOM with an uncontrolled clock generator (free running), you can select at run time the clocks (controlled or uncontrolled) that need to

© 2021 All Rights Reserved.

be stopped by an event, such as SDL trigger or simulator breakpoint. For more information, refer to Running Designs with Uncontrolled Clocks Using xeDebug in SA Mode.

File Inclusion

A single SDL program can span several different files through the use of File Inclusion.

Aliases

The Alias facility enables to assign any boolean expression to a single name, and then use that name in different places in the SDL program instead of writing the whole expression.

Use of Tcl Variables

The value of any global Tcl variable can be used inside the SDL program using the `$var` or `${var}` syntax. The value of the variable is substituted once at the beginning of the run. For more details, refer to Using Values of Tcl Variables Inside SDL.

Assertions and CPF Objects

Expressions in SDL program can check not only values of design nets, but also the state of assertions and CPF objects. See Assertion State Conditions for more details.

Auto Probing

The auto probing facility enables to request from XEL to automatically probe every net or assertion that is actively used in the SDL program. Nets or assertions that appear in alias definitions that are not used actively in the SDL program are excluded.

Fast versus Accurate

By default, the SDL instrumentation is sampling the design signal values with a delay of up to 2 fast clock (FCLK) cycles. In STB mode, the delay is almost always two FCLK cycles, except rare cases (for example, on the first cycle of the run), where the delay can be less than two. Where possible, the system tries to hide the effects of this delay from you. For example, the delay does not affect requests to filter out probe samples in specific cycles through Conditional Acquisition (ACQUIRE or NO_ACQUIRE), thanks to special instrumentation that re-synchronizes the timing of the probe samples with SDL. Also, the cycle number of the

© 2021 All Rights Reserved.

trigger point as reported on the GUI or by XEL, or in an SDL trace dump is not showing the delay.

However, this delay becomes visible when the emulator stops because of a trigger, or when emulation clock are suspended during execution of XEL commands through the `EXEC` statement. If you request from XEL at that point to view any SDL status information (such as state name or SDL counter values) you will see that the status information matches the cycle in which the trigger happened according to the SDL program, but is lagging after the design signals by up to 2 FCLK cycles.

When running under IXCOM (in cosimulation mode), the 2 FCLK cycles might fall within the same time slot in which the condition that caused the `TRIGGER` or `EXEC` happened, or it might fall in the next time slot.

Another possible anomaly that can be observed is when the run is stopped because of an event outside of SDL. For example, reaching the specified amount of time to simulate, or an assertion that stops the run. In that case, an `EXEC` or `TRIGGER` meant to happen on the same cycle (or time slot in IXCOM) in which the run stopped, might not be processed.

It is possible to override this behavior and eliminate the delay (switching to what's called *accurate triggering*), but this comes with a speed penalty. In STB mode in such case, the emulator speed is only 1/3 of the regular speed. Switching between stopping the emulator with delay and without delay is performed by the XEL command:

```
xeset stopDelay 0|1
```

This command can only be executed while the emulator clocks are stopped (or suspended while executing an `EXEC` statement). A value of 0 means no delay between the trigger and the stop point (and running at 1/3 speed).

In SA mode, using `stopDelay 0` will typically have less impact on overall run speed, but this depends on the percentage of time used up by the software simulator. Also, it only guarantees that any `EXEC` or `TRIGGER` will cause the design to stop within the same time slot in which the condition that caused it happened, but it does not guarantee that the run will stop within the same FCLK cycle.

In some cases, one can avoid the delay without incurring the 1/3 speed penalty by switching from fast to accurate triggering only during strategic time segments in which triggering is expected. This is practically done by executing the `stopDelay` command itself from within SDL using the `EXEC` statements.

© 2021 All Rights Reserved.

Delayed execution of TRIGGER and EXEC under IXCOM

Under IXCOM, on top of the above delay of 2 FCLK cycles, `TRIGGER` and `EXEC` actions are executed only at the end of the simulation time slot after all the behavioral processes finished evaluation. This means that even if `stopDelay` was specified as `0`, the conditions that caused the `TRIGGER` or `EXEC` actions, might have already disappeared by the time the `TRIGGER` or `EXEC` are actually executed. This is true also for the SDL state itself. SDL will continue to process conditions, and possibly change states before the TRIGGER or EXEC request is finally executed.

SDL Instrumentation Effect on Critical Path in SA Mode

If the critical path contains the `xc_top.sdlStop` signal, it indicates that the critical path goes through both the SDL instrumentation and the co-simulation instrumentation. If this behavior causes excessive increase in step count, you can break this critical path by inserting a flip-flop between the SDL instrumentation and the co-simulation instrumentation. To do this, use the following command:

```
precompileOption -add sdlTriggerPathBreak
```

The following command removes the flip-flop in case you want to override a previous request to add the flip-flop:

```
precompileOption -rm sdlTriggerPathBreak
```

Insertion of the flip-flop causes an addition of one FCLK delay in the path. This may cause in some cases an additional delay in SDL actions: `TRIGGER` and `EXEC`, or DRTL system tasks: `$qel("trigger")` and `$qel("<CMD>")`, where <*CMD*> is a TCL or XEL command.

The critical path is reported by default in the `tmp/et3compile.msg` file.

For information on how to control critical path reporting, refer to the description of the `compile` and `getCriticalPathDesignNetNames` commands in the *Compile-Time Commands in ICE Mode* chapter of the *VXE Command Reference Manual*.

Debugging SDL Operation with the SDL Trace Dump Facility

SDL comes with a debugging facility called *SDL Trace Dump,* which enables to troubleshoot the behavior of SDL programs.

During the run, key information on the state machine component of each SDL instance is recorded into dedicated memory, which is arranged as a circular buffer. The buffer's depth is 16k samples, and newer data samples overwrite the oldest. The data is recorded only in *interesting clock cycles*. By default, an interesting clock cycle is one in which any SDL action

© 2021 All Rights Reserved.

(except `ACQUIRE` or `NO_ACQUIRE`) is executed. You can change the default through XEL and designate what cycles to record based on the type of actions that SDL executes. For example, you can request to record information only in cycles in which SDL executes a GOTO action, or on the other extreme, record continuously every cycle.

The recorded data includes information such as the emulation time, value of the two SDL counters, SDL state name, and what actions were executed by the state machine in that cycle. The simulation time or cycle number that appears in the recorded data for TRIGGER and EXEC actions show the time when the expression that triggered them became true. The actions themselves are typically executed with a delay of 2 FCLK cycles.

The data is always collected during a run, but you can decide through XEL whether or not to dump it to a file. When the data is printed in textual form, the information is most detailed and can include (based on XEL switches) even the filename and line number in the SDL file for each action executed.

There is a limitation regarding SDL trace dump in text mode: if a label (for a `TRIGGER`, `DISPLAY`, or `EXEC`) is disabled or enabled in the middle of the run (for example, with `sdl -disable -label -now` ...), the resulting SDL trace might not correctly show the actions associated with that label.

A waveform format is also available, but it shows only SDL state name and values for the two counters.

Recording of SDL activity into the SDL trace memory is stopped when a trigger happens. You can request that the tracing stop immediately upon trigger, or only when regular probing is stopped (as specified by the post trigger amount), depending on settings in the `sdl -traceOn` XEL command.

The trace is dumped to a file by executing the `sdl -traceDump` XEL command. When the command is executed, it dumps to the specified file SDL activity traced in the most recent run. If no trigger happened in the most recent run, the trace dump includes whatever data was captured so far.

By default, at the beginning of each new run, the SDL trace memory is reset, which means that any traced data from the previous run is discarded. However, you can request through XEL to reset the SDL trace memory not every run, but only when dumping the data to the disk, or even request to never reset the SDL trace memory.

Note that if a trace dump is requested while the emulator is running, SDL tracing is suspended while the data is dumped into the file.

When running from GUI, you can request to automatically create an SDL trace dump every run by selecting the `Upload SDL Activity` in the *Control* tab or *Infinitrace* tab. The waveform will show SDL activity only for the last run, although you can override this behavior

© 2021 All Rights Reserved.

in the `Advanced SDL Control` window (which you launch from the *SDL* tab). This window enables you to control other advanced aspects of SDL and SDL trace dump in specific.

Saving and Restoring the Internal State of SDL

The internal state of the SDL program can be saved into a file and restored later, either in the same session, or in a new session. The saved file includes information for each SDL instance, such as the name of the SDL instance saved, the name of the current SDL state, current values of SDL counters, and whether they were auto-decrementing or not.

Saving (or restoring) SDL state is typically done together with saving (or restoring) the emulated design state, to enable to return to a previously saved checkpoint, including the SDL state.

The save/restore mechanism only requires matching of state and instance names, therefore it can be used to restore the state of one SDL program into another SDL program. The only requirement is that the two SDL programs agree with names of instances and states.

Detection of Events without Triggering

If you only want to detect whether specific events occurred during a run, and do not want to stop the emulator (or data tracing) each time a specified event happens, use the `-onlyReport` switch of the `sdl` command. Here, the term event refers to a specified expression becoming true. When the `-onlyReport` switch is turned on, trigger conditions defined in the Global Definition Section or Instance Definition Section are disabled from stopping design or capture clocks, but they are still reported by the `sdl -getTriggers` and `sdl -report` commands. Triggers coming from the state machine portion of SDL are not disabled, and will cause the design and/or capture clocks to stop. For more details, refer to the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Limitations when Running with 1x Sampling Ratio

If the design was compiled to run with sampling ratio of 1x (Refer to *Appendix Clocking* in the *VXE Compile User's Guide* concerning `clockOption` technology CAKE), it is possible that a signal will make more than one transition within the same sampling cycle (one FCLK cycle).

If requested to test whether a signal is 0 or 1, SDL will pick the value at the end of the FCLK cycle. This means that SDL might miss every second value on the fastest changing signal.

© 2021 All Rights Reserved.

If requested to test for a special value such as a positive edge, SDL will scan the entire FCLK cycle to determine whether the condition is true. For example, to qualify for a positive edge, it is enough that the signal will make a low to high transition anywhere in the last FCLK cycle. To qualify for a “stable low” value, the signal must be low throughout the entire FCLK cycle. For general information on the use of special values in expressions, refer to Special Non-Numeric Constants.

Limitations when Running with 0.5x Sampling Ratio

If the design was compiled to run with sampling ratio of 0.5x, special values such as edge detection should not be used in the SDL program because SDL will not handle them correctly.

Testing for regular values (0 or 1) will sample signals only once at the end of each sampling period.

Interaction between SDL and XEL

SDL and XEL can interact in several ways.

First, in emulation modes that support this feature, SDL can issue any Tcl or XEL commands during emulation using the `EXEC` facility (See descriptions of EXEC and Global EXEC). From a syntax point of view, any string that can be executed by the Tcl interpreter, including calls to Tcl procedures is accepted. There are some XEL commands that are not allowed in an `EXEC`, irrespective of whether these commands are called directly from the `EXEC` or indirectly through a Tcl procedure. In run modes that enable stopping the clocks, the design clocks are suspended during the execution of the XEL commands to guarantee time coherency. When running in LA or Dynamic Target mode, the clocks are free running, time coherency is not guaranteed, which means that `EXEC` requests are executed with significant delay.

Second, XEL can control SDL, or retrieve from it information through the `sdl` XEL command.

Here is a quick summary of the most important facilities provided by the `sdl` command. You will find more details as well as command syntax in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

- Specify the name of the SDL file (the file that contains the SDL program).
- Verify the contents (for syntax errors) of an existing SDL file without actually requesting to run it.
- Enable or disable the SDL feature as a whole, as well as enable or disable individual SDL instances, or individual `TRIGGER`, `DISPLAY`, `TX`, or `EXEC` statements.
- Retrieve current status information such as:

© 2021 All Rights Reserved.

- ❑ Current state name and counter values for each of the SDL instances.
- ❑ Did a trigger happen, and if it did, in what cycle/time, and which statement (or statements) caused it.
- ❑ Check whether specific trigger(s) happened by providing the identifying label.

- Generate a log of recent SDL activity, called SDL Trace Dump (for example, state transitions, actions, counter values and so on) in either text or waveform format, and specify what information should be logged.
- Turn on/off automatic probing of all design signals used by SDL.
- Save and restore SDL internal state to/from a file.
- Control destination of `DISPLAY` and `TX` data.

Several other XEL commands, variables, and run-time parameters also affect the SDL execution as follows:

- Symbols (representing vectors), defined through the `symbol` XEL command, can be used in SDL expressions.
- The value of any Tcl variable declared at global call level (that is, by the `set` Tcl command outside any proc definition, or by the `global` command), can be used inside an SDL program with the `$var` or `${var}` syntax, where `var` is the name of the Tcl variable. Any numeric value or a character string can be passed that way from the Tcl script to the SDL program.
- Default values for post trigger samples or trigger position can be set by the `xeset` XEL command, and will be used by the SDL instrumentation if not set inside the SDL program.

Each time before starting SDL, the SDL program is recompiled if any of the following occurs:

- The main SDL file or any other file that it includes has changed.
- The value of any Symbol defined by XEL and used in the SDL program has changed.
- The value of any Tcl variable used by the SDL program has changed.
- Any one of the XEL commands that can affect the SDL run-time characteristics have been executed.

Typically, SDL is started by executing the `run` command. However, SDL will not be recompiled (regardless of any changes) if the run is started using the `run -continue` command.

© 2021 All Rights Reserved.

These rules also apply to the independent trigger expression defined by the `sdl -expression` command (although the independent trigger expression is not considered part of the SDL program).

DISPLAY and TX Facilities

The `DISPLAY` and `TX` facilities enable SDL to produce output during run time. Both facilities share the same hardware resources, which are independent of the hardware resources that are used for waveform capture. The data can be either streamed continuously while the run is ongoing, or it can be buffered and uploaded on demand similar to the way regular waveforms are handled. Data streaming is done in parallel to the run, and therefore can be used even with dynamic targets, provided that the average rate at which the data is generated is low enough to be processed without slowing down emulation.

Recording Transactions with the TX facility

The `TX` statement writes transaction type output into an SST2 database, which can be viewed later on SimVision's waveform browser or Transaction Stripe Chart Viewer.

The `TX` statement can define the beginning and end of a transaction. It can also define signal values that are associated with the transaction. The recording is done through Cadence's SDI interface.

Partial and Illegal Transactions

Transaction recording is not extended across consecutive `run` commands (unless the `-continue` option is used). This means that a transaction that has its beginning in one run, and its ending in a subsequent run will appear in the SST2 database as two partial transactions instead of one complete transaction (unless the second run was issued with the `-continue` option).

The existence of a partial transaction might indicate a protocol error in the design or an error in the SDL program, However, since it might also indicate a benign situation as explained, no warning is issued and each partial transaction is recorded in the SST2 database with a special attribute marking it as such.

Some transaction recording requests can be identified as definitely illegal. For example, two consecutive requests to record the beginning of a transaction using same transaction ID. This indicates either a protocol error in the design or an error in the SDL program. Illegal transactions are considered an error and will stop the run unless instructed otherwise from XEL.

© 2021 All Rights Reserved.

Printing Messages from SDL with the DISPLAY Facility

The DISPLAY facility enables to print out messages from an SDL program based on specified conditions. The message might optionally contain signal values and simulation time at the exact emulation clock cycle from which the `DISPLAY` statement was called. The message is formatted using a syntax similar to the Verilog `$display` system task.

You can also use EXEC to print message, however, there are disadvantages to using `EXEC` and these might even render it useless for some situations. EXEC enables to execute any Tcl command; so, messages can be printed using the standard `puts` Tcl command. For example, printing the value of the signal `ABC`, and the current simulation time using `EXEC` might look like this:

```
if(some_condition)
    EXEC ”puts \”value of ABC at time [getTime]is [value ABC]\””;
```

Using a `DISPLAY` statement to print the same information might look like this:

```
if(some_condition)
    DISPLAY (”value of ABC at time %t is %d”, ABC);
```

Comparing DISPLAY and EXEC Methods

Use of `DISPLAY` statements has the following advantages over `EXEC`:

- Simplicity: The syntax of `DISPLAY` statement is simpler. Note that when using `EXEC`, you need to escape the inner set of double quotes with a backslash.
- Speed: Each `EXEC` statement causes the design clocks to pause while the Tcl interpreter executes the `puts` statement. This operation typically takes several hundred milliseconds, and might cause severe slow down when the number of `EXEC` requests is large.

  `DISPLAY` messages are buffered during the run in a special *display buffer* memory, and therefore do not affect execution speed as long as the workstation can read the contents of the display buffer fast enough. Reading the display buffer is done in large blocks while the design is running, so in most cases, the `DISPLAY` statements have no effect on the execution speed once the run starts.
- Use with Dynamic Target: If the design clocks are not allowed to stop, as might happen when running a design in Logic Analyzer, Dynamic Target mode, or with uncontrolled clocks under IXCOM, `EXEC` requests can easily get overwritten and lost because they are not buffered.

  Display requests are buffered. Therefore, `DISPLAY` can be used in these operation modes without losing any messages. The only requirement for not losing `DISPLAY` messages is that the rate at which the workstation reads the trace buffer contents must

© 2021 All Rights Reserved.

be faster than the average rate at which the DISPLAY messages are generated by SDL. And if this is not possible, you have an option to request a large display buffer depth at compile time.

- Time coherency: Values of arguments of the DISPLAY statements are captured in the exact cycle that invoked the DISPLAY statement. Therefore, the printed values will match the condition that invoked them.

  Not so with EXEC. The EXEC statements are typically executed with some delay after the condition that invoked them. For example, in STB mode, the delay is 2 cycles or less (See Fast versus Accurate). In LA mode, the delay can be thousands of cycles.

There is one advantage for printing messages with EXEC over DISPLAY:

- SDL program load time: Arguments of the DISPLAY statements must be routed to the display buffer when loading the SDL program into the emulator. This might add to the time it takes to start a run command with a new SDL program. Therefore, if your SDL program prints the message only a couple of times throughout the debug session, it might still be faster to use EXEC instead of DISPLAY.

Emulation Resources Consumed by SDL

The typical overhead for each SDL instance requested during compile time is approximately 1MB RAM and 3k gates, irrespective of whether that instance is actually used during the run time or not.

The actual overhead can be larger when increasing the default values for some of the compiler options that affect the size of the display buffer or the SDL trace buffer.

The total size of the display buffer (in bytes) is determined by the following formula:

```
D * (G/8 + W/8 + 2*N)
```

Here,

- D is the depth as specified by compiler option sdlDisplayDepth.
- G is the value specified by compiler option sdlGlobalDisplay.
- W is the value specified by compiler option sdlDisplayWidth.
- N is the value specified by compiler option sdlInstances.

For example, the display buffer for a display depth of 4k, 64 global DISPLAY statements, a display width of 1024, and 3 SDL instances, will consume 640kB of emulated memory:

```
4096 * (64/8 + 1024/8 + 2*3)  = 581,632 (equivalent to 568 kB)
```


© 2021 All Rights Reserved.

The total size of the SDL trace dump buffer (in bytes) for all SDL instances is determined by the following formula:

```
T * N * 20
```

Here,

- `T` is the value specified by compiler option `sdlTraceDepth`.
- `N` is the value specified by compiler option `sdlInstances`.

For detailed information about the compiler options mentioned above, refer to the description of `compilerOption` command in the *User Data Commands* chapter of the *VXE Command Reference Manual*.

Forming SDL Conditions in IXCOM Mode

When running in STB mode, the number of times that the `net1=1` string is printed out remains the same from one session to another (excluding asynchronous events coming from user interaction or external targets in non-blocking runs).

On the other hand, when running under IXCOM mode, the number of SDL-evaluation cycles between design-state transitions is not guaranteed to be consistent from one release to another, or between different debug sessions.

For example, consider the following SDL statement:

```
if(dut.net1) display("net1=1");
```

In IXCOM mode, the relationship between design-clock transitions and FCLK is not guaranteed. Therefore, the evaluation rate of SDL is not guaranteed to be synchronized with any design clock. The outcome is that the number of times that this string is printed, might vary between releases or even between debug sessions.

To get consistent SDL behavior in cases like regression tests, it is recommended to use edge detection. This can be done in one of the following ways depending on the objective:

- If the objective is only to find when the `dut.net1` signal changes state from `0` to `1`, use the following:

  ```
  if(dut.net1 == 'bP) display("net1=1");
  ```

  The expression `(dut.net1 == 'bP)` becomes true only when `dut.net1` make a positive transition.

- If the objective is to report every clock cycle in which `dut.net1` is high, select a design clock net and use it to qualify the condition. For example, suppose `dut.clk` is a design clock:

© 2021 All Rights Reserved.

```
if(dut.net1 && (dut.clk == 'bP)) display("net1=1");
```

Here, one line is printed each time when `dut.clk` makes a positive transition, while `dut.net1` is high.

Expressions made from UPF, CPF, or assertion conditions should also use the above-mentioned guidelines.

For example, to trigger on an assertion entering the `finished` state, use the following:

```
if((dut.assert1 == "finished") == 'bP) trigger;
trigger;
```

Summary of Basic SDL features

- Up to 64 SDL instances, each can be enabled or disabled individually.
- Each SDL instance can have a state machine with up to 64 states.
- If -else statements for testing conditions in any nested configuration.
- Each SDL instance has two auto decrementing general purpose 40 bits counters, each controlled independently.
- SDL may have any number of private expression counters.
- Ability to enable and suppress probed data acquisition in probing modes that support conditional acquisition.
- Ability to stop the run in some operation modes.
- Ability to execute Tcl or XEL commands during emulation in STB, LA or Dynamic Target modes (`EXEC` facility).
- Ability to generate an activity log (SDL trace dump) in either text or waveform format per SDL instance.
- Ability to generate formatted output in the form of plain text or transactions written into an SST2 database (`DISPLAY` and `TX` facilities).
- State-less `TRIGGER`, `DISPLAY`, `TX`, and `EXEC` statements.
- Ability to request auto probing of all nets and assertions referenced in the SDL program.
- Ability to save and restore internal SDL state.
- Ability to disable individual `TRIGGER`, `DISPLAY`, `TX`, and `EXEC` statements using their label.

© 2021 All Rights Reserved.

SDL Preparation, Compilation and Execution

The following sections describe the way SDL programs are prepared, compiled, and executed.

Setup during Design Compilation

The maximal number of instances that can be used by an SDL program can be specified during design compilation by the compiler directive:

```
compilerOption -add {sdlInstances <num-instances>}
```

<num-instances> must be an integer number between 1 and 64. This number sets the upper limit for the number of SDL instances that can be used at run time. If unspecified, a value of 1 is assumed.

The maximal number of samples that an sdl trace dump can have can be specified during design compilation by the compiler directive:

```
compilerOption -add {sdlTraceDepth <depth>}
```

<depth> must be a positive integer number. This number can use a postfix of `k` or `M` (for example, 4k means 4x1024). If unspecified, a default value of 16k is used. Values above 16M or below 1k are converted to the maximal value of 16M or the minimal value of 1k.

Other than that, all other aspects of SDL are programmable at run time.

Preparing the SDL Program

During run time, SDL is prepared by a text editor and has to be stored in a file (or a set of files) before it can be executed. The SDL text is known as an *SDL Program*. The editor can be any text editor of your choice, but the recommended method is to use the *SDL Editor* group box under the *SDL* tab in the main xeDebug GUI window.

Using the *SDL* tab has the following advantages:

- It has a dedicated set of predefined buttons and menus for each of the SDL keywords.
- It highlights SDL keywords in a different color.
- It automatically verifies the SDL program for any errors and save it to a file before each run.
- It enables you to iterate through any errors found in the SDL program, while highlighting the location of the offending text on the screen.

© 2021 All Rights Reserved.

- It enables you to drag and drop signal names from other browsers (for example, the Design Browser or the Schematic Browser).

When running in the xeDebug GUI, specify the filename in the *Edit File* field in the *State Editor* group box and click the button adjacent to it.

Before starting a run (either through the xeDebug GUI or by executing the `run` XEL command), the SDL program is compiled into a special binary image downloaded into the Palladium hardware. The SDL compiler, which performs that process, can produce error or warning messages. Any error conditions reported by the SDL compiler must be fixed before the run request can be executed.

SDL Enabling, Activation, and Execution

Once the file containing the SDL program is defined with the `sdl -setFile` command, and SDL (or any one of its components) is enabled by the `sdl -enable` command, SDL is ready for use (considered *enabled*). Assuming that the SDL program does not have any errors, SDL is compiled into a binary image and loaded into the emulator when any one of the following XEL commands is executed:

- `sdl -load`
- `run`
- `infiniTrace -goto trigger`

`sdl -load` only loads the SDL image into the emulator, which enables you to query about basic properties of the SDL program (for example, what are the names of instances being used). This is done whether SDL is enabled or not.

`run` and `infiniTrace -goto trigger` load the SDL image then start both SDL and design clocks and/or capture clocks (depending on operation mode).

Once SDL is loaded into the emulator, it is considered ***active***, whether it is enabled or not. Note that a successful `sdl -load` command will always load the SDL program into the emulator and make it active, while a successful `run` or `infiniTrace` command will load the SDL program and make it active only if it is enabled.

An active SDL program can be enabled during a run, even if the run started with the SDL program disabled, therefore if you want to start the run with an SDL program disabled, and then enable it while the run is in progress, you need to execute the `sdl -load` command before executing the `run` command. In the current release, this will work only if you do not make use of the independent trigger expression (with the `sdl -expression` *<expressions>* command).

© 2021 All Rights Reserved.

You can query whether SDL is active or not through the `sdl -isActive` command. The return values of the `sdl -isActive` command only refer to the SDL program and do not consider the independent trigger expressions specified using the `sdl -expression <expressions>` command.

Formatting and General Syntax Rules

The following sections describe the way the SDL Preprocessor handles comments, file inclusion, text macros, and conditional compilation. In addition, these sections list the formatting and syntactical rules you need to follow in SDL programs.

Format

SDL is a free-format language. You can break statements across multiple lines. An end of line is considered a white space. You can also put multiple statements on the same line.

The syntax of the File Inclusion facility is an exception. It must be contained on exactly one line.

Statements are typically terminated by a semicolon. However, where several statements are grouped into a single block using braces, then the right brace is used as the terminating token.

Comments

Insert comments by using `/* */` or `//` notation. For example:

```
/*  This is a comment. This comment style
          can span multiple lines. */
// This is also a comment. This comment style
// restarts at each new line.
```

Comments cannot be inserted in the middle of tokens (for example, keywords, symbols, constants).

White Space

SDL ignores spaces, tabs, end of lines and empty lines (all considered as “white space”), except where used as separators between tokens. You must use white space between tokens in cases where it would otherwise be impossible to separate them.

© 2021 All Rights Reserved.

Identifiers

Identifiers are user given names, used by various statements and constructs in SDL. A few examples: Labels, Aliases, and State Names.

Identifiers are case sensitive, might contain only letters, digits and underscores, and might not start with a digit.

Case Sensitivity

All SDL keywords are case insensitive; they are recognized in either upper, lower or mixed case.

For example, `Trigger`, `trigger`, `TRIGGER` are all recognized as the same keyword.

Names that represent design signals, or other user given names (aliases, instance names, labels, and state names) are case sensitive.

File Inclusion

SDL programs can include text from other files similar to the way this is done in Verilog through the use of an `include` directive.

The syntax of an include directive is as follows:

`` `include "<file-name>" ``

The line cannot contain any `/*…*/` style comments or substitution of TCL variables (with the syntax `$<var>` or `${<var>}`).

The first character is a backward single quote mark, while the filename must be enclosed in straight double quotes. The `include` directive is not a free format: It must be specified on a separate line, and the `include` keyword must be specified in lower case. On the other hand, the include directive can be inserted at any point in the SDL program.

Included files can include other files to any nesting level.

The string `<file-name>` can contain either an absolute directory path (starts with the character '/') or it can contain a relative directory path, or just the filename with no path at all.

If `<file-name>` does not contain an absolute directory path, then it is searched in the current directory. If not found in the current directory, then it is searched in a list of directories specified by the run time parameter `sdlIncludePath`. The list must be space separated. For example:

© 2021 All Rights Reserved.

```
xeset sdlIncludePath {/home/george/myfiles/sdl /home/ben/projectX}
```

To make this search path visible to all your sessions with debug tools, whether running in script or GUI mode, you can place such definition inside the `.xerc` or `.rtxerc` files. The `.xerc` file is an XEL script that is always executed (if it exists) during the software initialization phase, while `.rtxerc` is executed on the initial call to the `configPM` command. It can be placed either inside the current directory from where you start the debug tool, or under the home directory.

Text Macros

Define text macros that can be used to represent any string:

```
`define <name> <string>
```

Following this definition, the form `` `<name> `` or `` `{<name>} `` are replaced by <*string*>

For example:

```
`define INST top1.top2.top3
if(`INST.net) trigger;
```

The above statement is evaluated as:

```
if(top1.top2.top3.net) trigger;
```

The following statement will produce the same result:

```
if(`{INST}.net) trigger;
```

Text macros can also be undefined as follows:

```
`undef <name>
```

Conditional Compilation

The following directives can be used to conditionally include or exclude certain parts of the SDL program:

```
`if `ifdef `ifndef `elseif `else `endif
```

Following are some typical ways of using these directives:

```
`if <value>
<SDL_statements>
`elseif <value>
<SDL_statements>
`else
<SDL_statements>
`endif
```


© 2021 All Rights Reserved.

Note that `'ifdef` and `'ifndef` can be used instead of `'if` to test whether a symbol was previously defined (or not defined) with a `'define` directive. It can also be used together with `'elseif` and `'else` directives.

For example:

```
'ifdef <name>
<SDL_statements>
'elseif <value>
<SDL_statements>
'endif
```

Each of the above directives must be specified on a single line.

Here, <*value*> must be one of the following:

- A constant integer number: If the value is non-zero, all the following SDL statements up to the next `'elseif`, `'else` or `'endif` are compiled. If the value is `0`, these statements are skipped.
- The form `$`<*var*> or `${`<*var*>`}`, where <*var*> is the name of an existing global TCL variable. The value of <*var*> must be an integer constant. This value is used as explained in the above bullet. Also, <*var*> might be an indexed TCL array like `abc(def)`.
- The form `'`<*name*> or `'{`<*name*>`}` where <*name*> was previously defined in a `'define` directive. The value that was previously assigned to <*name*> must be an integer constant. This value is used as explained in first bullet in this list.
- If <*value*> is none of the above, an error is issued.

Using the `sdl -pp` <*output_file*> command, you can generate a text file that shows the effect of applying the conditional compilation directives. Refer to the *VXE Command Reference Manual* for details of the `sdl -pp` <*output_file*> command.

Using Values of Tcl Variables Inside SDL

Values of global Tcl variables can be used directly inside the SDL program using either the *$<tcl-variable-name>* or *${<tcl-variable-name>}* syntax.

Any value can be substituted, whether numeric or a string. The second form with the `{}` is useful when the variable value has to be concatenated with another string without any intervening delimiter.

For example, suppose that a Tcl variable is assigned the value `top.alu`, which represents a hierarchy in the design, by executing the following Tcl statement (either from the XE prompt or from a script):

© 2021 All Rights Reserved.

```
set path top.alu
```

Suppose that the SDL program contains the following construct:

```
state s1 {
      if(${path}.y[7:0] < ${path}.x[7:0]) trigger;
      ...
```

The above construct is equivalent to:

```
state s1 {
      if(top.alu.y[7:0] < top.alu.x[7:0]) trigger;
      ...
```

Note that `path` must be declared in the Tcl script at global call level. This means that if the variable `path` is assigned inside the definition of a Tcl `proc`, then it must be declared explicitly as global as shown in the following example:

```
proc foo {args} {
     global path
     set path top.alu
     ...
```

The Tcl variables referenced with the above syntax inside an SDL program are substituted with their values only when the SDL program is loaded into the emulator (for example, at the beginning of each run). This does not apply to Tcl variables that appear inside EXEC actions. The construct *$var* or *${var}* inside `EXEC` actions are evaluated each time when the `EXEC` action is executed. See more details in the description of the EXEC action.

The substituted contents of TCL variables should not contain quoted strings, comments or other substitutions. This means that the following characters or sequences are not allowed in the substituted contents:

```
//
/*
`
$
"
```

SDL Language Constructs

Any of the SDL constructs explained below is optional. An SDL program can be as simple as a single statement of the form:

```
TRIGGER = <signal-name>;
```

Or it can be a complex program made of several SDL instances spanning on multiple files (through the use of the File Inclusion facility).

Figure 43-1 depicts an SDL program with its various constructs. All SDL keywords appear in the example in uppercase bold letters.

© 2021 All Rights Reserved.

Figure 43-1 SDL Program Example

```
ACQUIRE = top1.op == 'bp;
expr1 = top2.cnx[15:0] == 153;
t1: TRIGGER = expr1 || top1.rst == 0;
TRIGGER = top1.bx[7:0] < top2.by[7:0];
```

Global Definition Section

```
INSTANCE inst1;
expr3 = top.xyz.st & top.xyz.xx;
TRIGGER = expr1 && !expr3;
ENABLE2 = clk2 == 'bp;
```

Instance Definition Section

```
STATE s0 {
    IF(!expr1 || expr3) t2: TRIGGER;
    ELSE GOTO s1;
}
STATE s1 {
    IF(expr1)
        DISPLAY ("expr1=%d T=%t", expr1);
    ELSE GOTO s0;
}
```

State Machine Section

First SDL Instance

```
INSTANCE inst2;
expr3 = top.abc.tbs[15:0] > 17;
TRIGGER = expr1 && !expr3;
```

Instance Definition Section

```
ENABLE2 = clk3 == 'bn;
STATE s0 {
    IF(!expr1 || expr3) {
        LOAD CNTR2 100;
        START CNTR2;
    }
    ELSE GOTO s1;
}
STATE s1 {
    IF(CNTR2 <= 0) TRIGGER;
    IF(expr3) GOTO s0;
}
```

State Machine Section

Second SDL Instance

An SDL program can have a Global Definition Section shared by all instances, followed by definitions of SDL Instances.

© 2021 All Rights Reserved.

Each SDL instance starts with a INSTANCE Statement, and extends until the next Instance statement (or the end of the file).

Each SDL instance can have its own Instance Definition section (whose definitions are limited only to that instance), followed by a State Machine section.

If the SDL program has only one instance, then the Instance statement can be omitted, and the software will assign to it a default instance name.

The State Machine section is made of a series of STATE Statements.

SDL main constructs are grouped into the following categories:

- Expressions
- Operators
- General Definitions
- Global or Instance Definition Sections
- INSTANCE Statement
- STATE Statement
- IF - ELSE statement
- Actions

Table 43-1 lists the predefined keywords and operators that can be used in expressions. Expressions will typically use, in addition, names of design signals and assertions.

Table 43-1 Predefined Keywords and Operators for use in Expressions

Type	Keywords and Operators	Usage

Counter Names	`Counter1, Cntr1`
	`Counter2, Cntr2`
		Used in one of the following ways:
		`COUNTER1 <= 0`
		`COUNTER1 > 1000`
		`COUNTER2 <= 150`
		`COUNTER2 > 0`

© 2021 All Rights Reserved.

Table 43-1 Predefined Keywords and Operators for use in Expressions

Type	Keywords and Operators	Usage
Special Operators	{}	Concatenation operator. Create a vector from a list of scalars.
	[]	Bit-select or part-select of vectored signals.
	()	Parenthesis for changing evaluation order.
Unary Operators	transition()	Detects any value transition
	count()	Counts events in an expression.
	prev()	Acts as a delay. The result is the expression delayed by one clock cycle.
	~	Unary bitwise negation
	!	Unary logical negation
Boolean Operators	>	greater than
	<	less than
	>=	greater than or equal to
	<=	less than or equal to
	==	equal to
	!=	not equal to
	&	unary reduction AND, bitwise AND
	^	bitwise XOR
	|	Unary reduction OR, bitwise OR
	&&	logical AND
	||	logical OR
Assertion state values	"active"	Used as in the following way: <*assertion-name*> == "finished"
	"inactive"	
	"finished"	
	"failed"	
	"off"	

© 2021 All Rights Reserved.

Table 43-2 lists the predefined keywords that can be used in the Global Definition Section or the Instance Definition Section. The list does not include keywords that can be used in expressions, and are listed separately. The keywords listed as General Definitions (scope, enum, tdef) can also appear between state statements.

Table 43-2 Predefined Keywords in a Global or Instance Definition Section

Type	SDL Keyword	Usage
Definitions of Numeric Values	`counter1, cntr1`	Specify default load value for one of the two counters.
	`counter2, cntr2`	
	`triggerPos`	Specify trigger position in percent. Can appear only in Global Definition section.
	`postTriggerSamples`	Specify the number of FCLK cycles that will be captured into the trace buffer after trigger. Can appear only in Global Definition section.
Counter Decrement Enable	`enable1`	Specify Decrementing clock for each of the two counters.
	`enable2`	
Simple IF Block	`if`	Used for global trigger, display, tx, acquire, no_acquire and exec
Trigger	`trigger, trig`	Global Trigger Definition
Command Execution	`exec`	Execute a Tcl/XEL command
Message Printing	`display`	Generate a formatted message
Transaction Recording	`tdef`	Define list of transaction attributes
	`tx`	Define beginning, end, or attribute values of a transaction
Conditional Acquisition	`no_acquire, noAcquire, nacq`	Specify when to acquire or not acquire probed data
	`acquire, acq`	

© 2021 All Rights Reserved.

Table 43-2 Predefined Keywords in a Global or Instance Definition Section

Type	SDL Keyword	Usage
General Definitions	scope	Scope resolution of design objects
	enum	Aids in formatting of arguments of display and tx statements
	tdef	Define list of transaction attributes

Table 43-3 lists the predefined keywords that can be used for all other SDL constructs. All the entries in this table, except INSTANCE are used for definitions in the State Machine section. The list does not include keywords that can be used in expressions, and are listed separately.

Table 43-3 Other Predefined Keywords

Type	SDL Keyword	Usage
Instance	`instance`	Beginning of a new SDL instance
Basic blocks of a State Machine	`state`	Definition of an SDL state
	`if`	Control execution based on values of expressions
	`else`	
Counter Names	`counter1`	Names of counters for use in counter actions
	`counter2`	

© 2021 All Rights Reserved.

Table 43-3 Other Predefined Keywords

Type	SDL Keyword	Usage
Actions	`decrement, decr, count`	Decrement counter by 1
	`increment, inc`	Increment counter by 1
	`load`	Load counter with initial value
	`start`	Start counter auto-decrement mode (Start to count every cycle)
	`stop`	Stop counter auto-decrement mode
	`goto`	Transfer control to another state
	`trigger, trig`	Stop design clocks or tracing
	`no_acquire, noAcquire, nacq`	Do not acquire traced data in current cycle
	`acquire, acq`	Acquire traced data in current cycle
	`exec`	Execute a Tcl or XEL command
	`display`	Generate a formatted message
	`tx`	Define beginning, end, or attribute values of a transaction

Expressions

Topics discussed in this section include:

- General Guidelines
- Names
- Counter Conditions
- Assertion State Conditions
- IEEE 1801 Conditions
- Detecting any illegal UPF Events
- Numeric Constants
- Special Non-Numeric Constants

© 2021 All Rights Reserved.

General Guidelines

Expressions are used in `IF` statements to define conditions, or in the Global Definition Section or Instance Definition Section for definitions of `TRIGGER`, `DISPLAY`, `EXEC`, `ACQUIRE`, `NO_ACQUIRE`, `ENABLE1`, and `ENABLE2`.

They can also be used as building blocks for other expressions through the use of Alias.

Expressions are made of Names, Numeric Constants, Operators, Assertion State Conditions, and Counter Conditions. An expression can also be a single name, a single assertion condition, or a single counter condition.

The use of boolean operators and parenthesis in expressions is very similar to their use in C or Verilog.

When evaluated during run, the result of an expression can be either a scalar boolean value (0 or 1), or a vector of boolean values. As in Verilog, a vector can be interpreted as an integer. This interpretation becomes significant when applying an operator between the vector and an integer constant. For example:

```
sig1[7:0] > 23
```

In this example, an 8 element vector representing an 8 bit integer is compared to the integer 23.

The same conventions used in C or Verilog are used when evaluating an expression as a condition: the result of a scalar expression is considered TRUE if it is 1, and FALSE if it is 0. The result of a vector expression is considered TRUE if *any* of its bits is 1, and FALSE if *all* its bits are 0.

Names

A name appearing in an expression can be one of the following:

- *Signal name* - a name of a design signal.
- *DRTL output port* - An output port of a DRTL instance.
- *Assertion name* - a name of an assertion from your design.
- Alias - a name representing a scalar or vector expression defined as an alias to an expression in the Global or Instance Definition Sections of the SDL program.
- *User Symbol* - a name defined externally to the SDL program through the `symbol` XEL command.

© 2021 All Rights Reserved.

- *CPF condition* - a name of a CPF object or event. For detailed information, refer to Specifying CPF Conditions in SDL Expressions of Chapter 33, "Specifying CPF Low-Power Intent on Palladium Z1."
- Name of a UPF object (used in either a UPF condition, or as an argument of a `DISPLAY` statement).

A User Symbol cannot represent arbitrary expressions. It is used by xeDebug for grouping unrelated scalar signals into vectors to streamline and facilitate some operations (such as reading or forcing signals).

An assertion name must contain the full hierarchical path. It can be used in Assertion State Conditions.

Signal Names must contain the full hierarchical path. If a signal name represents a vector, then it can be accompanied by an explicit index notation. If a name representing a vector appears without any index notation, then all its bits are used. An index notation can select a single bit of the vector (bit-select) as in:

```
sig1[3]
```

Or it can select a range of bits (part-select) in a vector as in:

```
sig2[7:2]
```

The order of bits is determined by the two indexes inside the `[]`. In the example above, bit 7 of sig2 is interpreted as the MSB, while bit 2 is interpreted as the LSB.

Aliases or User Symbols cannot be used with an explicit index notation.

If an Alias, User Symbol, or SDL keyword uses the same name as a design signal, then this does not cause an error condition, but the design signal becomes inaccessible to the SDL program. This is a potential issue only for top-level design signals specified without their hierarchy path.

A DRTL output port might be used in SDL like any design signal: for example, inside conditions or as arguments for DISPLAY or TX actions.

The DRTL output port can be referenced in the SDL file in the form:
<drtl_instance_name>.<drtl_output_port_name>

For example:

A DRTL instance is compiled from a verilog source using the following module definition:

```
module ftx (outp);
output[7:0] outp;
```


© 2021 All Rights Reserved.

The verilog source is then compiled and a DRTL instance is defined using the following XEL commands:

```
drtl -definemodule mdx drtl.v -topmodule ftx
drtl -addinst mdx idrt()
```

The output port `outp[7:0]` of the DRTL instance will be referenced in SDL as `idrt.outp[7:0]`.

For example:

```
IF(idrt.outp[3:0] > 5 || idrt.outp[7:4] < 3) TRIGGER;
```

Counter Conditions

Use the > and <= operators to compare the values of one of the two counters. Specify the counter name on the left side of the operator and the integer value on the right. The integer value must be between 0 and (2^{40} -1). Each SDL state can use only one unique integer value for comparing each counter.

Syntax:

```
COUNTER1 <= <value>
```

or:

```
COUNTER2 <= <value>
```

or:

```
COUNTER1 > <value>
```

or:

```
COUNTER2 > <value>
```

<*value*> is an integer value between 0 and 2^{40}-1.

The first two cases that make use of the <= operator are evaluated as 1 (or TRUE) if the counter is less than or equal to the integer value specified by <*value*>.

The third and fourth cases are the inverse of the first two.

For example, `COUNTER1 <= 0 7` is equivalent to `!(COUNTER1 > 0 7)`.

The keywords `CNTR1` and `CNTR2` can be used instead of `COUNTER1` and `COUNTER2`.

© 2021 All Rights Reserved.

Assertion State Conditions

The state of an assertion can be tested by an expression using the following syntax:

<assertion-name> == "<assertion-state>"

<assertion-name> is the full hierarchical name of the assertion from the RTL description.

<assertion-state> is one of the following five predefined strings:

`active inactive finished failed disabled`

These strings can be used in either lower, upper or mixed case.

For example here is a statement that will print out a message when an assertion gets to a finished state:

```
IF (top.ctl.my_assertion == "finished")
    DISPLAY ("my_assertion arrived to a finished state");
```

It is also possible to check whether any assertion in the design failed, by testing the value of the net `ASSERTION.FAILURE`. This net is defined by the compiler only if the design has assertions in it.

For example:

```
IF (ASSERTION.FAILURE)
    DISPLAY ("some assertion failed");
```

See Assertion States of Chapter 27, "Compiling and Running Designs with Assertions" for more information on assertion states.

IEEE 1801 Conditions

The state of an IEEE 1801 object can be tested by an expression using the following syntax:

<1801-object-name> == "<1801-state>"

<1801-object-name> is the full hierarchical name of the IEEE 1801 object.

The `power::` prefix can be added to any of the objects. This is useful if there is a need to distinguish it from a design net with the same name.

<1801-state> is a mnemonic description of the state. It is case sensitive.

Note: Depending on the object type, it is possible that multiple state mnemonics match the state of an object at the same time.

Example:

© 2021 All Rights Reserved.

```
IF (top.VDD1 == ”FULL_ON”) trigger;
```

Or,

```
IF (power::top.VDD1 == ”FULL_ON”) trigger;
```

Following is the list of possible mnemonics for each type of IEEE 1801 object:

- Supply Nets:
  - `OFF FULL_ON PARTIAL_ON UNDETERMINED`
- Supply Ports
  - User-defined states from the `add_port_state` command.
  - `OFF FULL_ON PARTIAL_ON UNDETERMINED`
- Supply Sets
  - User-defined state from the `add_power_state` command.
  - `NORMAL CORRUPT`
- Power Switches
  - User-defined state names from the `create_power_switch` command.
- Power State Table (pst)
  - User-defined state from the `add_pst_state` command.
- Power Domain
  - User-defined state from the `add_power_state` command.
  - `NORMAL CORRUPT`
- Isolation
  - `NORMAL CORRUPT`
  - `ISOLATED NOT_ISOLATED`

    Note: The `ISOLATED NOT_ISOLATED` mnemonics are specific for use in SDL.
- Retention
  - `NORMAL CORRUPT`
  - `RETAINED NOT_RETAINED`

    Note: The `RETAINED NOT_RETAINED` mnemonics are specific for use in SDL.

© 2021 All Rights Reserved.

In addition, it is also possible to test for power states that are described by a single bit (either true or false), as if they are simple nets using the following syntax:

```
<1801-object-name>@<state>
```

Example:

```
IF (top.VDD1@corrupted) trigger;
```

Or,

```
IF (power::top.VDD1@corrupted) trigger;
```

You can get a list of all of the possible conditions in that format for the current design using the `lp -check sdl [<eventType>]` XEL command.

For more details about a formal definition for all possible objects and states that can be used with this syntax, refer to the Using IEEE 1801 Conditions in SDL Expressions section in Chapter 34, "Working with IEEE 1801 Low-Power Designs on Palladium Z1."

Detecting any illegal UPF Events

UPF instrumentation constantly monitors for power sequences that are illegal. You can use the `UPF_ILLEGAL_EVENT` signal in an SDL expression. This signal goes high whenever there is an occurrence of any illegal power sequence (Available only with UPF. Not supported with CPF).

Individual illegal events might be disabled by the `lp -illegal` command; disabled events will not contribute to the `UPF_ILLEGAL_EVENT` signal.

Numeric Constants

Constants used in expressions can be specified with either binary, octal, decimal or hexadecimal radix. The radix notation for regular numerical constants is very similar to the one used by Verilog. For example, a 6-bit constant that represents the decimal value of 14 can be written as any one of the following:

binary	`6'b1110`
octal	`6'o16`
decimal	`6'd14`
hexadecimal	`6'hE`

Most of the Verilog conventions for writing integer constants are followed:

© 2021 All Rights Reserved.

- The radix letter as well as hexadecimal letters are case insensitive.
- If the number of bits is not specified, then it is inferred from the value. For example, `'b1110` is a 4 bit constant representing the value 14.
- A number without an explicit radix notation is assumed to be decimal.
- Underscores can be inserted anywhere in the string (except left most character) to improve readability, but they are otherwise ignored. For example, `'b1110_1111` is equivalent to `'b11101111`.

Special Non-Numeric Constants

SDL also recognizes a set of *special values* that can be used only in conjunction with the comparison operator (`==`). These values are represented by the following letters, which can be in either upper- or lower-case:

`P` - Positive edge.
`N` - Negative edge.
`T` - Any transition (Positive or negative edge).
`L` - Value is stable at a low value (0 for two consecutive FCLK cycles).
`H` - Value is stable at a high value (1 for two consecutive FCLK cycles).
`S` - Value is stable at either low or high value for two consecutive FCLK cycles.
`X` - Don't care; any value is accepted as a match.

The only thing that can be done with these values is compare them to a signal or to an expression with the (`==`) operator. Expressions that are used in such comparisons cannot contain counter conditions.

With the exception of `X` (`don't care`), all other values represent a combination of two consecutive values from the current evaluated cycle and the previous cycle. For example:

```
sig1 == 'bP
```

is testing for a positive edge on the signal `sig1`. The result is 1 (`TRUE`) if the signal `sig1` had the value 0 in the previous cycle and the value 1 in the current cycle.

Consider another example:

```
(sig1 ^ sig2) == 'bT
```

Here, any transition occurring on the XOR result of two signals (the left operand) will make the result of the entire expression 1 (`TRUE`). In other words, this expression is true if one of the signals `sig1` or `sig2` makes a transition, but not if both make a transition at the same time.

Constants using these values must be specified with an explicit radix notation of either binary, octal, or hexadecimal.

© 2021 All Rights Reserved.

When specified as part of a multi-digit constant, each of the letters can represent either 1 bit, 3 bits or 4 bits, depending on the radix, and you can mix in the same constant both numeric digits and any of the special non-numeric constants. For example, the expression:

```
aa[11:0] == 12'h5PX
```

is equivalent to:

```
aa[11:0] == 12'b0101PPPPXXXX
```

Comparing anything to the value `X` always yields a result of 1. This is useful to mask out selected bits when comparing a vector to a constant.

The above expression can therefore be re-written also as:

```
(aa[11:8] == 5) && (aa[7:4] == 4'bPPPP)
```

Note that the least significant 4 binary digits are `X` (`don't care`), so the sub-expression (`aa[3:0] == 4'bXXXX)` became redundant and was omitted altogether.

A typical use of the S (stable) value is to test for a change (or transition) in the value of a vector (meaning change in any of its bits). This is done in practice by checking whether all of the bits are stable, and negating the result, as in the following example:

```
TRIGGER = !(myvec[16:0] == 'hSSSS); //trigger on any change
```

Note: When comparing the value of a design signal with any of the special values except X (that is, any one of: P,N,T,H,L,S), there are limitations that must be observed:

- If the design was compiled to run in 1x or 0.5x sampling mode, it is possible that a signal will make two or more transitions within the same FCLK cycle. For details about how SDL handles this, refer to Limitations when Running with 1x Sampling Ratio and Limitations when Running with 0.5x Sampling Ratio.
- Performing a deposit/force/release through XEL on a signal can cause SDL to evaluate incorrectly any condition with special values that depend on that signal or other signals that it drives through a combinatorial network.
- SDL assumes on the first cycle of each run (except a `run` command with the `-continue` option) that values of design signals did not change from the previous cycle, regardless of what the value in the previous cycle was. In other words, an expression such as `(signal == 'bP)` is always evaluated on the first cycle of the run as false and (`signal == 'bS`) as true. This only affects the `ACQUIRE` or `NO_ACQUIRE` actions and does not affect other types of actions because all other types of actions start to evaluate only on the second cycle of the run.
- Creating an expression that tries to nest conditions with special values except `X` (that is, any one of: `P,N,T,H,L,S`), might not evaluate correctly on the first cycle of the run (except the `run -continue` command) because it requires knowledge of signal values for more than one cycle in the past. For example, an expression such as: `((signal ==`

© 2021 All Rights Reserved.

`'bs) == 'bp)` might not evaluate correctly on the first evaluation cycle of SDL. This limitation affects all types of SDL actions.

Operators

Topics discussed in this section include:

- Using Vector Operands with Different Widths
- Evaluation Order and Operator Precedence Rules
- Unary Bitwise Negation (~)
- Unary Logical Negation (!)
- Unary Reduction AND (&)
- Unary Reduction OR (|)
- Magnitude Comparison Operators (> < >= <=)
- Equal To (==)
- Not Equal To (!=)
- Bitwise AND (&)
- Bitwise XOR (^)
- Bitwise OR (|)
- Logical AND (&&)
- Logical OR (||)
- Concatenation ({...})
- transition()
- count()
- prev()

All the boolean operators can be used on any names (for example, signal names, aliases and so on), constants or expressions, with the only exception that the special non-numeric constants can only be used with the == operator and only against expressions that do not contain Counter Conditions.

© 2021 All Rights Reserved.

Using Vector Operands with Different Widths

SDL handles operations between two vectors that have different widths in a similar way to how Verilog handles unsigned operations, but with minor exceptions, as explained below.

Verilog semantics say that when evaluating a bit-wise operation (such as & | ^ == !=) between two vectors that are unsigned, if the two operands have different widths, then the operand with the smaller width is extended by adding most significant zero elements.

For example, the following expression:

```
A[5:0] == 4'b1111
```

is evaluated by SDL as follows:

```
A[5:0] == 6'b001111
```

SDL has two exceptions to this rule.

- If both operands are not constants, then SDL prints out a warning message if the sizes do not match.

  For example, the following expression is legal, and evaluated as explained above, but will produce a warning message anyway:

  ```
  A[5:0] == B[3:0]
  ```

- The second exception can happen with the == operator when the operand with the smaller width is a constant. If its most significant bit is one of the special non-numeric constants (P,N,T,L,H,S,X), SDL will flag the expression as an error.

  For example, the following will be flagged as an error:

  ```
  A[5:0] == 4'bxx11
  ```

Evaluation Order and Operator Precedence Rules

When several different operators appear in the same expression, the evaluation order is done according to the same precedence rules used by C or Verilog. This is depicted by Table 43-4. Operators in the same box have the same precedence. Parenthesis can be used to change evaluation order.

For example, the operator `==` has higher precedence than `&`, therefore:

```
A & B == C
```

Is equivalent to:

```
A & (B == C)
```


© 2021 All Rights Reserved.

Table 43-4 Operators Precedence

Priority	Operator	Description
Highest	■ ()	Parentheses
	■ transition()	Detect any transition
	■ count()	Count events
	■ prev()	Delay one cycle
	■ ~	Unary bitwise negation
	■ !	Unary logical negation
	■ &	Unary reduction AND
	■ |	Unary reduction OR
	■ >	greater than
	■ <	less than
	■ >=	greater than or equal to
	■ <=	less than or equal to
	■ ==	equal to
	■ !=	not equal to
Lowest	■ &	bitwise AND
	■ ^	bitwise XOR
	■ |	bitwise OR
	■ &&	logical AND
	■ ||	logical OR

Unary Bitwise Negation (~)

When applied to a scalar expression, this operator performs a boolean NOT (inverse) operation.

When applied to a vector expression, the result is also a vector with each bit inverted individually.

© 2021 All Rights Reserved.

Syntax:

```
~ <Expression>
```

Unary Logical Negation (!)

When applied to a scalar expression performs a boolean NOT (inverse) operation.

When applied to a vector expression, the result is a single bit with a value of 1 if all the bits in the vector are 0. If any of the bits in the vector are 1, the result is the single bit 0.

Syntax:

```
! <Expression>
```

Unary Reduction AND (&)

Performs a boolean AND operation between all the elements of a vector expression, producing a single bit.

Syntax:

```
& <Expression>
```

For example, the expression:

```
&A[3:0]
```

is equivalent to:

```
A[3] & A[2] & A[1] & A[0]
```

Unary Reduction OR (|)

Performs a boolean OR operation between all the elements of a vector expression, producing a single bit.

Syntax:

```
| <Expression>
```

For example, the expression:

```
|A[3:0]
```

is equivalent to:

```
A[3] | A[2] | A[1] | A[0]
```


© 2021 All Rights Reserved.

Magnitude Comparison Operators (> < >= <=)

These operators compare between two scalar or vector expressions, and produce a single bit as a result. The result is 0 if the comparison fails, and 1 if the comparison succeeds. Each of the two operands is considered as an unsigned integer.

The number of bits in any one of the two operands is not limited by the SDL syntax, but keep in mind that magnitude comparison operators tax emulation resources more than basic equal/not equal operators, so a large number of bits can cause the SDL run time loader to fail.

Syntax:

Greater Than:

```
<Expression> > <Expression>
```

Less Than:

```
<Expression> < <Expression>
```

Greater Than or Equal To:

```
<Expression> >= <Expression>
```

Less Than or Equal To:

```
<Expression> <= <Expression>
```

Equal To (==)

Compares two scalar or vector expressions, and produces a single bit as a result.

The result is 1 if the two expressions match, and 0 if not.

Syntax:

```
<Expression> == <Expression>
```

The `==` operator can also be used to detect special time dependent conditions (for example, a positive edge). For details, see Special Non-Numeric Constants.

Not Equal To (!=)

Compares two scalar or vector expressions, and produces a single bit as a result.

The result is 0 if the two expressions match, and 1 if not.

Syntax:

© 2021 All Rights Reserved.

```
<Expression> != <Expression>
```

Bitwise AND (&)

Performs a boolean AND operation between two expressions. If the operands are vectors, the result is also a vector with the same number of bits as the operands.

Syntax:

```
<Expression> & <Expression>
```

Bitwise XOR (^)

Performs a boolean Exclusive OR operation between two expressions. If the operands are vectors, the result is also a vector with the same number of bits as the operands.

Syntax:

```
<Expression> ^ <Expression>
```

Bitwise OR (|)

Performs a boolean OR operation between two expressions. If the operands are vectors, the result is also a vector with the same number of bits as the operands.

Syntax:

```
<Expression> | <Expression>
```

Logical AND (&&)

This operation produces a single bit as result whether the operands are vectors or scalars.

If the operands are scalars, performs a boolean AND operation.

If any (or both) of the operands is a vector, then it is first converted to a scalar by performing an OR operation between all its bits. The final result is therefore 0 only if all the bits are 0 in at least one of the operands.

Syntax:

```
<Expression> && <Expression>
```


© 2021 All Rights Reserved.

Logical OR (||)

This operation produces a single bit as result whether the operands are vectors or scalars.

If the operands are scalars, performs a boolean OR operation.

If any (or both) of the operands is a vector, then it is first converted to a scalar by performing an OR operation between all its bits. The final result is therefore the boolean OR operation between all the bits of both operands.

Syntax:

```
<Expression> || <Expression>
```

Concatenation ({...})

A Verilog-style concatenation operator enables you to concatenate scalars and vectors to create new vectors for use in expressions.

Syntax:

```
{<name>,<name>,...}
```

Example:

```
{a,b,c,d,f[3:0]}
```

The result is an 8 bit vector with signal `a` as the most significant bit, and `f[0]` as the least significant bit. It is equivalent to:

```
{a,b,c,d,f[3],f[2],f[1],f[0]}
```

Elements inside the concatenation can only be signal names and symbols defined by the `symbol` XEL command.

Numeric constants, Aliases, or other expressions are not allowed as elements inside concatenation.

transition()

The keyword transition is recognized in upper and lower case.

The syntax is:

```
transition(<expression>)
```

Here, <*expression*> is typically a vector, but can also be a scalar.

© 2021 All Rights Reserved.

The result of the transition operator is true if any of the bits in <*expression*> makes a low-to-high or high-to-low transition.

If <*expression*> is a scalar, the `transition(`<*expression*>`)` is equivalent to `(`<*expression*>`== 'bT)`.

A transition operator on a multi-bit expression can also be translated to an expression with the `==` operator.

For example:

```
transition(A[3:0])
```

is equivalent to the expression:

```
!(A[3:0] == 'hS)
```

Or, to the expression:

```
A[3] == 'bT || A[2] == 'bT || A[1] == 'bT || A[0] == 'bT
```

count()

The `count()` operator represents an actual counter that is private to the enclosing expression — that is, the counter is not shared with other expressions.

This counter should not be confused with *counter expressions* that use the general purpose SDL counters and have a syntax like the following:

```
(counter1 <= 8)
```

The keyword `count` is recognized in either uppercase or lowercase.

The syntax for the `count()` operator is as follows:

`count(`<*expression*>`, `<*num*>`)`

Or:

`count(`<*expression*>`, `<*num*>`, `<*mode*>`)`

Where,

- <*expression*> is any SDL expression. However, it should not contain a counter expression.
- <*num*> is a positive integer.
- <*mode*> is one of the following strings in either uppercase or lowercase: `P S PS SP`

© 2021 All Rights Reserved.

Without <mode>:

The result of the operator becomes `1` after counting a total of <*num*> cycles (not necessarily consecutive) in which the SDL expression is high. The result remains 1 till the counter is reset at the beginning of the next run.

With <mode>:

The mode can be specified in either lower or uppercase. When the mode is specified, the counter operates as follows:

- If <*mode*> is `S`: The counter is reset each time when SDL enters the specific state in which the expression exists — that is, the `count()` operator becomes 1 only if the counter completes counting from 1 to <*num*> without SDL leaving the state.
- If <*mode*> is `P`: The counter counts the number of times <*expression*> changes from 0 to 1, instead of the number of times it stays at value 1.
- If <*mode*> is `PS` or `SP`: This mode is equivalent to the combination of modes `S` and `P`. The counter is initialized at the beginning of the state instead of beginning of the run, and the counter counts 0 to 1 transitions of the expression.

The `S` mode is ignored when the expression is used in a condition for a global action (outside any state statement).

The `P` mode serves as a shortcut for the syntax (<*expression == 'bp*>). For example,

```
count(abc[15:0] == 13, 50, P)
```

is equivalent to:

```
count((abc[15:0] == 13) == 'bp, 50)
```

For example:

```
if(count(net1,10,p) || count(net2,20))
trigger;
```

In the example above, trigger happens if either the value of `net1` undergoes 10 transitions from 0 to 1, or if `net2` stays high for a total of 20 FCLK cycles.

prev()

The `prev` operator acts as a delay — that is, the result of a `prev` operator is the expression delayed by one clock cycle. The keyword `prev` is recognized in either uppercase or lowercase. The syntax is:

```
prev(<expression>)
```


© 2021 All Rights Reserved.

<expression> is any SDL expression, scalar or vector. It should not contain a counter expression, however, *<expression>* can contain the `count` operator.

General Definitions

Topics discussed in this section include:

- Alias
- enum
- tdef
- scope

Any of the above definitions might appear anywhere in the SDL program, outside of `STATE` statements. The definition affects all subsequent statements until the end of the SDL program, the end of the SDL instance, or a new definition that overrides it (Details vary, depending on the type of statement, see the details mentioned below).

Alias

An alias defines a new name which can be used later instead of typing the whole expression. The expression used for defining an alias can itself contain names of other aliases already defined.

Syntax:

```
<name> = <Expression>;
```

<name> can contain only alphanumeric characters or an underscore (_), and cannot begin with a digit. Names of aliases are case sensitive.

An alias appearing in a Global Definition Section can be used anywhere below that definition, including all the SDL instances. An alias definition in an Instance Definition Section is private to that Instance.

The same alias name can be used to define a different alias value in different instances. However, if an alias was defined in the Global Definition Section, then it cannot be redefined in any Instance Definition Section.

An alias cannot be indexed and cannot be used as part of a hierarchical path. For example, if YYY is defined as an alias like `YYY = XXX`, then the following usage is not supported:

- `YYY[0]` cannot be used as `XXX[0]`

© 2021 All Rights Reserved.

- `YYY.ABC` cannot be used as `XXX.ABC`

enum

Provides an efficient way to chose a string out of a list of several strings, based on an integer value, when formatting integer values in `DISPLAY` or `TX` statements.

The syntax of the enumeration statement is:

```
enum <enum_type>
     {
        <string> [=<value>],
        <string> [=<value>],
        ...
     };
```

For more details, refer to DISPLAY.

tdef

Defines a default set of attributes for `TX` statements.

The syntax of the `tdef` statement is as follows:

```
tdef <tdef_id>
     {
        <arg>,
        <arg>,
        ...
     };
```

For more details, refer to TX.

scope

Defines a hierarchical scope for nets and other design objects that appear in expressions.

To set the scope:

```
scope <scope>;
```

To reset the scope:

```
scope;
```

In the first form, `<scope>` specifies the hierarchical scope. A period is used as a hierarchy separator.

© 2021 All Rights Reserved.

The second form in which `<scope>` is missing, resets the scope, which means objects are searched only under top hierarchy (The default behavior if the SDL program does not contain any `scope` statements).

The scope definition affects all the expressions below it up to the next scope statement or the end of the SDL program. Any design object below the scope definition is first searched in the design hierarchy as defined by the scope statement, and if not found there, it is searched in the top hierarchy.

For example, consider the following:

```
scope top.h1.h2;
if(a1.n1) trigger;
```

SDL will use for the trigger statement the signal `top.h1.h2.a1.n1` if it exists. If there is no such signal, it will use the signal `a1.n1`, and will issue an error message if this signal also does not exist.

Global or Instance Definition Sections

Topics discussed in this section include:

- Global Definition Section
- Instance Definition Section
- Global Definitions
- Trigger Position
- Post Trigger Samples
- Count Enable
- Counter Load Value
- Global TRIGGER
- Global ACQUIRE
- Global NO_ACQUIRE
- Global ACQUIRE for Streaming Probes
- Global EXEC
- Global DISPLAY
- Global TX

© 2021 All Rights Reserved.

Global Definition Section

The Global Definition section appears at the top of the SDL program above the first INSTANCE Statement. Its definitions apply to all the SDL instances.

Instance Definition Section

The Instance Definition section appears between the INSTANCE statement and the first STATE statement of that instance. Its definitions apply only to that instance.

See Figure 43-1 on page 2006 as an example.

Global Definitions

Global definitions in an SDL program are statements that appear either in the Global Definition section, or in the Instance Definition section.

If the SDL program has only one instance with no INSTANCE statement, then the Global Definition section and the Instance Definition section are the same thing.

Except arguments of DISPLAY statements, expressions used in all other global definitions cannot rely on any state machine resources. This means they cannot contain counter values.

Global definitions use either the syntax:

```
<name> = <value>;
```

or the syntax:

```
if(<expression>) <action>;
```

<*name*> can be one of the keywords: `TRIGGER, ACQUIRE, NO_ACQUIRE, COUNTER1, COUNTER2, ENABLE1, ENABLE2, TRIGGERPOS, POSTTRIGGERSAMPLES`.

<*value*> is either a numeric value or an expression depending on what is being defined.

The second syntax is supported by the following statements types: `TRIGGER, DISPLAY, TX, ACQUIRE, NO_ACQUIRE,` and `EXEC`, which are similar (both syntax and functions) to the actions with same name inside the SDL state machine.

Each of the possible global definitions is discussed below.

© 2021 All Rights Reserved.

Trigger Position

SDL can override the value of the `triggerPos` run-time parameter by providing a different value in the Global Definition Section.

The syntax is:

```
triggerPos = <value>;
```

The keyword `triggerPos` can appear in lower, upper, or mixed case. *<value>* is an integer number between 0 and 100. This value is assigned to the `triggerPos` run-time parameter (overwriting any previous value) when the `run` command is executed, and it is used during the run.

A `triggerPos` definition can appear only in the Global Definition Section, but not in the Instance Definition Section. Both `triggerPos` and `postTriggerSamples` definitions cannot appear together in the same SDL program.

For more information on the semantics of the `triggerPos` run-time parameter, refer to the description of the `xeset` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Post Trigger Samples

SDL can override the value of the `postTriggerSamples` run-time parameter by providing a different value in the Global Definition Section.

The syntax is:

```
postTriggerSamples = <value>;
```

The `postTriggerSamples` keyword can appear in lower, upper or mixed case. *<value>* is a non-negative integer number. This value is assigned to the `postTriggerSamples` run-time parameter (overwriting any previous value) when the `run` command is executed, and it is used during the run.

A `postTriggerSamples` definition can appear only in the Global Definition Section, but not in the Instance Definition Section. Both `triggerPos` and `postTriggerSamples` definitions cannot appear together in the same SDL program.

For more information on the semantics of the `postTriggerSamples` run-time parameter, refer to the `xeset` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

© 2021 All Rights Reserved.

Count Enable

Defines a condition upon which one of the two counters will decrement when in auto-decrement mode (using the START and STOP actions). The Global Definition Section or Instance Definition Section can contain at most one count enable definition for each counter.

A count enable definition appearing in the Global Definition Section applies to all SDL instances. A count enable definition in an Instance Definition Section is private to that Instance.

Different enable definitions can be defined in different instances. However, if an enable definition appears in the Global Definition Section, then it cannot be redefined in any Instance Definition Section.

If no enable definition exists for a specific counter, then that counter will decrement once every FCLK cycle (when in auto-decrement mode).

Note: The VD and ICE run modes have clear semantics that define a linear relationship between FCLK cycles and simulation time.

This construct is therefore most useful for Simulation Acceleration mode which does not have such clear semantics.

Syntax:

```
ENABLE1 = <expression>;
```

or:

```
ENABLE2 = <expression>;
```

A typical use is decrementing on a specific clock edge from the design, as in the following example (count on positive edge of `mainclock`):

```
ENABLE1 = mainclock == 'bp;
```

Counter Load Value

Defines the default numeric value for a LOAD action if none was specified with the `Load` action itself. The Global Definition Section or Instance Definition Section can contain at most one counter load value definition for each counter.

A load value definition appearing in the Global Definition Section applies to all SDL instances. A load value in an Instance Definition Section is private to that Instance.

© 2021 All Rights Reserved.

Different load values can be defined in different instances. However, if a load value was defined in the Global Definition Section, then it cannot be redefined in any Instance Definition Section.

Syntax:

```
COUNTER1 = <value>;
```

or:

```
COUNTER2 = <value>;
```

The shorthand notations `CNTR1` and `CNTR2` are also recognized.

<value> is an unsigned integer value between 0 and 2^{40}-1. The value can be specified as a decimal number or in a different radix (as specified in Numeric Constants on page 2017).

Global TRIGGER

Defines a trigger condition. The trigger is executed whenever the specified expression becomes true regardless of any other trigger statements. Any number of global trigger statements can be declared in any of the Global Definition Section or Instance Definition Section, and each is evaluated independently every cycle. A true condition in any of them will cause a trigger. The condition can be constructed as explained in Expressions, except that it cannot contain counter conditions (for example, `COUNTER1 <= 0`).

Syntax:

```
<Label>: TRIGGER = <Expression>;
```

The above construct has an equivalent form using an `IF()` clause as follows:

```
IF(<Expression>)
      <Label>: TRIGGER;
```

The shorthand notation `TRIG` is also accepted instead of `TRIGGER`.

<Label> is a string made of letters, digits, or underscores, and must start with a letter or underscore. It can be used by XEL for the purpose of identifying, enabling or disabling any `TRIGGER` statements that have a specific label. *<Label>* does not have to be unique within the SDL program. The construct *<Label>*`:` is optional. It can be omitted if labeling the statement is not needed.

For more details on this statement, see TRIGGER.

© 2021 All Rights Reserved.

Global ACQUIRE

Defines a condition which when true in any cycle, will cause probed data capturing in that cycle. Any number of global `ACQUIRE` statements can be declared in any of the Global Definition Section or Instance Definition Section, and each is evaluated independently every cycle. A true condition in any one of them will cause data capturing in that cycle. The condition can be constructed as explained in Expressions, except that it cannot contain counter conditions (for example, `COUNTER1 <= 0`).

`ACQUIRE` and `NO_ACQUIRE` statements (whether inside a STATE or a global/instance definition) cannot be used in the same SDL program.

Syntax:

```
ACQUIRE = <Expression>;
```

or an equivalent form:

```
IF(<Expression>)
    ACQUIRE;
```

The shorthand notation `ACQ` is also accepted instead of `ACQUIRE`.

Global NO_ACQUIRE

Defines a condition which when true in any cycle, will suspend probed data capturing in that cycle. Any number of global `NO_ACQUIRE` statements can be declared in any of the Global Definition Section or Instance Definition Section, and each is evaluated independently every cycle. A true condition in any one of them will suspend data capturing in that cycle. The condition can be constructed as explained in Expressions, except that it cannot contain counter conditions (for example, `COUNTER1 <= 0`).

`ACQUIRE` and `NO_ACQUIRE` statements (whether inside a STATE or a global/instance definition) cannot be used in the same SDL program.

Syntax:

```
NO_ACQUIRE = <Expression>;
```

or an equivalent form:

```
IF(<Expression>)
    NO_ACQUIRE;
```

The shorthand notations `NOACQUIRE` and `NACQ` are also accepted instead of `NO_ACQUIRE`.

© 2021 All Rights Reserved.

Global ACQUIRE for Streaming Probes

Defines conditions that control when to acquire streaming probes data, independent of regular probe data capturing.

Syntax:

```
if(<expression1>) acquire stream stop;
if(<expression2>) acquire stream start;
if(<expression3>) acquire stream;
```

For more information, refer to the Conditional Acquisition for Streaming Probes section.

Global EXEC

Defines a condition and an XEL command (or set of commands). The condition, when true, will initiate execution of the XEL command. In emulation modes that enable stopping the clocks, emulation clocks are suspended while the command executes. Refer to EXEC for more details on the `EXEC` facility.

The condition can be constructed as explained in Expressions, except that it cannot contain counter conditions (for example, `COUNTER1 <= 0`).

Syntax:

```
IF(<Expression>)
   <Label>: EXEC "<command>";
```

`<Label>` is a string made of letters, digits, or underscores, and must start with a letter or underscore. It can be used by XEL for the purpose of identifying, enabling or disabling any `EXEC` statements that have a specific label. *`<Label>`* does not have to be unique within the SDL program. The construct *`<Label>`*`:` is optional. It can be omitted if labeling the statement is not needed.

For more details on this statement, see EXEC.

Global DISPLAY

Defines a condition and a message to be printed. The condition, when true, generates a message printed to a file and/or the console.

Syntax:

```
IF(<Expression>)
   <Label>: DISPLAY (F=<ID>, L=<Level>, "<format-specifier>",
                     <Arg>, <Arg>, ...);
```


© 2021 All Rights Reserved.

<Expression> is a condition. The DISPLAY message is printed whenever this *<Expression>* is true. It can be constructed as explained in Expressions, except that it cannot contain counter conditions (such as, COUNTER1 <= 0).

By default, a maximum number of 256 DISPLAY and TX statements can be in the Global Definition Section and Instance Definition Sections (from all instances). If required, you can change this number by using the sdlGlobalDisplay option of the compilerOption command.

For more details on this statement, see DISPLAY.

Global TX

Defines the beginning, end, or additional attributes of a transaction.

Syntax:

```
IF (<Expression>)
   <label>: TX(<Arguments>);
```

For more details on this statement, see TX.

TX shares hardware resources with DISPLAY. For additional information on <expression> and maximal number of statements see the Global DISPLAY section.

INSTANCE Statement

Defines the beginning of a new SDL instance, and assigns a name to the instance. The name is used for identification by XEL, but other than that is not used inside the SDL program itself. See Setup during Design Compilation on page 1999 for information on how to use more than one instance.

Syntax:

```
instance <InstanceName>;
```

<InstanceName> is a string made of letters, digits or underscores and must start with a letter or underscore.

STATE Statement

Declares the start of a state definition. Each state performs actions based on conditions and tests.

© 2021 All Rights Reserved.

You can declare up to 64 states. The first state in the SDL program is the starting state. Only one state is active in any given cycle.

Syntax:

```
STATE <StateName>
{
    <StateBody>
}
```

where `<StateName>` is a string, and `<StateBody>` is a set of `IF-ELSE` statements or Action statements. The `<StateName>` is a string made of letters, digits or underscores and must start with a letter or underscore.

IF - ELSE statement

A basic `IF` statement evaluates an expression. If the expression is `TRUE`, then it executes a statement, or a set of statements. An `ELSE` statement can also be specified to execute a different set of statements if the expression evaluated to FALSE.

Syntax:

```
IF (<Expression>)
    <Statement>
```

or:

```
IF (<Expression>)
{
    <Statement>
    <Statement>
     . . .
}
```

An `ELSE` statement can immediately follow an IF statement.

Syntax:

```
ELSE
    <Statement>
```

or:

```
ELSE
{
    <Statement>
    <Statement>
     . . .
}
```

`<Expression>` can be scalar or vector. A scalar expression is considered `TRUE` if it has the value 1. A vector expression is considered `TRUE` if any of its bits is 1.

© 2021 All Rights Reserved.

<Statement> can be another IF-ELSE statement or any Action statement (See Examples: SDL Program on page 2063). There are no restrictions on depth of nesting of IF-ELSE statements, although there are limitations on the maximal number of IF statements that can be used in each state.

When an IF statement is the only statement nested inside an ELSE block, then no braces are needed for the ELSE block, and the nesting takes the well known structure of IF-ELSE-IF chain. A nested IF chain can be written as follows:

```
IF (<expression1>)
    <action1>
ELSE IF (<expression2>)
    <action2>
ELSE IF (<expression3>)
    <action3>
ELSE
    <action4>
```

The expressions are evaluated from top to bottom, and the action attached to the first expression that is true is executed.

The following truth table shows how SDL interprets the above example:

<expression1>	<expression2>	<expression3>	Action Executed
true	don't care	don't care	<action1>
false	true	don't care	<action2>
false	false	true	<action3>
false	false	false	<action4>

A state can contain more than one chain of IF-ELSE-IF constructs. Independent IF statements are executed concurrently. If there is a possibility for a conflict between actions from different IF statements within the same state (for example, GOTO actions to different destinations), then the SDL compiler will issue a warning message, and the first action from the top will be executed. Obvious cases, in which one of the actions has no chance of ever getting executed will be flagged as errors.

Limitations Related to Expressions and IF Statements

SDL has an inherent limit on the maximal number of IF statements in each state, which is mostly unrelated to how these IF statements are nested. The limit depends on the presence of counter conditions in the expressions inside the IF statements. Without counter conditions, the rule is simple: there can be no more than 8 expressions (or IF statements) in each state.

© 2021 All Rights Reserved.

There is no limit on the number of expressions constructed exclusively of counter conditions. If an expression contains both counter conditions and signal names, the SDL compiler tries to convert the expression into several sub-expressions that are either counter conditions, or expressions based on signal names. The resulting number of sub-expressions based only on signal names add up towards the maximal limit of 8. These restrictions do not apply to IF statements in the global or instance definition sections, and it also does not apply to IF statements that do not control directly any actions. Example:

```
IF (exp1) {
    IF(exp2) { GOTO nxt; increment counter1;}
}
```

In the above example, `IF(exp1)` does not control any action directly, therefore `exp1` is not counted towards the maximum of 8 expressions.

The SDL compiler might also fail to download complex expressions into the emulator due to lack of sufficient emulation resources to emulate these expressions. When this happens, an error message might provide some suggestions for working around the failure. One of the possible suggestions is to increase the step count per FCLK cycle. To change the step count, see description of the `extra_steps` option of the `compilerOption` XEL command in the *User Data Commands* chapter of the *VXE Command Reference Manual*.

Actions

The following actions can be called from inside STATE Statements:

- GOTO
- LOAD
- DECREMENT
- INCREMENT
- START
- STOP
- TRIGGER
- ACQUIRE
- NO_ACQUIRE
- EXEC
- DISPLAY
- TX

© 2021 All Rights Reserved.

- TRACE_TSM

GOTO

Transfer execution from the current state to the specified state, in the next cycle.

Syntax:

```
GOTO <StateName>;
```

where *<StateName>* is the name of a state.

LOAD

Loads one of the counters with an initial value. Each `LOAD` action can use a different load value. The `LOAD` action must be executed before executing a `START` or `DECREMENT` actions to guarantee that the counter starts from a known value.

Syntax:

```
LOAD COUNTER1 <value>;
```

or:

```
LOAD COUNTER2 <value>;
```

<value> is an integer value between 0 and (2^{40}-1). The value can be specified as a decimal number or in a different radix (see Numeric Constants on page 2017).

If *<value>* is not specified, SDL uses a default value as explained in Counter Load Value on page 2034.

The counter is loaded with the new value on the cycle that immediately follows the LOAD action.

If SDL previously executed a `START` action, and did not execute a `STOP` action, or if a START action was executed concurrently with the Load action, then the counter is first loaded with the new value, and begins decrementing in the cycle after that.

For example, if at cycle 100 the following two actions were executed:

```
START COUNTER1
LOAD COUNTER1 20
```

then counter1 will show the value 20 in cycle 101, and the value 19 in cycle 102.

© 2021 All Rights Reserved.

DECREMENT

Decrements the specified counter by 1. This is done unconditionally regardless of the Count Enable. This statement is meaningful only when the counter is in a stopped state. If a counter is auto-decrementing because a previous START statement defined it as free running and its enable control is true, the DECREMENT statement has no effect on the counter.

Syntax:

```
DECREMENT COUNTER1 ;
```

or:

```
DECREMENT COUNTER2 ;
```

When the counter reaches 0, decrementing stops.

The keywords `DECR` and `COUNT` are recognized as synonyms for `DECREMENT`.

INCREMENT

Increments the specified counter by 1 unconditionally. If a counter is auto-decrementing because a previous START statement defined it as free running, the INCREMENT statement overrides the default auto-decrementing and causes the counter to increment instead of decrementing in that cycle.

Syntax:

```
INCREMENT COUNTER1 ;
```

or:

```
INCREMENT COUNTER2 ;
```

When the counter reaches its maximum value of (2^{40}-1), incrementing stops.

The keyword `INC` is recognized as a synonym for `INCREMENT`.

START

Puts the counter in auto-decrement mode. In this mode, the counter decrements by 1 each time when the expression represented by the Count Enable is true. If no count enable is specified, then the counter is decremented once every cycle. When the counter reaches 0, decrementing stops, but only the STOP action takes the counter out of the auto-decrement mode.

Syntax:

© 2021 All Rights Reserved.

```
START COUNTER1 ;
```

or:

```
START COUNTER2 ;
```

STOP

Stops the specified counter from auto-decrementing.

Syntax:

```
STOP COUNTER1 ;
```

or:

```
STOP COUNTER2 ;
```

TRIGGER

The effect of the `TRIGGER` action depends on the operation mode.

When running in LA mode, or in Dynamic Target mode the `TRIGGER` action causes probed data tracing to stop after the specified amount of post trigger samples are captured into the trace buffer. After trigger the SDL program continues to run, so the `NO_ACQUIRE` action can still control what data samples should be traced into memory (if the number of requested post-trigger samples is non zero).

When running in SA, Vector Debug, or STB modes, the trigger cause the design running on the emulator to stop (after the number of requested post-trigger samples is captured into the trace buffer).

The default behavior of `trigger` can be modified from XEL through the `sdl -haltOnTrigger` and `clockConfig -stopMode` commands. (For example, make Trigger stop the design clocks in LA mode).

Syntax:

```
<Label>: TRIGGER;
```

<Label> is a string made of letters, digits or underscores and must start with a letter or underscore. It can be used by XEL for the purpose of identifying, enabling or disabling any `TRIGGER` statements that have a specific label. *<Label>* does not have to be unique within the SDL program. The construct *<Label>*: is optional. It can be omitted if labeling the statement is not needed.

© 2021 All Rights Reserved.

The effect of the TRIGGER action is independent of other TRIGGER statements. Each can cause a trigger independently.

When running in SA or STB mode, the SDL examines design signals with some delay. For precision debugging, you can eliminate the delay by running at a lower speed. See Fast versus Accurate on page 1987 for more details.

ACQUIRE

Enables probed data sampling in the current cycle. Use this action to more-selectively fill the trace buffer.

Syntax:

```
ACQUIRE ;
```

When using the ACQUIRE action, only cycles in which an ACQUIRE action (or a global acquire) is executed will capture a probe sample.

The use of the ACQUIRE action precludes the use of the NO_ACQUIRE action or the global NO_ACQUIRE definition.

NO_ACQUIRE

Temporarily prevents probed data sampling in the current cycle. Use this command to more-selectively fill the trace buffer.

Syntax:

```
NO_ACQUIRE ;
```

The effect of this action is independent of the Global NO_ACQUIRE. Each can prevent data sampling independently.

The use of NO_ACQUIRE precludes the use of the ACQUIRE action or the global ACQUIRE definition.

EXEC

This action causes execution of a Tcl or XEL command, as if it was executed from the XE command line prompt. In emulation modes that enable stopping the clocks, emulation clocks are suspended while the command executes. This feature is available in STB and LA modes and also under IXCOM with either static or dynamic target. It is not available when running in infiniTrace or VD modes.

© 2021 All Rights Reserved.

Syntax:

```
<Label>: EXEC "<command>";
```

<Label> is a string made of letters, digits or underscores and must start with a letter or underscore. It can be used by XEL for the purpose of identifying, enabling or disabling any EXEC statements that have a specific label. *<Label>* does not have to be unique within the SDL program. The construct *<Label>*: is optional. It can be omitted if labeling the statement is not needed.

<command> represents almost any string that can be executed by the XEL command interpreter during a debug session, including calls to Tcl procedures defined earlier, or even a sequence of commands separated by semicolons.

For example, suppose that my_proc is a Tcl procedure that takes a single numeric argument and was already defined. The following prints a short message, then calls the procedure with an argument 11.

```
EXEC "puts {executing my_proc} ; my_proc 11";
```

The following restrictions apply:

The whole command string needs to be enclosed in double quotes. So, it cannot include any double quotes unless they are properly escaped with a '\' (Following normal Tcl escape rules). For example:

```
EXEC "puts \"time now is: [clock format [clock seconds] -format %c]\"";
```

Any Tcl variable reference that appears inside the <command> string (with the syntax $var or ${var}) is substituted with its value at the time when the EXEC action is executed. For example consider the following snippet from an SDL program:

```
IF (signalx == 1)
     EXEC "puts \" xyz = $xyz\"; incr xyz";
```

If the Tcl variable xyz was initialized before the run command to the value 5 and the EXEC action is executed 3 times during the run, then the following output is produced:

```
5
6
7
```

EXEC actions are always executed at global Tcl call level. In other words, variables local to a Tcl proc are not accessible to EXEC.

Some XEL commands normally allowed in a debug session cannot be executed by an EXEC, whether they are called directly from the EXEC statement, or indirectly from a Tcl procedure called from the EXEC statement. The list of XEL commands that are not allowed to be executed from an EXEC is:

© 2021 All Rights Reserved.

- `configPM`
- `debug`
- `download`
- `host`
- All `infiniTrace` command options except:

  `-prepare -on -off -close`
- `restart`
- `reset`
- `run`
- `save`
- `vector`
- `waitWhileBusy`
- `xc`
- The following options of the `sdl` command:
  - `-setFile -traceOn -verify -expression -restoreState -load -onlyReport`

`EXEC` requests are serviced whether running in blocking or non-blocking mode.

When running in STB mode, SDL examines design signals with a delay of up to 2 Fast Clock (FCLK) cycles. This means that any `EXEC` based on values of design signals will be delayed by up to two FCLK cycles.

For precision debugging, you can eliminate the delay by running at 1/3 of the regular speed. See Fast versus Accurate on page 1987 for more details.

When running in STB mode, each `EXEC` request cause FCLK to pause and the software service the request before any other `EXEC` requests are issued. For all other operation modes SDL maintains a small buffer for holding `EXEC` requests until the software can service them. The buffer can hold `EXEC` requests from up to `4` separate FCLK cycles.

When running in emulation modes that do not enable to stop the clocks, such as LA mode, Dynamic Target mode under IXCOM, or with uncontrolled clocks under IXCOM, clocks are not suspended while `EXEC` requests are executed, and the execution delay of the `EXEC` request can be thousands of cycles. Moreover, since the clocks do not stop, it is possible that

© 2021 All Rights Reserved.

SDL issues a new EXEC request before all the previous EXEC requests in the EXEC buffer are serviced, and in such case, the new EXEC request is lost. This is called an EXEC overflow condition.

The system buffers up to four EXEC requests at a time. To avoid EXEC overflow, ensure that the EXEC requests are adequately spaced in time and a new EXEC is not issued before the previous ones finish execution, otherwise, an error message will be generated.

SDL can determine if two or more pending EXEC requests are waiting in the buffer by checking the value of the EXEC_BUSY signal. A high value for this signal indicates that a new EXEC command will overflow the buffer. The following example shows the usage of the EXEC_BUSY signal in SDL expressions in order to avoid EXEC overflow:

```
if(cond && !EXEC_BUSY)
EXEC "my_tcl_command";
```

In other co-simulation modes under IXCOM, the design is paused to execute the EXEC action with possibly some delay. The amount of delay depends on the design itself and the selected run mode. If you executed the xeset stopDelay 0 command earlier, then the EXEC command will be scheduled for execution at the end of the simulation time stamp from which it was invoked. If the stopDelay parameter was selected as 1, then the EXEC will be scheduled for execution either at the end of the time stamp from which it was issued, or at the end of the following time stamp. If more than one EXEC actions are scheduled for execution in the same simulation time stamp, but issued from different FCLK cycles, then these EXEC requests might cause an EXEC overflow condition if they overflow the buffer size of 4 stages. One way to avoid an overflow is to make sure that SDL is evaluated no more than once per each simulation time stamp. This can be done with the following XEL command:

```
sdl -simTime 1
```

By default, an EXEC overflow condition will stop the run with an error message; but this behavior can be modified from XEL (with the sdl -execOvf command) to either ignore the condition altogether or only produce a warning message without stopping the run.

DISPLAY

Prints a message to the console and/or a file, using a syntax similar to the Verilog $display system task.

Syntax:

```
<Label>: DISPLAY (F=<ID>, L=<Level>, "<format-specifier>", <Arg>, <Arg>, ...);
```

Any of the components in the above syntax are optional, except the keyword DISPLAY and the parenthesis. If *<Label>*:, F=*<ID>*, L=*<Level>* are taken out, the above line takes the familiar form of the $display() statement from Verilog.

© 2021 All Rights Reserved.

<Arg>, <Arg>, ... are optional arguments to be printed. Each is an expression as explained in Expressions, except that they cannot contain SDL counter conditions. However, the counter name (counter1 or counter2) might be used as an argument by itself. In such case the integer value of the counter is printed, with a delay of up to 2 FCLK cycles.

<Label> is a string made of letters, digits or underscores and must start with a letter or underscore. It can be used by XEL for the purpose of identifying, enabling or disabling any , DISPLAY statements that have a specific label. *<Label>* does not have to be unique within the SDL program. It is also inserted as part of the output message if *<format-specifier>* is missing. The construct *<Label>*: is optional. It can be omitted if labeling the statement is not needed.

F=*<ID>* is an optional argument. *<ID>* is an identifier that enables XEL to route the message to a specific file associated with this identifier.

The file associated with *<ID>* is opened by the following XEL command:

```
sdl -display -id <ID> -file <name>
```

When the value of <ID> is the reserved string MARKER, the DISPLAY statement is used for creating time markers for viewing in the VERDI waveform browser or for use as bookmarks in InfiniTrace. For information on how to create time markers and bookmarks, refer to the Creating Time Markers and Bookmarks section.

L=*<Level>* is an optional argument. *<Level>* is an integer value that enables the messages to be enabled or disabled from XEL based on a single threshold value. The command sdl -display -loglevel *<loglevel>* specifies a single integer value *<loglevel>*. The output from the DISPLAY statement is suppressed if *<level>* is less than *<loglevel>*. If *<level>* is not specified, then a default value is assumed. The default value is defined by the XEL command sdl -display -deflevel *<deflevel>*.

<format-specifier> is a string containing the main text of the message. It might possibly contain escape sequences, which are substituted with other strings according to Table 43-5 below. The sequences %h %d %i %o %b %s %[<enum-type>] and the arguments *<Arg>, <Arg>, ...* are matched in sequence. The *<format-specifier>* is an optional argument. If not specified, the output message takes a standard form as specified below.

The format specifier might also contain special characters using a leading backslash (or other combinations), as shown in Table 43-5. A backslash followed by any other character will be ignored and removed from the output string.

Tcl variable substitution using $*name* and ${*name*} is also supported inside the format specifier. The Tcl variable value substitution is performed at the beginning of each run.

© 2021 All Rights Reserved.

Table 43-5 Escape Sequences in the Format Specifier of a Display Statement

Escape Sequence	Explanation
`%h` or `%H`	Convert one *<arg>* using unsigned hexadecimal conversion.
`%d` or `%D`	Convert one *<arg>* using unsigned decimal conversion
`%i` or `%I`	Convert one *<arg>* using signed decimal conversion
`%o` or `%O`	Convert one *<arg>* using unsigned octal conversion
`%b` or `%B`	Convert one *<arg>* using unsigned binary conversion
`%c` or `%C`	Convert one *<arg>* using ASCII character conversion
`%s` or `%S`	*<arg>* is a design signal: Convert *<arg>* using ASCII conversion *<arg>* is a UPF or assertion object: Convert *<arg>* to mnemonic.
`%n` or `%N`	Name of SDL instance
`%t` or `%T`	Simulated time
$<var>	Insert the value of a global TCL variable
${<var>}	Insert the value of a global TCL variable
`%`**[***<enum_type>***]**	Convert a numeric value to a string using `enum`
`%[...]`	Use a TCL command to perform complex conversions
`\n`	Newline
`\t`	Tab
`\\`	The character `\`
`\"`	The character `"`
`\$`	The character `$`
`%%`	The character `%`

The format specifier character can be either lower or upper case. The only difference between lower case and upper case is when using a hexadecimal conversion. The resulting string from

© 2021 All Rights Reserved.

`%h` will be constructed with only lower case characters, while the resulting string from `%H` will be constructed with only upper case characters.

The field size (number of characters) allocated when substituting an argument is set to the maximal possible number of characters that the argument might require. For example, a `%b` conversion for an `8` bit argument will take `8` characters, while a `%d` conversion will take `3` characters (because the maximum decimal value of an `8` bits argument is `255`). A `%i` conversion will take `4` characters because it also needs to account for a sign character.

The value is right justified in the field. Binary, octal and hexadecimal conversions are padded with zeros on the left, while decimal values are padded with spaces. The padding can be eliminated and the result forced into the minimum required field by inserting the digit `0` immediately to the right of the `%`.

For example, suppose that the `8` bit argument `data[7:0]` represents the decimal value `12` (binary `1100`):

```
DISPLAY ("DATA=:%d:%0d::%b:%0b:", data[7:0],data[7:0],data[7:0],data[7:0]);
```

This produces the following output:

```
DATA=: 12:12::00001100:1100:
```

Note that the conversion rules are similar to those used by the Verilog `$display()` system task, except that `%t` is not matched with any argument, and Verilog does not support `%i` conversion.

`%c` prints out a single character representing the ASCII code of the low-order byte of the argument, regardless of the number of bits in the input argument.

`%s` can operate in one of two ways depending on the argument type:

If the corresponding argument is a multi-bit bus composed of regular design nets, then `%s` prints out one or more characters representing the ASCII code of the input argument. For example, if the argument is `ABC[31:0]`, the output string will have four characters with the first one representing the ASCII value of `ABC[31:24]` and the last one `ABC[7:0]`.

If the corresponding argument is the name of a UPF or assertion object, then `%s` prints the value of that object as a mnemonic string. (For example, `failed` is one of several possible mnemonic values for an assertion).

The list of possible mnemonics for UPF objects is specified at: IEEE 1801 Conditions. In addition, voltage of supply nets or supply ports can be printed with the `%s` specifier by appending `@voltage` to the name of an object. For example:

```
display(”voltage of supply net xyz is %s”,xyz@voltage);
```


© 2021 All Rights Reserved.

Depending on the object type, the printout from a single `%s` specifier might show more than one mnemonic. For example, both voltage and state are printed for supply nets with the `@voltage` specifier. For retention and isolation rules, the displayed string contains the state of the rule's supply and whether the rule is active.

`%n` prints the name of the SDL instance from which the `DISPLAY` statement is executed. It is not paired with an argument.

`%t` is simulated time, which is formatted using the default time units that were set by the `xeset timeUnit` XEL command. It is not paired with an argument.

The printed value of *`%t`* contains both a floating point number and time units.

For example: *`25.3 ns`*

The precision (number of digits in the fractional part of the printed value) can be specified by a decimal point followed by the number of digits. For example:

```
display("%8.3t");
```

This will print exactly 3 digits to the right of the decimal point, and the total number of characters printed will be no less than 8.

If a precision is not specified, the printed value will contain maximum accuracy and only trailing zeros are truncated.

Formatting with Enum

The enumeration type conversion `%[<enum_type>]` provides an efficient way to chose a string out of a list of several strings, based on an integer value. The integer value must be unsigned with upto 64 bits.

A format conversion using a TCL procedure can also do the same job (for example, with the generic TCL statement `lindex`). However, it is expected to be slower because it has to be processed by the TCL interpreter. Therefore, if a conversion can be done using `%[<enum_type>]`, then it is preferable over using a TCL procedure conversion when processing time has a potential to slow down the run.

The syntax of the enumeration conversion is:

```
%[<enum_type>]
```

The square brackets above are part of the syntax. The `<enum_type>` is an identifier of the enumeration type. It must follow SDL lexical requirements for an identifier. It is defined by an enumeration definition as follows:

```
enum <enum_type>
{
```


© 2021 All Rights Reserved.

```
    <string> [=<value>],
    <string> [=<value>],
    ...
};
```

The *<string>* is either an identifier or an arbitrary string enclosed in double quotes.

The *<value>* must be an unsigned integer value, upto 64 bits.

A value that is not covered by the enumeration definition will be formatted as a decimal integer. The enumeration definition might appear anywhere inside the SDL program, except inside state statements.

The following example defines weekday as *<enum_type>*:

```
enum weekday {monday, tuesday, wednesday=3, thursday, friday };
```

In this example, the enumeration values of weekday are defined as:

```
monday=0
tuesday=1
wednesday=3
thursday=4
friday=5
```

Suppose, this enumeration is used in the following statement:

```
display ("%[weekday]", day[2:0]);
```

If `day[2:0]` has the value `3` then the attribute value will be formatted into the string "wednesday", but the value 2 will be formatted into the string "2".

Complex Formatting Using the TCL Programming Language

`%[...]` enables use of the full programming power of TCL scripting to further format the results, and even add information such as time of day. The string inside the square brackets is first formatted by performing the regular conversions on the specifiers: `%h`, `%d`, `%i`, `%o`, `%b`, `%c`, `%s`, and the resulting string is evaluated as a TCL command. The entire string starting from the `%[` and ending with the `]` is then substituted with the result returned by the TCL command. The command is typically a TCL proc that you have defined, but you can also use generic TCL commands.

The following example prints the arithmetic sum of `A[7:0]` and `B[7:0]`:

```
DISPLAY ("A+B=%[expr %d + %d]",A[7:0], B[7:0]);
```

With this example, if `A[7:0]` represents the value `12` and `B[7:0]` represents the value `7`, then the above statement will produce the following message:

```
A+B=19
```

XEL commands are not allowed to be called from a `%[]` specifier.

© 2021 All Rights Reserved.

Controlling Field Size and Alignment

By default, the field size (number of characters) allocated when substituting an argument is set to the maximal possible number of characters that the argument might require. For example, a `%b` conversion for an 8 bit argument will take 8 characters, while a `%d` conversion will take 3 characters (because the maximum decimal value of an 8 bits argument is 255). A `%i` conversion will take 4 characters because it also needs to account for a sign character.

The value is right justified in the field. Binary, octal and hexadecimal conversions are padded with zeros on the left, while decimal values are padded with spaces. This padding can be eliminated and the result forced into the minimum required field by inserting a `0` after the `%`.

Additional padding with spaces (regardless of conversion size) can be added by specifying a minimum field size as a decimal number that follows the `%`. If preceded by a `-` character, the value will be left justified (padded with spaces on the right). Otherwise, it is right justified. This applies to all the escape sequences that start with a `%`.

The complete escape sequence from left to right is as follows:

1. The character `%`
2. Optional `-` (specifies that the result should be left justified)
3. Optional `0` (specifies that any leading zeros or spaces from a numeric value should be stripped off)
4. Optional minimum field size (decimal number)
5. The format specifier character or sequence from Table 43-5

For example, suppose that the 8 bit argument `data[7:0]` represents the decimal value `12` (binary `1100`). This will produce the following output depending on the format specifier as follows:

- `%b` print the string `00001100` in an 8 character field
- `%0b` print the string `1100` in a 4 character field
- `%9b` print the string `00001100` right justified in a 9 character field
- `%-9b` print the string `00001100` left justified in a 9 character field
- `%09b` print the string `1100` right justified in a 9 character field
- `%-09b` print the string `1100` left justified in a 9 character field

The following example shows a complete display statement:

© 2021 All Rights Reserved.

```
DISPLAY ("DATA=:%d:%0d::%9b:%-06b:",
        data[7:0],data[7:0],data[7:0],data[7:0]);
```

This will produce the following output:

```
DATA=: 12:12:: 00001100:1100 :
```

Compatibility with Verilog Syntax

The conversion rules of the format specifier are similar to those used by the Verilog `$display()` system task, with the following exceptions:

- In SDL, `%t` is not matched with any argument. In Verilog it must be matched with the `$time` argument.
- Verilog does not support the `%i` specifier (signed integer).
- Verilog does not support the `%[<tcl_command>]` and `%[<enum_type>]` specifiers.
- Verilog does not support field size and field alignment modifiers, although it supports the '0' modifier (no padding).

Default Formatting when a <format_specifier> is Missing

If the <*format_specifier*> argument is an empty string or missing altogether, each generated message is written into a single line of text using a standard format that contains the following fields (space-separated):

<time_stamp> <level> <label> <arg1> <arg2> ...

<*time_stamp*> is written as a plain floating point number (without the time-unit string) representing the simulation time in nanoseconds.

<*level*> is written as an integer decimal value.

<*label*> is the label of the statement, if defined, or the single character * (asterisk) if a label was not defined.

<*arg1*> <*arg2*> ...are space-separated argument values formatted in hexadecimal.

Other Considerations

Messages from `DISPLAY` statements are buffered in the memory of the emulator. While the emulator is running, the buffer contents are dumped into the workstation several times each second, formatted and sent to the console and/or a file. In emulation modes that enable stopping the clocks, the emulator clock will pause if the buffer becomes full before the workstation has a chance to read it. In LA or Dynamic Target mode, if the rate of writing into the buffer is higher than the rate at which the workstation reads it, then some messages are lost. This is called an overflow condition. You can increase the depth of the buffer during compilation using compiler option `sdlDisplayDepth` to address the overflow issue.

© 2021 All Rights Reserved.

The total number of different arguments (`<Arg>, <Arg>, ...`) that can be used in all `DISPLAY` statements depends on the width of the display buffer in the hardware. If the same signal appears as an argument in more than one `DISPLAY` statement in the SDL program, that signal is counted only once for the purpose of calculating the required buffer width. The width of the display buffer can be increased at compile time using compiler option `sdlDisplayWidth`.

Several aspects of the `DISPLAY` facility can be controlled from XEL through the `sdl` command. For example, you can direct the messages to a file instead of (or in addition to) the console.

TX

The `TX` statement is used for recording transactions into an SST2 database. The `TX` statement can define a transaction's beginning, ending, or additional attributes (such as, signal values associated with the transaction).

Terminology and Basic Concepts

This section explains the following terms:

- Stream and Stream Name
- Transaction ID
- Transaction Type
- Transaction Beginning and Ending time
- Event and Error Transactions
- Transaction's Attribute

A *Stream* is a row on the waveform display, on which transactions are displayed. A single stream can show multiple transactions. A stream is identified by a name and a hierarchical scope.

A *Transaction* typically has a beginning time and an ending time, which are paired by a unique identifier called a *Transaction ID*. Some *Transactions*, called *Event* or *Error* have zero duration, which means that their beginning and ending happen at the same time point. A transaction is considered as open only during the time period between its beginning and its ending.

The *Transaction ID* is a user given name. It is used for matching a specific open transaction with different `TX` statements that specify various attributes of the same transaction, but triggered at different simulation time. The same *Transaction ID* can be used for several

© 2021 All Rights Reserved.

transactions, as long as they do not have overlapping time. A specific Transaction ID cannot belong to more than one stream or one scope. If the stream scope or stream name are not defined in the `TX` statement, they are inherited from another `TX` definition with same Transaction ID. If none exist then they get default stream and scope name.

A *Transaction Type* is a user given name that identifies the general properties of the transaction.

Each *Transaction* is associated with several distinct pieces of information called *Attributes*. Each *Attribute* is made of two components: *Attribute Name* and *Attribute Value*, both of which are strings.

There are several types of attributes:

- *Attributes* that are pre-defined (for example, the transaction's beginning time).
- *Named Attributes* are defined by the user as arguments inside the `TX` statement using the syntax: `<name> = <value>`. *Named Attributes* can only be defined at the beginning of the transaction.
- *Signal Attributes* are also defined as arguments inside the `TX` statement, and show design signal values, optionally with additional formatting instructions. *Signal Attributes* can be defined at any time while the transaction is open.

Syntax of the TX Statement

The `TX` statement can be used anywhere inside the SDL program to record into the SST2 database the beginning or the ending of a transaction, or append additional data (Signal values) to an open transaction.

A complete transaction is defined by either a single `TX` statement with an *Event* or *Error* specifier, or by two or more `TX` statements, one with the `begin` specifier, followed by another with the `end` specifier with the same *Transaction ID*, and possibly additional `TX` statements in between with the `set` specifier.

Attributes can be defined at any point, either at the beginning, the ending or while the transaction is still open. However, *Named Attributes* can only be defined at the beginning of the transaction (only in the `TX` statements with the `begin`, `event`, or `error` specifier).

```
[<label>:] tx (<specifier>, <tx_id> [, <arg>, <arg>, ...]);
```

The `<label>` is an optional SDL label that can be used for identifying or disabling the statement at the XEL command level. The same syntactical and semantical rules are followed as the labels used for the `trigger`, `exec`, or `display` statements.

- `<specifier>`: Specifies the type of action represented by this statement:

© 2021 All Rights Reserved.

- ❑ `begin`: Indicates the beginning of a transaction.
- ❑ `end`: Indicates the ending of the transaction.
- ❑ `event`: Indicates an *Event* transaction. The `TX` statement defines both the beginning and ending.
- ❑ `error`: Indicates an *Error* transaction. The `TX` statement defines both the beginning and ending.
- ❑ `set`: Sets additional attributes values for a transaction that is already open.

- ■ `<tx_id>`: The *Transaction ID* identifies a specific transaction. This is a user given name that must follow SDL lexical requirements for an identifier.

The `<specifier>` and `<tx_id>` are always mandatory.

The arguments that follow (each of the `<arg>`) are optional. Each is a *Named Attribute*, *Signal Attribute*, or *Transaction Attributes Definition* argument. The *Named Attributes* that are not defined will assume default values as defined below:

Named Attribute

The *Named Attributes* are as follows:

- ■ `type = <type>`

  The *Transaction Type* is a user given name that follows SDL lexical requirements for an identifier. It is used for identifying the type of the transaction in messages and in the SimVision browsers. If not specified, it will be set to the value specified by `<tx_id>`.

- ■ `stream = "<stream_name>"`

  The name of the stream as it appears on the waveform display. If not specified, it will be copied from another `TX` statement with same *Transaction ID*. If none exists, then it will assume the default string `SDL_DEFAULT_TX`.

- ■ `scope = "<stream_scope>"`

  Design scope of the stream as it appears on the waveform display. If not specified, it will be copied from another `TX` statement with same *Transaction ID*. If none exists, then it will assume the default string `SDL_TX_SCOPE`.

- ■ `title = "<title>"`

  In SDI terminology (and on the SimVision TX Explorer window) this is called *Label*. A different word is used here because the term label is already used for other purposes in SDL. The `<title>` is a very short description of the transaction. If not defined, the value will be set to `<type>`.

© 2021 All Rights Reserved.

- `description = "<description>"`

  An arbitrarily long description for this transaction. If not defined, it will be set to an empty string.

Any of the keywords `type`, `stream`, `scope`, `title`, or `description` can be shortened to use just the first few characters. For example, `st` can be used instead of `stream`, and `ti` can be used instead of `title`.

The double quotes can be omitted for the values of `<stream_name>`, `<stream_scope>`, `<title>`, or `<description>` if they conform to the definition of a legal SDL identifier.

Named Attributes can only be defined with specifiers: `begin`, `event`, or `error`.

Signal Attribute

The *Signal Attributes* are as follows:

- `<signal_name>`

  This is a transaction attribute that shows a signal value. The `<signal_name>` is the full hierarchical name of a design signal. The signal name serves as the attribute name. The signal name along with its value is displayed on the screen in hexadecimal format as an attribute of the transaction.

- `{"<format>", <signal_name>}`

  This is a transaction attribute that shows a signal value. The `<signal_name>` is the full hierarchical name of a design signal. The signal name serves as the attribute name. The signal name along with its value is displayed on the screen as an attribute of the transaction. The `<format>` is a string that specifies how to format the value of the signal into a string.

- `{"<attribute_name>", "<format>", <expression>, <expression>, ...}`

  This is a transaction attribute that shows a string value that was constructed from zero or more expressions. The `<attribute_name>` is the name of the attribute to be displayed in SimVision. The `<expression>, <expression>, ...` specifies zero or more legal SDL expressions. The `<format>` is a string that specifies how to format the expressions into a single string.

Transaction Attributes Definition

The *Transaction Attributes Definition* is as follows:

- `tdef = <tdef_id>`

© 2021 All Rights Reserved.

This argument enables to define a set of attributes through a separate statement and use it inside the `TX` statement. See *Defining Attributes with a tdef Statement* for more details.

Any number of *Attribute* arguments might appear in a single `TX` statement. However, the `type`, `stream`, `scope`, `title`, and `description` arguments might appear only once each, and only if the `<specifier>` is `begin`, `error`, or `event`.

Formatting of Attribute Values

Formatting in the `TX` statement (the `"<format>"` string) follows the same syntax and semantics as the format string in SDL `display()` statements. For example, `"%d"` will format the argument using unsigned decimal conversion.

Defining Attributes with a tdef Statement

Default values can be defined for any set of attributes in any `TX` statement using the `tdef` construct. Either *Named Attributes* or *Signal Attributes* can be defined with `tdef`. The `tdef` statement must appear above any `TX` statement where it is being used, and outside any `STATE` statement. If defined inside an `INSTANCE` statement, then its scope is limited to that instance.

The syntax of the `tdef` statement is as follows:

```
tdef <tdef_id>
{
   <arg>,
   <arg>,
   ...
};
```

The `<tdef_id>` is an identifier, which must follow SDL syntax requirements for an identifier.

The `<arg>` is same as defined for the `TX` statement, and might be any *Named Attribute* or *Signal Attribute*. Any argument defined here will be used in all `TX` actions that contain the `tdef = <tdef_id>` argument.

Transaction Linking

The statement tlink specifies how transactions should be linked using a specified attribute from each transaction as the link tag.

The syntax of the tlink statement is as follows:

```
tlink <tx-id1> <attribute-name1> <tx-ID2> <attribute-name2>;
```

The `<tx-id1>` is the transaction ID of the first transaction to be linked.

© 2021 All Rights Reserved.

The *<attribute-name1>* is the Attribute Name from the first transaction that is used as the link tag.

The *<tx-ID2>* is the transaction ID of the second transaction to be linked.

The *<attribute-name2>* is the Attribute Name from the second transaction that is used as the link tag.

For attributes that look like *<signal-name>* or {"*<format>*", *<signal-name>*}, the Attribute Name is the *<signal-name>* component.

For attributes that look like {"*<attribute-name>*", "*<format>*", *<expression>*, ...}, the Attribute Name is the *<attribute-name>* component.

During the run, every two transactions that have the specified transaction IDs are linked if the specified attributes have the same value, and if the begin time of transaction *<tx-ID2>* happens at or after the begin time of the transaction with *<tx-ID1>*.

The first transaction (with *<tx-ID1>*) is marked as predecessor, and the second transaction (with *<tx-ID2>*) is marked as successor.

The `tlink` statement must appear outside any `STATE` statement.

During run time, for each new transaction being recorded, with either `<tx-ID1>` or `<tx-ID2>`, the software will search for a match starting at the end time of the transaction and going backward in time until another matching transaction is found. At that point the matching transactions are linked, and the search stops.

Examples of Transaction Recording Statements

The following is a simple example that uses only the basic `TX` statement. It defines the beginning and the ending of a transaction using a transaction ID `tx01`. The transaction will appear in Simvision on a stream called `mem`, and will have a transaction type `tx01`.

```
if(top.mem.rd == 'bp)
    tx (begin, tx01, stream=mem,
        title="Memory Read Transaction",
        {"%d", top.mem.address[23:0]}
        ); //begin transaction
if(top.mem.data_ready == 'bp)
    tx(set, tx01, top.mem.data[15:0]); //record data being read as attribute
if(top.mem.rdack == 'bp)
    tx (end, tx01); //end transaction
```

The following example uses the `tdef` statements to simplify the `TX` statements by providing additional details, such as a transaction type, stream scope, a long description, and some *Signal Attributes*.

© 2021 All Rights Reserved.

```
tdef Tw1 {
    type = Twrite,
    stream = "reg",
    scope = "top",
    title = "REG write Transaction",
    description = "Write Transaction into register file from REG",
    top.reg.data[63:0],
    {"%d",top.reg.address[3:0]}
};
tdef Tw2 {
    top.reg.data[63:0],
    {"%d",top.reg.control[3:0]}
};
if(top.reg.wr == 'bp)
    tx (begin, tx02, tdef=Tw1);
if(top.reg.wack == 'bp)
    tx (end, tx02, tdef=Tw2 );
```

The above example is equivalent to the following:

```
if(top.reg.wr == 'bp)
    tx (begin, tx02, type = Twrite,
        stream = "reg",
        scope = "top",
        title = "REG write Transaction",
        description = "Write Transaction into register file from REG",
       {"startData","%d",top.reg.data[63:0]},
        {"%d",top.reg.address[3:0]}
    );
if(top.reg.wack == 'bp)
     tx (end, tx02,{"endData","%d",top.reg.data[63:0]},
        {"%d", top.reg.control[3:0]}
        );
```

Example of using Transaction Linking

The following example shows two transactions with transaction ID `tx01` and `tx02`. Transaction `tx01` has an attribute named `A[7:0]`, and transaction `tx02` has an attribute named `B[7:0]`. During the run, a transaction with ID `tx01` is linked with a transaction with ID `tx02`, if the value of attribute `A[7:0]` in transaction `tx01` matches the value of attribute `B[7:0]` in transaction `tx02`.

```
tlink tx01 A[7:0] tx02 B[7:0];
if(cond1) tx(begin, tx01,A[7:0], S[15:0]);
if(cond2) tx(end, tx01);
if(cond3) tx(begin, tx02, S[15:0]);
if(cond4) tx(end, tx02, B[7:0]);
```

TRACE_TSM

This action will cause the current clock cycle to be unconditionally traced into the SDL trace memory (which can be later dumped to a file using the `sdl -traceDump` XEL command).

© 2021 All Rights Reserved.

The `TRACE_TSM` action itself is not logged into the SDL trace dump file, and unlike other actions, it cannot be disabled from XEL. (The `sdl -traceOn` XEL command can configure which SDL actions will cause SDL tracing, but it has no effect on `TRACE_TSM`).

Examples: SDL Program

Example 1: A State-Less SDL Program

The following SDL program compares two 8 bit vectors `A[7:0]` and `B[7:0]` as unsigned integer numbers and triggers if the number represented by the vector `A[7:0]` is greater than the number represented by the vector `B[7:0]`.

In addition, the program filters out from captured data all samples in which the low order 4 bits of the vector `A` equal the number 13.

```
if(A[7:0] > B[7:0]) trigger;
if(A[3:0] == 13) no_acquire;
```

An equivalent form of the above, using a different variant of the syntax is:

```
trigger = A[7:0] > B[7:0];
no_acquire = A[3:0] == 13;
```

Example 2: A Single SDL Instance

The following SDL program has two states. It triggers if a specific signal becomes high within 100 cycles or less after `sig1` is high. The search for this pattern is restarted each time when sig1 remains low for 100 cycles or more.

Note: The name of the net tested in the first `IF` statement inside `state Two` is taken from the value of a Tcl variable (called `signame`). This enables to change the name of the net tested from XEL without modifying the SDL file.

```
/* Trigger on sig1 high followed by sig2 high, within 100 cycles.
    The number of cycles is counted by counter1
*/

State One       // wait until sig1 goes high
{
    if (sig1)
    {
        load counter1 99;
        start counter1;
        goto Two;
    }
}

State Two
{
```


© 2021 All Rights Reserved.

```
    if ( $signame && Counter1 > 0 )
        trigger ; // trigger if $signame is high while counting
    if ( sig1 )
        load counter1 99; // restart the counter if sig1 high
    if ( Counter1 <= 0 )
        goto One ;      // start all over again
}
```

Example 3: SDL Program with Two Instances

The following SDL program demonstrates use of different SDL constructs. It has two SDL instances, and the program spans on two files.

The bodies of each of the two state machines are exactly the same text, derived from a single file called `instance.tdf`. However, the two state machines execute differently because they rely on the definition of the alias expr, which gets a different value in each of the two instances. Each of the two instances define also a global trigger and a global acquire.

```
// Global Definition Section
exp1 = counter1 > 0 || top.ctl.xyz; // an alias
exp2 = top.ctl.bus1[7:0] != ’b10011 || top.ctl.sig1 == ’bs; // another alias
address_plus_data = {address[15:0],data[15:0]}; // alias defined by concatenation
counter2 = 100; // define default load value for counter2
glob_trig1: trigger = top.cyx[7:0] == top.fsq[15:8]; // global trigger #1

// begin first instance definition
Instance INST_1;
expr = address_plus_data == 32’h874A_FFFF; // new alias using previous alias
`include ”instance.tdf”

// begin second instance definition
Instance INST_2;
expr = address_plus_data == 0; // new alias using previous alias
`include ”instance.tdf”

------- contents of file instance.tdf start below ------

glob_trig2: trigger = expr & !top.enbr; // global trigger (one for each instance)
acquire = !expr; // global acquire definition (one for each instance)
if(exp2)
    display(”Address=%H Data=%H”,address[15:0],data[15:0]);
// Definition of States
state first {
    load counter2 $cnx; //load counter from a value defined in a Tcl variable
    load counter1 511;
    if (exp1 && !(top.vrx[3:0] || top.vx[7:0] >= top.vy[7:0]))
        goto second;
    else if (expr) {
            if (top.bxx != 1)
                goto second;
            else
                acquire;
    }
    else
    {
        acquire;
        display (”uploading probes. time=%t”);
        exec ”database -upload”;
```


© 2021 All Rights Reserved.

```
    }
}
state second {
    if( {ra,rb,ctr1,ctr2} == 5)
        tsm_trig: trigger;
    if(exp2 == 0 || exp1 && counter2 <= 0) {
        trigger;
        start counter2;
        decrement counter1;
    }
    if(!(top.ctl.bus2[15:8] == 'hps || top.xyz != top.vpprt))
        goto first;
}
```

Importing Breakpoints from the Xcelium Simulator into SDL

In SA mode, you can use the `stop` command to define on the simulator an arbitrary number of breakpoints independent of each other. These breakpoints do not function correctly after swapping the design into the Palladium Z1 emulator if they rely on design objects that reside in the DUT. It is possible to import some of these breakpoints into an SDL program that can run on the Palladium Z1 emulator. The only types of breakpoints that can be imported to SDL are `Object` and `Condition`. The following breakpoint options can be imported into SDL programs:

- `-if`
- `-execute`
- `-skip`
- `-delbreak`
- `-continue`

The breakpoints import operation is initiated either by the XEL command:

```
sdl -simBreakPoints -import
```

or automatically during swap-in by setting up the following run-time parameter:

```
xeset sdlAutoSimBreakpoints 1
```

Importing breakpoints involves more than just creating an SDL file. If the imported breakpoints contain `-skip` or `-delbreak` options, then the skip or delbreak counts are imported into the xeDebug run-time environment as values of Tcl variables. A successful import operation also loads the imported SDL file into the emulator, as if the `sdl -load` command was executed.

© 2021 All Rights Reserved.

Disabled breakpoints are not imported.

Theory of Operation

The `Object` or `Condition` component of the imported breakpoint is combined with the expression of the `-if` component if one was specified, and the result is used to create the `if(<expression>)` of a global SDL `TRIGGER` and/or `EXEC` statements.

The `skip`, `delbreak` and `execute` features are handled by SDL `EXEC` statements, while keeping track of the `skip` and `delbreak` counts is done by elements of a Tcl array. The values in the array represent `skip` and `delbreak` counters and their values are initialized by the `sdl -simBreakpoints -import` command.

Expressions in Breakpoints

When breakpoints are imported to SDL, the following modifications are made to the expressions that come from the `if` component or the main condition of a `Condition` type breakpoint:

- References to signal values in the form `#<signal-name>` in expressions are translated to SDL by removing the `#`. The syntax `[value <signal-name>]` is not supported by the import operation.
- The operators **=** and **===** in the original simulator breakpoints are translated to **==**.
- VHDL STD_LOGIC and integer constants are translated to SDL format (for example, `"10110"` is translated to `'b10110`).
- Not all operators allowed in simulator breakpoints can be imported. For example, XM-Sim enables you to use a multiplication operator inside a breakpoint condition, but this cannot be imported to SDL. See Table 43-4 for a list of operators allowed in SDL conditions.

Restrictions and Error Conditions

All breakpoints are not translated to SDL. The following rules decide which breakpoints are to be imported, and which cause the operation to fail:

- Breakpoints that rely only on design objects located in the TB (or that do not rely on any design object) remain functional on the simulator, and are not imported.
- Breakpoints of type `Condition` or `Object` are imported only if they use signals from the DUT, and expression syntax that can be processed by SDL.

© 2021 All Rights Reserved.

- Breakpoints of the CPF object types are specific changes in isolation rules, retention rules, or power domain states. Only the following options are supported:
  - For the `iso_rule` breakpoint, the supported sub-options are: `-iso_disable` and `-iso_enable`.
  - For the `pdname` breakpoint, the supported sub-options are: `-isolation`, `-pd_off`, `-pd_on`, `-pd_trans`, and `-retention`.
  - For the `sr_rule` breakpoint, the supported sub-options are: `-sr_restore` and `-sr_save`.
- In all other cases, the software makes an effort to determine whether the breakpoint in question continues to operate correctly under the simulator after the swap-in.
  - If it is determined that the breakpoint does not operate correctly, the import operation fails.
  - If the software cannot determine for sure, then only a warning is printed.

Examples

- If a breakpoint contains an `if` component that uses a mix of TB and DUT objects, an error condition will be generated.
- If a breakpoint uses an expression that compares a DUT signal with a value of `unknown` (`X`), an error is generated because Palladium Z1 does not support unknowns. Note that, in SDL `'bX` means `don't care`, while `'bX` in a breakpoint means `unknown`.
- An isolation rule breakpoint in the DUT does not function correctly after swap-in, but only generates a warning because the current implementation cannot determine the location of the isolation rule.
- A delta breakpoint almost always behaves differently after swap-in, unless the entire DUT was input in the form of a gate-level netlist, therefore it generates a warning message.

Error Recovery Methods

By default, any error condition during import of any of the breakpoints causes the whole import operation to fail. If the import is done automatically during swap-in (by using `xeset sdlAutoSimBreakpoints 1`), then the swap-in fails and the system reverts to simulation mode.

You can choose to ignore breakpoints that generate error conditions during the import, and proceed with any breakpoints that can be imported successfully by using the following command:

© 2021 All Rights Reserved.

```
sdl -simBreakpoints -errorok 1
```

This affects all future requests to import breakpoints to SDL. In such case, the same error messages are reported (if any error conditions are found), but they do not cause the operation to fail and an SDL file with any breakpoints that could be imported successfully is generated and loaded into the emulator. This also prevents a swap-in from failing due to errors in automatic import.

Limitations and Differences in Behavior

Even breakpoints imported successfully to SDL might produce a behavior that is not exactly the same as running under pure simulation. Following are the differences:

- The import operation ignores the options `-silent` and `-noexecout,` although all other options are honored.
- When running with Xcelium, a breakpoint is completely deleted when it reaches its `delbreak` count. Under SDL, the corresponding trigger is only disabled when it reaches the delbreak count.
- Tcl variables referenced as `$name` by an `execute` component in the breakpoint reference Tcl variables in the Xcelium Tcl name space, while the imported SDL program will reference Tcl variables in xeDebug's Tcl name space.
- Xcelium breakpoints cause the simulator to stop immediately at the exact simulation time in which the breakpoint's condition becomes true. When using SDL, such behavior is not guaranteed, and the run might stop with some delay (See Fast versus Accurate for more details).
- For a Condition type breakpoint to stop the Xcelium simulator, the expression defined by the condition must be true, and an *event* must happen. The *event* is generated by a change in the value of any object used in the expression, and if the expression contains any bit select of a vector (for example *`name[17]`*), then any change in value of any bit in that vector will also generate an event, whether that bit is included in the expression or not.

  The translated SDL program uses, by default, simpler semantics to reduce the amount of emulation resources. With the simplified semantics SDL will stop the run each time when the result of the condition makes a transition from false to true.

  It is possible to request that the translated SDL use exactly the same event-driven behavior that Xcelium uses for breakpoints with the `sdl -simBreakpoints -exact 1` command.
- Using `uxc_hdl.var` file with xeDebug causes unpredictable behavior. Any breakpoints defined in the `uxc_hdl.var` file might not be imported correctly.

© 2021 All Rights Reserved.

Creating Time Markers and Bookmarks

SDL enables you to generate time markers based on various SDL conditions. The time markers can be used later as bookmarks in the infiniTrace sessions, and/or as visual markers on the Verdi waveform display. These time markers are shown as labeled time markers on the Verdi waveform display.

Note: Markers are supported only for the Verdi waveforms and are supported only in STB and LA mode.

The markers are written into one or more files for later use in the infiniTrace run and the waveform display. Each marker contains an identifier (string defined by you) and the time information. It is written into several files as follows:

1. If a regular FSDB waveform database is open, the marker is written into a *Marker* file with a name that is derived from the name of the waveform database.
2. If a FSDB streaming waveform database is open, the marker is written into a marker file with a name that is derived from the name of the streaming waveform database.
3. The marker is also written into a *Bookmark* file in a format that is compatible with the XEL command: `bm -import <file>`. The bookmark file enables the marker to be used later in infiniTrace run.

Generating Markers in Markers File

The marker file is generated for both regular and streaming probes. You must ensure that the FSDB waveform database is already open for write operation before executing the `run` command. When a run is started with an SDL program that contains a `DISPLAY` action with the `F=marker` argument, the marker file is opened for write operation for the waveform database. Subsequent runs will append to the existing marker file until the waveform database is closed.

When the FSDB waveform database is opened for write operation, if a prior marker file with the same name as that of the database exists, the marker file is deleted.

Syntax:

SDL generates a time marker each time when a DISPLAY action with *`F=marker`* parameter is executed. The syntax to specify the marker is as follows:

```
display (F=marker, "<marker-ID>");
```


© 2021 All Rights Reserved.

When the `DISPLAY` action is executed, a marker with the name <*marker-ID*> is written to the designated marker and bookmark files.

The keyword `marker` is case insensitive. No other arguments are allowed inside the parenthesis. <*marker-ID*> must follow regular SDL conventions for an identifier — it can contain only underscore or alphanumeric characters and should not start with a digit.

SDL `DISPLAY` actions can possibly try to generate multiple markers with conflicting data. The markers are processed according to certain rules. The rules are as follows:

- If you have multiple markers with same time and different identifiers, only one of the markers is written into the file, and the rest are discarded.
- If you have multiple markers with same identifier and different time, then the first marker with earliest time is written with its original identifier. The subsequent markers are written with identifiers that are composed from the original identifier plus a unique suffix. The suffix is an underscore followed by an integer. For example, if the marker identifier in the SDL program is `ABC`, then the first marker is written as `ABC`, the second as `ABC_1` the third as `ABC_2`, and so on.

Viewing Markers in Verdi Waveforms

When the Verdi waveform is brought up through the xeDebug GUI, the marker file for regular probes is read automatically and displayed on the first Verdi waveform window each time after new probed data is read from the DCC. The first Verdi waveform window is referred to as the one that was first opened by xeDebug. The markers are displayed only in the first window and do not affect other waveform windows that you might open later.

If you need to display the markers on other waveform windows, you must do that manually from the waveform window menu. When manually reading the marker file into the Verdi waveform, the markers will show the correct time only if the selected time unit in the waveform is ns. This is due to a limitation of the Verdi tool. When the marker file is read automatically (that is, controlled by xeDebug) into the first Verdi Waveform, xeDebug ensures that the correct time unit is used.

To ensure automatic loading of the marker file for regular probes in the Verdi viewer, you need to add probes with the `probe -waveform` command in the xeDebug GUI. The Verdi waveform does not display the markers when the signals are not selected from the FSDB file. This is a known limitation of Verdi. To see the markers, you must have some signals in the display. With the `probe -waveform` command, the signals and markers will be displayed in the waveform. For example:

```
xeDebug -gui -fsdb -input <qel_file>
database -open <database_name>
probe -waveform <object_name(s)>
```


© 2021 All Rights Reserved.

The marker file is created from SDL DISPLAY action with `F=marker` parameter where the time unit is ns.

```
$ cat trace.rpt
bk2 105
bk1 120
bk1_1 125
bk1_2 130
bk1_3 135
bk1_4 140
```

Waveform markers file are not loaded automatically for streaming probes in the Verdi viewer. These files need to be loaded manually in the Verdi viewer, as follows:

1. From the Verdi menu bar, select the *waveform->marker* option that opens the marker dialog box.
2. In the marker dialog box, click *Restore* and select the marker file.

Generating Markers in Bookmark File

The bookmark file is opened for write the first time when xeDebug uses an SDL program that contains the `display` (`F=marker`, "<*marker-ID*>") action. The bookmark file remains open and is appended with new markers until xeDebug exits. The default name of the generated bookmark file is `sdl.bookmark`.

Example 1: Generating sdl.bookmark

The SDL file `x.tdf` contains the following:

```
State s1 {
     display(F=marker, "bk1");
     goto s2;
}
State s2 {
      display(F=marker, "bk2");
      goto s3r;
State s3 {
       display(F=marker, "bk2");
       trigger;
}
```

You execute the following XEL commands:

```
debug .
host .
download
database -open xyz
sdl -enable
sdl -setfile x.tdf
run
```


© 2021 All Rights Reserved.

A bookmark file with the name `sdl.bookmark` and a marker file with name `xyz.rpt` will be created. These files will have the following markers:

```
Identifier         Time

bk1          1
bk2          2
bk2_1        3
```

Example 2: Using sdl.bookmark in InfiniTrace Session

You can load the bookmark file later in the InfiniTrace session:

```
debug .
host .
download
configPM -stb
bm -import sdl.bookmark
infiniTrace -observe <prepare_session_name>
infiniTrace -goto bk2
......
infiniTrace -goto bk2_1
```

Additional Tips

This section lists some tips that you can use when using SDL.

Getting Live Feed Of Display Messages

The `DISPLAY` feature enables you to route different messages to different files based on the `F=<ID>` argument in the `display` statement. This enables to get a copy of a select set of display messages, without any other messages.

To see these messages scrolling on the screen at run time, open a separate shell window with the Unix command `xterm`, and in that window, execute the following Unix command:

```
tail -f <filename>
```

This will show you a live feed of the DISPLAY messages that are written to that file.

Using An Additional HDL Module To Track Complex Design Behavior

Some hardware protocols might be too complex for SDL to handle directly.

A way to approach this is to provide a *helper* module, written in Verilog or VHDL, that can monitor the design, and generate signals that SDL can use in conditions. The helper module

© 2021 All Rights Reserved.

has to be compiled together with the design, so it becomes an integral part of the design database. Here is how you can do it without changing anything in the original design:

The helper module should be written in a separate file, and access signals in the original design by using their absolute path.

When using the IXCOM compiler, the file containing the helper module can simply be specified with the `ixcom` command. No other step is needed.

When using ICE flow, use the `import bind` command to add the file to the existing design, as follows:

1. Compile the file containing the helper module into a netlist using HDL-ICE Compiler.
2. Create a new file with the following single line:

   ```
   bind <topCellName> <moduleName> <instanceName>();
   ```

   <*topCellName*> is the name of the top cell in the original design.

   <*moduleName*> is the name of the helper module.

   The helper module will be instantiated with the instance name <*instanceName*> under <*topCellName*>.
3. Include the file containing the `bind` command as well as the netlist file with the helper module that was generated by HDL-ICE Compiler, and the original design netlist as inputs to the `import` command.

You can create any number of helper instances in this manner under the top cell (each will require its own `bind` command).

As an example, consider the following `bind` command:

```
bind mytop Hmod HLP();
```

`mytop` - is the top cell name of the design.

`Hmod` - module name of helper module.

The helper module will be instantiated as `mytop.HLP`.

© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

44

Controlling the Run with Dynamic RTL

Palladium Dynamic RTL (DRTL) is a run-time debugging feature that enables you to write control and monitoring programs in an industry standard HDL (Verilog/VHDL/System Verilog). The DRTL programs are compiled and executed at run time of an emulation session and can also be removed at run time without the need to recompile your design.

You define DRTL modules to identify complex conditions and act upon them, using Verilog, VHDL, or SystemVerilog. DRTL has a similar role and capabilities as SDL, and can be used independently or together with other SDL programs.

You can use DRTL to perform the following tasks:

- Stop the design clocks
- Stop or pause waveform capture
- Insert TCL or XEL debugging commands into the run using the `$qel()` system task. The commands might be arbitrary. For example, forcing the value of a design net, dumping the contents of a memory in a file, and so on.
- Print out formatted messages using standard `$display()` system task.

You can use DRTL in either of the followings ways:

- Use DRTL together with SDL, to track and identify patterns in a design state, which would be difficult to achieve with the SDL syntax alone. In this usage, SDL can reference output ports of the DRTL instances the same way it references the design nets.
- Use the following system tasks in the DRTL modules for enhanced debugging operations:
  - `$display(),$qel("trigger"),$qel("acquire start"), $qel("acquire stop"),$qel("acquire")`
  - `$qel("<CMD>")`, where <*CMD*> is a TCL command

    For more information, refer to the Using System Tasks in DRTL Modules section.

This chapter covers the following topics:

© 2021 All Rights Reserved.

- Supported Modes
- Defining DRTL Modules
- Creating DRTL Instances
- Using DRTL Instances
  - Using DRTL Instances in SDL Programs
  - Using DRTL Output as the Input of Another DRTL Instance
  - Using System Tasks in DRTL Modules
  - Accommodating Large DRTL Instances
- Using Out of Module Reference (OOMR) with DRTL Modules
- Re-Instantiating DRTL Instances
- Saving and Importing DRTL modules
- Resetting the DRTL Instances
- Restrictions and Errors
- Dynamic RTL Example
- Initiating DPI Calls from DRTL Modules
- Analyzing Capacity of DRTL Modules

Supported Modes

DRTL can control data capture in In-Circuit Emulation (ICE) modes (both STB and LA modes) and Simulation Acceleration (SA) mode. It can stop the running design in Vector Debug (VD), STB and SA modes, and schedule execution of XEL or TCL commands during a run in STB, LA, and SA modes.

In both SA and ICE modes, DRTL is fully supported in both CAKE 2X and CAKE 1X modes.

The DRTL modules can be defined and instances can be added in the InfiniTrace Observe sessions as well. The system task `$display()` is supported in the InfiniTrace Observe sessions.

Note: DRTL is not supported in xeDebug offline sessions and vvmDebug.

© 2021 All Rights Reserved.

Defining DRTL Modules

A DRTL module comprises Verilog, VHDL, or SystemVerilog code that is synthesized with the HDL-ICE Compiler and then loaded at run time into the available capacity of the current run-time session. You can define multiple DRTL modules.

Requirements

- The HDL files should contain RTL code written in SystemVerilog, Verilog, or VHDL. Ensure that the RTL code is synthesizable by the HDL-ICE Compiler. Currently, mixed RTL for Verilog and VHDL in one file is not supported.
- You must specify one or more Verilog, VHDL, or SystemVerilog files to define the DRTL module to be used at run time. However, a top module must be present in the files.

To define a DRTL module, use the following command:

```
drtl -definemodule <DRTL_module_name> [-verilog|-vhdl|-sv] <file_list>
[-topmodule <top_module_name>] [-compilescript <compile_script>][-parameter
{<parameter_name>=<value>}
```

If you do not specify the top module name using the `-topmodule` option, the DRTL module name is used as the top module name by default. For example,

```
drtl -define mytest mon.v
```

In the above definition, `mytest` will be used as the default DRTL top module name.

You can also specify your own compile script to compile the DRTL modules. For information on the requirements and format of the compile script, refer to the description of the `drtl -definemodule` command in the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

You can also pass parameters to DRTL Verilog modules in the same way as done in Verilog. Thus, with different parameters, a DRTL module can be reused in different projects as required. Use the `-parameter` option to specify the parameters while defining a DRTL module.

Using Input and Output Ports

The top module might have zero or more input ports and zero or more output ports. When a DRTL instance is created, the input ports are connected to design signals or other DRTL outputs.

© 2021 All Rights Reserved.

If a DRTL module does not have any input ports, then it must have a direct reference to the design signals through OOMR (Out Of Module Reference). The output port of a DRTL instance can be used in SDL like any other design net or be connected to the input port of another DRTL instance when the instance is created.

Note: A DRTL module might also not have any output port. If a DRTL module has no output port defined, it must have either DPI calls or `$display` or `$qel` system tasks.

The following module does not have any output port defined:

```
-------------monitor.v--------------
module ft (clk, in);          // ft is the top module name of the DRTL code
input clk;                   // clk is an input port
input [3:0] in;             //in is an input vector
```

So, you must use either a DPI call or at least one `$display` or `$qel` statement, as follows:

```
always @(*)
    begin
      if(clk && in == 4'b1010)
         begin
         $display("T=%t DRTL: %08b,%08b,%08b", $time, dut.cnt_u0.dat[7:0],
         dut.cnt_u1.dat[7:0], dut.cnt_u2.dat[7:0]);
// Shows time and value of signals when in[3:0] == 4'b1010
// dut.cnt_u0.dat[7:0], dut.cnt_u1.dat[7:0], dut.cnt_u2.dat[7:0]
// are all OOMRs referring to the user design signals
         $qel("trigger");   // Same semantics as SDL trigger
         end
     end
endmodule
```

For information on how to implement DPI calls in DRTL modules, refer to Initiating DPI Calls from DRTL Modules.

The `inout` type is not supported in DRTL module port definition. For example, the following definition will issue an error message while creating DRTL instances because the definition contains an `inout` port:

```
module dtest(input logic d_data, input dclk, input logic d_inp,inout [7:0] d_bidir,
output logic d_outp );
```

For information on how to define the DRTL modules throughout the xeDebug GUI, refer to the description of the DRTL Tab in the Using the xeDebug GUI chapter.

Creating DRTL Instances

A DRTL instance is a replica of a DRTL module with specific port connections. When creating a DRTL instance, you need to connect its input ports to either design signals, output ports of other DRTL instances, or constant values.

© 2021 All Rights Reserved.

An instance is used and downloaded to the emulator only if it is used by the run-time environment. Accordingly, the instance has enabled system tasks, or its outputs are referenced by other components that are used by the run-time environment (SDL or other DRTL instances).

The DRTL circuit runs only as an observer, and cannot directly affect the behavior of the design (although it can affect the running design indirectly through the `$qel()` system tasks). The DRTL instances are run in the available space (that is, the instances are scheduled in remaining logic processors that are available after the design is loaded).

- To use DRTL modules at run time, open the design in xeDebug and issue the `drtl -definemodule` command to load the HDL files and define a DRTL module. The `drtl -definemodule` command reads the specified source files and compiles them.

  Example:

  ```
  drtl -definemodule drtlA monitor.v -topmodule ft
  ```

  The above command compiles the Verilog file, `monitor.v`, and defines a DRTL module, `drtlA`, that has `ft` as the top-level module.

- Create an instance as follows:

  ```
  drtl -addinst drtlA m_inst(dut.clk, dut.cnt_u3.dat[3:0])
  ```

  The above command creates a DRTL instance `m_inst` from the DRTL module `drtlA`. The command also specifies names of the design signals that will be connected to the input ports of the top module `ft`.

Note: Every DRTL operation should be performed after the host is set — that is, every DRTL command must be issued after executing the `host` command.

The `drtl -addinst` command accepts also the Verilog style syntax of named arguments. For example:

```
drtl -addinst drtlA m_inst(.clk(dut.clk), .in(dut.cnt_u3.dat[3:0]))
```

You can also connect a constant value to an input port. For example:

```
drtl -addinst drtlA m_inst(.clk(dut.clk), .in(4'b1010))
```

The constant value must follow the Verilog syntax.

If the DRTL top module has an input vector — for example, `in[3:0]` — to create a DRTL instance, you need to connect a design signal vector with bit range to the corresponding input vector. For example, `in(dut.cnt_u3.dat[3:0])`. Specifying `.in(dut.cnt_u3.dat)` is incorrect because of missing the bit range, `[3:0]`.

© 2021 All Rights Reserved.

DRTL enables you to specify different bit widths for the connected design signals and the input vectors. Missing inputs are replaced by connecting the unmatched MSB input ports to `0`, and unmatched inputs signals are ignored at the MSB side.

Example 1

```
.in(dut.cnt_u3.dat[5:1])
```

The extra MSB `dut.cnt_u2.dat[5]` will be ignored.

```
hookup signal(1): dut.cnt_u2.dat[4] -> in[3]
hookup signal(2): dut.cnt_u2.dat[3] -> in[2]
hookup signal(3): dut.cnt_u2.dat[2] -> in[1]
hookup signal(4): dut.cnt_u2.dat[1] -> in[0]
```

Example 2

```
.in(dut.cnt_u3.dat[3:2])
```

Here, the MSB of `in` will be extended by `0`.

```
hookup signal(1): 0 -> in[3]
hookup signal(2): 0 -> in[2]
hookup signal(3): dut.cnt_u0.dat[3] -> in[1]
hookup signal(4): dut.cnt_u0.dat[2] -> in[0]
```

If a constant value is bound to the input of the DRTL top module, the value is converted into binary value and considered as a vector.

Use the following commands to enable all DRTL instances, and then run:

```
drtl -enable *;   # Enables system tasks($display, $qel) in the DRTL instances.
run 3000;   # Runs the design
```

Sample Running Result

```
T=6196 DRTL: 00000000,00000000,00011001
INFO (legacy-49994): Run stopped by SDL Trigger DRTL:m_inst (triggered at 6196
fclk)
```

Using DRTL Instances

The output ports of a DRTL instance represent different condition results. You refer to them with the instance name followed by the port name, separated by a period. For example, if you define and create a DRTL instance `DRTL_memory_violation`, which has single bit output `out`, you refer to the output by specifying `DRTL_memory_violation.out`. DRTL instances can either pass data through their output ports used as an input by other DRTL instances or SDL programs, or an instance can be self-contained and execute system tasks.

© 2021 All Rights Reserved.

Using DRTL Instances in SDL Programs

When you need to deal with complex logic that is not practical to implement as an expression in SDL, you can choose to implement the logic in DRTL, and use its outputs in SDL. In SDL programs, use the output ports of DRTL instances instead of design signals.

For example, if `drtl_inst.out1` refers to the output port `out1` of the DRTL instance `drtl_inst`, use `drtl_inst.out1` in your SDL program.

Example 1

```
if( drtl_inst.out1 )
trigger;
if( drtl_inst.out2 )
EXEC "force Top.CPU.stop 1";
```

Example 2

```
State S1
{
       if( DRTL_memory_violation.out )
            trigger;
}
```

Using DRTL Output as the Input of Another DRTL Instance

You can use the output of a DRTL instance as an input of another DRTL instance.

Important

Ensure that the DRTL instance for which the output is being used as the input already exists.

Example:

```
drtl -addinst myDRTL_B mydrtlB(.in(DRTL_memory_violation.out))
```

In the above example, `DRTL_memory_violation` should exist. Otherwise, an error message will be issued. Here, a reference relationship is created between the DRTL instances `mydrtlB` and `DRTL_memory_violation`.

When two DRTL instances have a reference relationship, you need to be cautious while removing the instances. For example, you cannot remove `DRTL_memory_violation` because `mydrtlB` exists and uses `DRTL_memory_violation.out` as input. In such cases, a warning is displayed.You must remove `mydrtlB` first and then remove `DRTL_memory_violation`. Or you can remove `mydrtlB` and

© 2021 All Rights Reserved.

`DRTL_memory_violation` simultaneously using the `drtl -rminst` command, as follows:

```
drtl -rminst DRTL_memory_violation mydrtlB
```

Using System Tasks in DRTL Modules

VXE supports the following system tasks in the DRTL modules:

- `$display(…)`
- `$qel(…)`
- `$fwrite (…)`
- `$fopen (…)`
- `$fdisplay (…)`

These system tasks offer powerful debugging operations that can be done in the same manner as done using SDL, such as trigger and acquire. You can also print DRTL messages on the console or write them into log files.

If a DRTL instance has system tasks, the system tasks will be disabled by default immediately after the instance is created. You can enable the system tasks using the following command:

```
drtl -enableSysTask <instance_name>
```

You can change the default settings for enabling system tasks for all DRTL instances using the following command:

```
drtl -systemTasksDefault [enable | disable | 1 |0]
```

To view the status of the system tasks for all instances, execute the `drtl -listinst` command.

Printing Messages on Console

The `$display` system task is a standard system task in Verilog that is used to print messages on the console. `$display` is supported in the DRTL modules. You can specify any format string supported by the HDL-ICE Compiler except some special cases. For example:

```
$display("T=%t DRTL: %08b,%08b,%08b", $time, dut.cnt_u0.dat[7:0],
dut.cnt_u1.dat[7:0], dut.cnt_u2.dat[7:0]);
```


© 2021 All Rights Reserved.

The following table lists the supported formats for `$display` system tasks:

Table 44-1 Supported $display Formats

Format	Description
`%h %H`	Display in hexadecimal format
or	
`%x %X`	
`%d %D`	Display in unsigned decimal format
`%o %O`	Display in octal format
`%b %B`	Display in binary format
`%c %C`	Display in ASCII character format
`%m %M`	Display hierarchical name
`%s %S`	Display as a string
`%t %T`	Display in current time format
`\n`	The newline character
`\t`	The tab character
`\\`	The \ character
`\"`	The " character
`%%`	The % character

The following table lists the unsupported formats for `$display`:

Table 44-2 Unsupported $display Formats

Format	Description
`%l %L`	Display library binding information
`%v %V`	Display net signal strength
`%u %U`	Unformatted 2 value data
`%z %Z`	Unformatted 4 value data
`%e %E`	Display real numbers in an exponential format

© 2021 All Rights Reserved.

Table 44-2 Unsupported $display Formats

Format	Description
`%f %F`	Display real numbers in a decimal format
`%g %G`	Display real numbers in exponential or decimal format, whichever format results in the shorter printed output
`%p or %P`	Display as an assignment pattern

Executing TCL Commands from DRTL Designs

The `$qel` lets you execute almost any TCL command from inside the DRTL designs. `$qel` accepts a text string as the first argument. The string is the complete command line for the TCL command.

For example:

```
$qel("signal -force top.reset 1")
```

`$qel` system tasks with the following strings have a special meaning to the system and do not represent TCL commands. They are used to generate triggers (with same semantics as the trigger in SDL), and to control waveform data acquisition.

- `$qel ("trigger")`: Causes an SDL trigger
- `$qel ("acquire")`: Captures the current cycle when the driving condition changes from false to true.
- `$qel ("acquire stop")`: Stops the waveform data acquisition when the driving condition changes from false to true.
- `$qel ("acquire start")`: Starts the waveform data acquisition when the driving condition changes from false to true.

Note: It is possible that one or two extra probe sample might be added to the end of each run, even if that time segment was not supposed to be captured due to conditional acquisition.

Tcl procedures are also supported inside `$qel` task in DRTL modules. The infrastructure of DRTL `$qel` code is same as that of SDL EXEC statement. Some XEL commands are not allowed, whether called directly or indirectly from a TCL script. For the list of excluded commands, refer to the EXEC section.

At the beginning of each run, waveform acquisition may be either turned on (started state) or off (stopped state). If SDL is not used, each run starts with acquisition turned on. If SDL is

© 2021 All Rights Reserved.

used and the SDL program contains an acquire action, then the run will start with acquisition in a stopped state.

Writing Formatted Data to Files

VXE supports `$fdisplay` and `$fwrite` Verilog system tasks in DRTL modules. Both these system tasks are used to write formatted data to files. Both `$fdisplay` and `$fwrite` insert a new line after every execution.

You can log various DRTL messages to different log files using these system tasks.

You can print messages either on the console window or output the messages to a specified file. Additionally, different display messages can be printed to different files from one DRTL module.

Steps

Follow the given steps to use these system tasks in DRTL Verilog files:

1. Declare the file handle as an integer:

   ```
   integer handle;
   ```

2. Use `$fopen` in initial block. This task returns the file handle by passing the file name as an argument. Specify the file name as the argument for `$fopen`.

   ```
   handle = $fopen(“fileName”);
   ```

3. Use `$fdisplay` or `$fwrite` to print out messages to files:

   ```
   $fdisplay(handle, "The binary value of dreset is: %B", dreset) ;
   $fwrite(handle, "The octal value of dreset is: %O\n", dreset) ;
   ```

The following code shows an example of a DRTL module with `$fdisplay` and `$fwrite` system tasks:

```
module dtest(input dclk, dreset, [5:0] d_in, input d_ld, output reg
[5:0]d_out);
integer handle1, handle2, handle3, handle4;
initial
begin
   handle1 = $fopen("file1");
   handle2 = $fopen("file2");
   handle3 = $fopen("file3");
   handle4 = $fopen("file4");
end

always @(posedge dclk) begin
d_out <= d_in;

$display("dclk = %b dreset = %b d_in = %b d_ld = %b d_out = %b ",dclk, dreset,
d_in,d_ld, d_out);
$display("The decimal value of dreset is: %d", dreset) ;
```


© 2021 All Rights Reserved.

```
$fdisplay(handle1, "The octal value of dreset is: %o", dreset) ;
$fdisplay(handle2, "The binary value of dreset is: %b", dreset) ;
$display("The hexadecimal value of dreset is: %h", dreset) ;
$fdisplay(handle2, "The hexadecimal value of dreset is: %H", dreset) ;
$display("The binary value of dreset is: %B", dreset) ;
$fwrite(handle3, "The octal value of dreset is: %O\n", dreset) ;
$fdisplay(handle4, "The decimal value of dreset is: %D", dreset) ;
end
endmodule
```

Output

If the specified file name already exists, the original file data will be overwritten with new messages. If a file does not exist with the specified name, a new file will be created. This file will be closed automatically after debug process finishes — you need not call `$fclose` in `always` block.

If both `$fdisplay` or `$fwrite` use the same file handle as input, the messages are appended to the same file. The messages are appended to the file in one debug session, even if run multiple times.

The messages from `$display` are printed only on console. The messages from `$fdisplay` and `$fwrite` are displayed on the console and also saved to a specified file by default.

You can change this setting using the `-displayconsole` option of the `drtl` command, as follows:

```
drtl -displayconsole [enable | disable | 1 | 0]
```

- `disable` or `0`: Shows the messages in a file for `$fdisplay` and `$fwrite` and not on console.
- `Enable` or `1`: Shows the messages in the file and also prints on the console.

The `drtl -displayconsole` command affects all DRTL instances. You can change this option before each run.

Example: DRTL Module with System Tasks

```
module sub(input clk);
parameter p5 = 100;
always @(negedge DRTL.RESET)
$display("p5 = %0d",p5);
endmodule
module dtest(input clk, input int cnt);
parameter p1 = 4;
parameter p2 = 9;
parameter p3 = 14;
parameter p4 = 19;
int acq;
integer noacq;
```


© 2021 All Rights Reserved.

```
sub s1(clk);
initial
begin
   acq = $fopen("acq_file");
   noacq = $fopen("noacq_file","a+");
end
always @(posedge clk) begin
   if(((cnt >= p1) && (cnt <= p2)) || ((cnt[31:0] >= p3) && (cnt[31:0] <= p4)))
begin
   $qel("acquire start");
   $fdisplay(acq,"cnt = %0d \\ acquiring started \t @ time = %0t
\n",cnt[31:0],$time)
end
else
   begin
   $qel("acquire stop");
   $fdisplay(noacq,"cnt = %0d \" acquiring stopped @ time = %0t
\n",cnt[31:0],$time);
   end
end

always @(negedge DRTL.RESET)
begin
   $qel("acquire stop");
   $fwrite(noacq,"acquiring stopped %% "," at negedge DRTL.RESET @ time = %0t
\n",$time);
   $display("p1 = %0d p2 = %0d p3 = %0d p4 = %0d",p1,p2,p3,p4);
end
endmodule
```

Limitations

- `$fdisplay` is not supported in DRTL modules which use VHDL.
- `$fwrite` is partially supported. The following format without "`\n`" is not supported:

  ```
  $fwrite(fhandle, "messages...");
  ```

  `$fwrite` always inserts a new line after every execution.
- Only `$fopen` is supported in `initial` block. All other statements in initial block have no effect in DRTL modules. For details, refer to Restrictions and Errors.
- `$fopen` ignores the second argument if exists. The default action is to write to a specified file. The append and other file open modes are not supported.

Accommodating Large DRTL Instances

Large number of DRTL instances need xeDebug to generate logic to implement the instances. The logic is added through a dynamic netlist, which uses the free resources that are not occupied by design logic. The number of gates to be added for DRTL instances is limited to the availability of free resources.

© 2021 All Rights Reserved.

Use the following `compilerOption` command in your compile script to reserve resources at run time:

```
compilerOption -add {DynamicNetlist <N>}
```

For more information, refer to the description of the `compilerOption` command in the *User Data Commands* chapter of the *VXE Command Reference Manual*.

Using Out of Module Reference (OOMR) with DRTL Modules

The DRTL modules can reference design signals through Out of Module Reference (OOMR). You can use OOMR with the DRTL modules to reference external signals without going through the I/O port of the module. For example, you can directly use the design signal `dut.cnt_u0.dat[7:0]` in DRTL modules.

The following are also supported while using OOMR with DRTL modules:

- Escaped names
- Concatenated signal names (that is, multiple signals separated with commas, inside curly braces { })

Using OOMR simplifies the DRTL code. You can avoid defining input ports in the DRTL top module and connecting the design signals to the input ports of a DRTL instance.

However, with OOMRs, you cannot apply the DRTL module to different places of the design. For example, if you write a PCIe bus monitor in a DRTL module with no OOMRs. Then, you can instantiate the monitor multiple times to monitor different PCIe buses in the design.

The following table lists the OOMRs support for different object types:

Table 44-3 Support for OOMRs with DRTL Modules

Object	Support
Objects are nets or variables of a scalar or vector type	Supported
Objects are a parameter, function, or task	Not Supported
Multidimensional arrays	Not Supported
Out of module references to memory nets or variables	Not Supported

© 2021 All Rights Reserved.

Limitations

- If an OOMR is a design vector signal:
  - You should explicitly specify the bit range in DRTL modules. For example, `dut.cnt_u0.dat[7:4]` is correct and `dut.cnt_u0.dat` is incorrect.
  - You cannot use a loop index as bit range in DRTL modules. For example:

```
always @(posedge clk) begin
for(int i=0;i<3;i++) $display("time = %t, cnt_u0.dat[%0d] =%d", $time, i,
dut.cnt_u0.dat[i]);
end // incorrect
```

Caution

The above example shows a loop index `i` ***as bit index of OOMR,*** `dut.cnt_u0.dat`***. The compiler will not generate any error, however, using a loop index as bit range in DRTL modules is illegal and you will not get expected results because DRTL cannot determine the valid bit range for OOMRs.***

- When using a top-level signal name as an OOMR, you should prefix `DESIGN_TOP` before the top-level OOMR. For example:

```
DESIGN_TOP.clk // correct
clk //incorrect
```

Re-Instantiating DRTL Instances

You can change the port connections of an existing DRTL instance. To do this, define again the instance with the option `-overwrite` in the `drtl -addinst` command.

For example:

```
drtl -definemodule drtlA monitor.v -topmodule monitor
drtl -addinst drtlA m_inst(.clk(dut.clk), .in(dut.cnt_u3.dat[3:0]))
```

To reconnect `dut.cnt_u3.dat[6:3]` to `in[3:0]`, issue the following command:

```
drtl -addinst -overwrite drtlA m_inst(.clk(dut.clk),.in(dut.cnt_u3.dat[6:3]))
```

This command does not affect the internal behavior of DRTL instances and the use of `m_inst` by other SDL or DRTL instances.

© 2021 All Rights Reserved.

Saving and Importing DRTL modules

Defining DRTL modules, loading the DRTL files, and compiling the files to get the DRTL module netlist cost time. To save time, you can save a defined DRTL module and use it later without recompiling the module. Use the following commands:

```
drtl -save [-overwrite] <file_name> [DRTL_module_name_pattern_list]
drtl -load [-overwrite | -ignore] <file_name> [DRTL_module_name_pattern list]
```

For example:

```
drtl -save save.txt dtop1 dtop2 mtop? moni*
drtl -load -overwrite mtop ? moni*
or
drtl -load -ignore moni*
```

The `drtl -save` command enables creation of libraries of compiled DRTL modules that can be reused once they are created. The pattern list could also be empty — which is equivalent to specifying the wildcard character *. If the pattern list is not empty, it must match the defined modules in `drtl -save` command. The save operation fails if a file with the same name already exists and you do not specify the `-overwrite` option.

Note: Only DRTL modules can be saved. You cannot save the instances.

To load modules from a specified file, use the `drtl -load` command. All modules matching a specified module pattern will be loaded from the specified file. If the pattern is not specified, all modules are loaded from the file. Loading a module fails:

- If none of the modules in the file match the specified pattern — use the `-overwrite` option to overwrite the existing DRTL modules.
- If you use an existing module name — use the `-ignore` option to skip loading modules if there are already existing modules with same name.

Resetting the DRTL Instances

You can reset the DRTL instance circuit into a specific state at the beginning of each run, using a predefined signal called `DRTL.RESET`.

You can either connect an input port of the DRTL instance to `DRTL.RESET`, or use `DRTL.RESET` directly inside the DRTL module definition. The `DRTL.RESET` signal starts high at the beginning of each run for a few cycles. After a few cycles, the signal remains low throughout the rest of the run.

© 2021 All Rights Reserved.

Note: This behavior does not apply if the run is initiated with the `run -continue` command. In such cases, `DRTL.RESET` remains low from the first cycle. Use the `DRTL.RESET` signal to reset the state of the DRTL instance, according to your requirement.

To detect `DRTL.RESET`, you should use `negedge DRTL.RESET`. For example:

```
1 module dtest( dclk, d_addr, d_flag, d_out);
2  input dclk;
3  input [11:0] d_addr;
4  output [4:0] d_out;
5  input d_flag;
6
7  always @(negedge DRTL.RESET) begin
8      $display("DRTL START, %d", d_flag);
9      /* do reset actions */
10  end
11
12  always @(posedge dclk) begin
13      if (d_addr[4:0] == 0) begin
14          $display("%d: time=%t, count0=%d, dout=%d", d_flag, $time,
testbench.dut.count0[7:0], d_out[4:0]);
15      end
16  end
17 endmodule
```

Restrictions and Errors

- DRTL restricts using both Verilog and VHDL code in the same HDL file.
- If the DRTL code is written in multiple files, the redefinition of the modules in multiple files is not permitted.
- The DRTL code must be synthesizable by the HDL-ICE Compiler.
- The system does not support DRTL modules without external output and system tasks. For example, defining the following DRTL module will issue an error message:

```
module ft (clk, in); // No external output
input clk;
input [3:0] in ;
always @(*)
     begin
        if(clk && in == 4'b1010)
           begin //No system task
        end
       end
endmodule
```

- The condition detected by DRTL has a two cycle delay.
- Some formats in `$display` are not supported even though the HDL-ICE Compiler reports no errors. For information on the supported and unsupported `$display` formats, refer to Supported $display Formats and Unsupported $display Formats.

© 2021 All Rights Reserved.

- `$qel` does not accept extra arguments.
- `$display` system tasks require hardware resources that are shared with SDL. By default, the total number of `$display` system tasks and the number of SDL global display is limited to 256. However, you can increase the number with the compiler option `sdlGlobalDisplay`.
- By default, the total number of bits in arguments of all `$display` system tasks and SDL display actions is limited to 2048. However, you can increase the number with the compiler option `sdlDisplayWidth`.
- The maximum allowed total number of `$qel` general commands and SDL global exec actions is 1024.
- The maximum allowed total number of `$qel("trigger")` and SDL global trigger actions is 1024.
- Only `$fopen` is supported in Verilog initial blocks in DRTL modules. All other statements in an initial block have no effect in DRTL modules. The run-time software displays a warning message if you define a DRTL module with code including initial blocks. Parallel blocks are also not supported in the DRTL code.
- DRTL input and output ports must be single bit or vector. `struct` or `union` type as input and output ports are not supported.
- Currently, DRTL does not support loading and parsing of RTL library file, VHDL packages, and include files.

Dynamic RTL Example

The following example explains the usage of DRTL using Verilog code:

riscMon24.v

```
module riscMon(clk, rst, data, ld, PC,OP, LOAD);
     input clk, rst;
     input [8:0] data;
     input ld;
     output [5:0] PC;
     output [2:0] OP;
     output LOAD;
     reg [5:0] rPC;
     reg [2:0] rOP;
     reg rLD;
     reg [3:0] currentState;
     reg [3:0] nextState;
     localparam STATE_INIT = 4'b0000;
     localparam STATE_LOAD = 4'b0001;
     localparam STATE_FETCH = 4'b0010;
```


© 2021 All Rights Reserved.

```
     localparam STATE_EXECUTE= 4'b0011;
     assign PC = rPC;
     assign OP = rOP;
     assign LOAD = rLD;
always @(posedge clk or negedge rst)
begin
       if (rst == 1'b0 )
           currentState <= STATE_INIT;
       else
           currentState <= nextState;
end
always @( * )
begin
     case (currentState)
     STATE_INIT: begin
     if ( ld == 1'b1) begin
     nextState = STATE_LOAD;
     end
end
STATE_LOAD: begin
     rPC = data[5:0];
     rOP = data[8:6];
     rLD = ld;
     nextState = STATE_FETCH;
end
STATE_FETCH: begin
if ( ld == 1'b0) begin
nextState = STATE_INIT;
end
end
endcase // case (currentState)
end
endmodule
```

- To read the HDL file `riscMon24.v`, compile it to create a DRTL module, and create an instance in an xeDebug session, use the following commands:

  ```
  drtl -definemodule drtl1 riscMon24.v -top riscMon
  ```

  ```
  drtl -addinst drtl1 monInst(.clk(clk), .rst(rst), .data(data[8:0]),
  .ld(register.ld_reg) )
  ```

  where,

  - `drtl1`: Module defined by the `drtl` command
  - `riscMon24.v`: File that contains the DRTL code
  - `monInst`: Instance created for the `drtl1` module

- To access an instance's output port in SDL programs, use this format: <*instance_name*>.output_port

  ```
  State Monitor
  {
  if ( monInst.LD == 1'b1 )
  {
  display("PORTS: PC: %h OP: %h ", monInst.PC[5:0],monInst.OP[2:0] );
  display("SDL : PC: %h OP: %h \n",data[8:3], data[2:0]);
  goto L0;
  ```


© 2021 All Rights Reserved.

```
}
}
```

- To save `drtl1` to a file for later use:

```
drtl -save -overwrite save.txt drtl1
```

- To load a saved DRTL module from a file and use the module in an xeDebug session:

```
drtl -load -overwrite save.txt drtl1
drtl -addinst drtl1 monInst(cntr_addrMem[5:0], opcode[2:0], data[8:0],
muxOUT_addrMem[5:0], data_en, register.ld_reg )
drtl -enablesystask monInst
```

Initiating DPI Calls from DRTL Modules

You can initiate DPI calls from the DRTL modules — that is, you can toggle signals in the DRTL modules that triggers a DPI call that is part of the DUT and testbench modules inside the emulator. To do this, you must use the GFIFO monitor template provided with the release. This template is discussed in the following sections.

The steps to initiate the DPI calls from the DRTL modules are as follows:

1. Define a GFIFO monitor, which is used to initiate the DPI calls. Cadence recommends you to use the GFIFO monitor template provided with the release, at `<INSTALL_PATH>/tools.lnx86/gift/tutorial/dpi`. For details, refer to Defining GFIFO Monitor Modules section.
2. Instantiate one or more GFIFO monitors in your design. One GFIFO monitor is required for each DRTL module. For details, refer to Instantiating GFIFO Monitor section.
3. Define the DRTL code to trigger the DPI call. For details, refer to Initiating DPI Calls section.
4. Define the C functions using the template provided. Refer to Defining C Functions section.

DPI calls can also be used with parallel GFIFO. The steps and the template to implement DPI calls with parallel GFIFO are the same as that of regular GFIFO.

Note: DPI within DRTL modules are not supported in Infinitrace sessions and with capture and replay operations.

Defining GFIFO Monitor Modules

Cadence recommends you to use the following Verilog template to define the interface for the GFIFO monitor. In this template, GFIFO monitor module is `DrtlMon`.

© 2021 All Rights Reserved.

```
module DrtlMon;
   parameter WD = 512;
   wire           en_internal, en;
   wire [WD-1:0]  data_internal, data;
   import "DPI-C" function void drtlFunc(bit[WD-1:0] x);
   always@(en_internal)
      drtlFunc(data_internal);
   IXCdrtlPort#(WD+1) p({en_internal, data_internal},
{en, data});
`ifdef AXIS
   initial $ixc_ctrl("gfifo", "drtlFunc", "tb_time");
`endif
endmodule
```

The GFIFO monitor template comprises the following sections:

- `en` and `data` signals: The `en` and `data` signals can be accessed from outside the `DrtlMon` module. These signals will be accessed from the DRTL code. When `en` gets toggled, the `DrtlMon` module calls the C function, `drtlFunc`. The DRTL code packs the parameter of the C function into the `data` signal — This usage is described in Initiating DPI Calls section.

  Note: The `en` and `data` signals are the only interfaces for `DrtlMon`. Other signals cannot be accessed outside `DrtlMon`.

- C function `drtlFunc`: The C function, `drtlFunc` has a single parameter, which is a pointer to `svBitVecVal`, `svBitVecVal`points to an array of 32 bit values. When DRTL module calls `drtlFunc`, all parameters are packed into an array of 32 bit values. For information about the function definition, refer to Defining C Functions section

- `IXCdrtlPort` ensures that DRTL drives the `en` and `data` signals properly.

- The `$ixc_ctrl` directive directs to the IXCOM compiler to use the GFIFO monitor to implement the DPI call.

Instantiating GFIFO Monitor

Instantiate the GFIFO monitor in your design's top module. You must estimate the required number of instances and the data bits needed for each instance. The default number of bits is 512. Modify the top level Verilog module to add the instances and recompile the design.

The following example instantiates two `DrtlMon` instances: `Dmon1` and `Dmon2`. `Dmon1` has 128 bits and `Dmon2` has the default number of bits, that is, 512.

```
module top;
    wire clk;
    IXCclkgen #(200) (clk);
    sub1 s1(clk);
    sub2 s2(clk);
    DrtlMon#(128) Dmon1();  // DRTL monitor using DPI-C
```


© 2021 All Rights Reserved.

```
    DrtlMon Dmon2(); // DRTL monitor using DPI-C
endmodule // top
```

Initiating DPI Calls

Use the following template to write the DRTL code to trigger the DPI call. In this template, the DRTL code assigns an ID for different functions, packed function parameters, and in `data` of `DrtlMon`, and toggles `en` of `DrtlMon` to call the DPI function.

```
module DRTL;
   wire clk = top.s2.clk;
   wire [32*1-1:0] c1  = top.s2.c1[32*1-1:0];
   wire [32*2-1:0] c2  = top.s2.c2[32*3-1:32];
   bit  mon_en;
   bit  [32*4-1:0] mon_data;
   wire [32*1-1:0] id = 1;
   assign top.Dmon1.en = mon_en;
   assign top.Dmon1.data[32*16-1:0] = mon_data;
   always@(posedge clk) begin
      if(c1[3:0] == 12) begin
      mon_en = ~mon_en;
      mon_data = {c2, c1, id};
      end else if (c1[3:0] == 4) begin
      mon_en = ~mon_en;
      mon_data = {c2, c1, id};
      end
   end
   bit  mon2_en;
   bit  [32*2-1:0] mon2_data;

   wire [32*1-1:0] id2 = 2;
   assign top.Dmon2.en = mon2_en;
   assign top.Dmon2.data[32*2-1:0] = mon2_data;
   always@(posedge clk) begin
      if(c1[3:0] == 6) begin
      mon2_en = ~mon2_en;
      mon2_data = {c1, id2};
      end
   end
endmodule
```

The above code calls two C functions. The GFIFO monitor module `DrtlMon` uses IDs to distinguish various function calls. If `id = 1`, `DrtlMon` calls the first C function. If `id = 2`, `DrtlMon` calls the second C function.

The first C function is triggered by toggling `top.Dmon1.en` as follows:

```
assign top.Dmon1.en = mon_en;
mon_en = ~mon_en;
```

`mon_data` has two parameters packed in `top.Dmon1.data` together with the id.

```
assign top.Dmon1.data[32*16-1:0] = mon_data;
mon_data = {c2, c1, id};
```


© 2021 All Rights Reserved.

Similarly, the second C function is triggered by toggling `top.Dmon2.en`

```
assign top.Dmon2.en = mon2_en;
mon2_en = ~mon2_en;
```

`mon2_data` has one parameter packed in `top.Dmon2.data` together with the id.

```
assign top.Dmon2.data[32*2-1:0] = mon2_data;
mon2_data = {c1, id2};
```

The C function, `drtlFunc`, has a `switch` statement to call different functions based on ID — See Defining C Functions section.

Defining C Functions

Following are the requirements to define the C functions that are called from within the DRTL modules.

- The name of the C function must match the function name used by the GFIFO monitor, `DrtlMon`.
- The argument of the function is a pointer to the array of 32 bit integers.
- The first integer in the array is the function ID.
- The number and the order for other 32 bit integers must match the corresponding data in the DRTL code.

The definitions provided in the template are as follows:

```
void F1(int n1, int n2)
{
      printf("F1: n1=%d, n2=%d\n", n1, n2);
}
void F2(int n1)
{
      printf("F2: n1=%d\n", n1);
}
void drtlFunc(svBitVecVal *d)
{
      int id = d[0];
      switch( id )
      {
      case 1: F1(d[1], d[2]); break;
      case 2: F2(d[1]); break;
      default: printf("Error: invalid function id.\n");
      }
}
```


© 2021 All Rights Reserved.

Executing DPI Embedded DRTL Modules

Open the design in xeDebug and enable the DPI, define the required DRTL modules, and create the DRTL instances.

To enable the DPI, invoke xeDebug with the `-sv_lib <gfifo_lib>` option. The `<gfifo_lib>` is the name of the compiled library containing your DPI C functions.

Example

1. Execute the `drtl -define` command to define DRTL modules, as follows:

   ```
   drtl -define MyDrtlModule mydrtl.v
   ```

2. Use the `drtl -addinst` command to create DRTL instances:

   ```
   drtl -addinst  MyDrtlModule inst1(.in1(Top.n1), .in2(Top.n2))
   ```

For details, refer to Defining DRTL Modules and Creating DRTL Instances section.

Analyzing Capacity of DRTL Modules

You can estimate the gate count for one DRTL instance for one or more DRTL source file. Use the following command at the unix prompt to get the gate count capacity for a DRTL instance:

```
drtlUtility -getcapacity [-verilog/-vhdl/-sv] <file_list> -topmodule
<top_module_name> [-compilescript <user_defined_compile_script>] [-parameter
{<parameter_name>=<value>}][-1Xmode]
```

For details on the options, refer to the description of this command in the *Programs and Utilities* chapter of the *VXE Command Reference Manual*.

When you add DRTL instances, xeDebug needs to generate more logic to implement the DRTL. The extra logic is added through a dynamic netlist, which uses free resources that are not occupied by the design logic. You can use the `compilerOption -add {DynamicNetlist <N>}` command in your compile script to reserve the required resources at run time.

Follow the given steps to use this utility:

1. Execute the `drtlUtility -getcapacity` command at the unix prompt. For example:

   ```
   drtlUtility -getcapacity -sv dtest.sv -topmodule dtest
   ```

   You get the gate count as the output of this command.

   Sample Output:

   ```
   512
   ```


© 2021 All Rights Reserved.

2. Open the compile script of your design which uses a DRTL module. In the `compilerOption -add` {`DynamicNetlist` <*N*>} command, replace `N` with the number from the output of the `drtlUtility -getcapacity` command.

The output number is the suggested reserved gate count for one DRTL instance. If you have multiple DRTL instances from the same DRTL module, multiply the netlist number to get the actual gate count.

© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

45

Debugging Designs using vvmDebug

This chapter gives an overview of the vvmDebug tool and explains various operations that can be performed using vvmDebug.

This chapter includes the following sections:

- Introduction
- Creating VVM Sessions
- Invoking the vvmDebug GUI
- Opening VVM Sessions
- SDL Trigger and Waveform Generation
- Creating Bookmarks
- Creating SDL Programs
- Creating Probes
- Controlling Memory Dump Operations
- Debugging RTL Files and Gate Level Netlists
- Defining Fast Probes
- Analyzing Power Consumption in VVM Sessions
- Run-Time Commands Support for VVM

© 2021 All Rights Reserved.

Introduction

Virtual Verification Machine enables you to debug your designs in the offline mode without occupying the emulator. VVM offers enhanced offline debugging features for Palladium Z1 generated databases. Before debugging the designs, you need to create VVM sessions online. The VVM sessions can be created either through the command line or using the xeDebug GUI. When a VVM session is being created online, the emulation is paused periodically to capture necessary data, which includes DCC and memory snapshots. Therefore, online emulation runs might take a longer time to finish.

After the VVM session has been created, launch the vvmDebug GUI and open your VVM session to begin debugging in the offline mode. You have the option to access the memories quickly during the offline session. vvmDebug enables you to perform the following operations:

- Generate waveforms
- Perform memory dump operations. The memories can be recreated at any time that was captured in the online session.
- Create bookmarks, SDL programs, and probes
- Debug RTL files and Gate Level Netlist
- Analyze power consumption of designs

Note: VVM is supported in STB mode in ICE and SA mode. It is, however, not supported in LA and VD mode in ICE and SA mode with uncontrolled clocks.

Creating VVM Sessions

The VVM sessions can be created either through the XEL command line or using the xeDebug GUI:

To create VVM sessions through the XEL command line, run the following command:

`vvm -prepare [-overwrite] <session_name> [-memoryReplay [<memories>]]`

For example, `vvm -prepare ghi -memoryReplay m0.m_mema m0.m_memb`

This command creates a new VVM session directory called `ghi.vvm`.

- The `vvm -prepare` command starts a new *.vvm session and assigns a name for this session. When the design is running, Palladium captures and uploads the DCC and memory data automatically into the VVM session. The session name is used to derive the name of the session directory that contains data for the VVM session.

© 2021 All Rights Reserved.

- The `-overwrite` option overwrites any existing VVM session. If the `-overwrite` option is not specified and a session with the specified name already exists, an error message is issued.
- The `-memoryReplay` option enables memory replay in the offline session, which allows you to recreate memory contents at any point in simulated time in the offline session. If this option is specified, all the memory instances in the design can be replayed. By default, the VVM sessions are created without `-memoryReplay`.
- You can specify a list of memory names after the `-memoryReplay` option. Naming specific memory instances reduces the time taken to recreate them in the offline session.

To create VVM sessions through the xeDebug GUI:

1. Launch the xeDebug GUI and download the design.
2. To create a VVM session, go to the *Control* tab and select the *VVM Prepare* radio button.

Figure 45-1 xeDebug Control Tab

© 2021 All Rights Reserved.

3. In *Session Name*, specify a name for the VVM session and click the button. If a session already exists with the same name, you will be prompted to overwrite the existing session.

Figure 45-2 Memory Replay Option

4. Select the *memoryReplay* option to enable memory replay in the offline session.

   Note: The memory replay increases the prepare time and disk space required for the session directory.

Figure 45-3 Fast Memory Replay List

5. The *Memory Instance* list displays all the memory instances in the current design. Select the required memory instance for fast memory replay and click the **>** icon to move the selected memory instance to the *Fast Replay* list. Select more instances, if required.

6. Click *Confirm* to confirm the memory instances and close the window.

7. Issue the `run` command and specify the number of cycles to run, as follows. The VVM data will be automatically captured during the run.

   ```
   run <number of cycles>
   ```

   For example, `run 10000000`

   The emulation is paused periodically to upload the VVM data. The *System Status* group box shows the progress of uploading the data in (percentage). The current simulator time and SDL status is also displayed.

© 2021 All Rights Reserved.

Figure 45-4 Run Status

Important

To open another VVM session, you must first close the current VVM session.

To close a VVM session, issue the following command:

```
vvm -close
```

If this command is used in an online session, it closes the current VVM session and stops capturing. If used in an offline session, it closes the current VVM session that is being debugged.

Example: Creating VVM Sessions in SA mode

```
debug .
host .
:
database -open -shm uatrace
xc xt0 zt0
```


© 2021 All Rights Reserved.

```
xc on -autorun
run -swap
#Specifying fast probe signals
probe -create -shm -fast SM.cen SM.clk SM.inc SM.local_counterSM.offset
#Adding SDL signals to the fast probes
sdl -addProbes -fast trigger.tdf
#Creating a VVM session
vvm -prepare first -memoryReplay tbench.mem1.mem
run
#Closing the VVM session
vvm -close
```

Invoking the vvmDebug GUI

After the VVM sessions are created, launch the vvmDebug GUI for debugging in offline mode. Execute the following command from the UNIX command line to invoke the vvmDebug GUI:

```
vvmDebug -gui
```

Note: For information about the `vvmDebug` command options, refer to the `vvmDebug` command description in the *Programs and Utilities* chapter of the *VXE Command Reference Manual*.

© 2021 All Rights Reserved.

Figure 45-5 vvmDebug GUI

The vvmDebug GUI looks similar to the xeDebug GUI. On the bottom of vvmDebug GUI, there is a console window where you enter the XEL commands like you do in the xeDebug GUI. The command prompt for vvmDebug is `XEvvm`. Currently, the vvmDebug GUI has the following tabs:

- *Setup*: Used to open VVM sessions and perform debugging. For details, see Opening VVM Sessions.
- *Control*: Used for SDL trigger and waveform generation. For details, see SDL Trigger and Waveform Generation.
- *Bookmark*: Used to mark specific time in the design database. For information on how to create bookmarks, see Creating Bookmarks.
- *SDL*: Used to create and edit SDL (State Description Language) programs. For details, see Creating SDL Programs.

© 2021 All Rights Reserved.

- *Probe*: For details, see Creating Probes.
- *Memory*: Used to control the memory dump operations. For details, see Controlling Memory Dump Operations.
- *RTL/Gate*: Used to control the RTL files and gate-level netlist debug functions. For details, see Debugging RTL Files and Gate Level Netlists.

Important

For information about the supported commands for VVM, see Palladium VVM Commands Support Matrix. For information on the toolbar buttons, see the Toolbar section in Appendix 42, “Using the xeDebug GUI”.

Opening VVM Sessions

To open a VVM session from the XEL command line in vvmDebug, use the following command:

```
vvm -open <session_name>
```

For example, to open a VVM session named *abc*, use the following command:

```
vvm -open abc
```

In the vvmDebug GUI, the *Setup* tab shows all the available VVM sessions in your design directory. You can open a VVM session and then debug it.

© 2021 All Rights Reserved.

Figure 45-6 Setup Tab: Initial Screen

Database and Work Directory group box: This group box is the same as the xeDebug GUI. You can select the design and the working directory.

- *Design Directory*: This field displays the location of the compiled design database. The path to the compiled design database can be changed manually or by using the directory tree displayed. However, to bring the change to effect, click the ✔ button adjacent to the field.
- *Work Directory*: This field displays the location of directory from which the vvmDebug GUI was invoked. This field is linked to the design directory.
- *User Session*: This drop-down list allows you to enter a new session name, or select an existing session subdirectory.
- Click *Open Design* to open the selected design for debugging.

© 2021 All Rights Reserved.

Figure 45-7 Setup Tab

The *VVM Session group box* enables you to select a VVM session and open it for debugging.

- Specify the VVM session name or you can bring up the file selection dialog by clicking the icon.
- By default, the *Local Session* option is selected, which lists all VVM sessions under the design directory. To view a list all VVM sessions under the design directory and subdirectory, select *All Sessions*.

  For example,

  The design path, `/xx/design` has two VVM sessions, `/xx/design/11.vvm` and `/xx/design/swap/22.vvm`. Selecting *Local Session* will list only `11.vvm`, however, selecting *All Sessions* will list both `11.vvm` and `22.vvm`.
- Select the required VVM session and click *Open Session* to open the VVM session.

© 2021 All Rights Reserved.

If you are running the design for the ICE mode and need connectivity information, then specify the *cds.lib* file and snapshot:

- Specify the library definition by adding the location of a *cds.lib* field. By default, the file created during the compile is located in *INCA_libs/worklib* directory.
- In the *Snapshot* field, specify the design snapshot. The snapshot is used by simvision to access the complete design hierarchy.

To view detailed information about each VVM session in the current session directory, issue the following command:

```
vvm -list -info
```

To view all the intervals for the current VVM session, issue the following command:

```
vvm -intervals
```

Note: For more information on the `vvm` command and options, refer to the `vvm` command description in the *Run-time Commands* chapter of the *VXE Command Reference Manual*.

© 2021 All Rights Reserved.

SDL Trigger and Waveform Generation

Once the VVM session has been opened, the *Control* tab as shown in the following figure is displayed. This tab enables you to perform the following operations:

- Select the SDL trigger modes for SDL instances
- Run the designs.
- Manage captured trace data

If an SDL Program meets a trigger condition on the first cycle, SDL will respect it on the first run. However, on the next run, SDL will not trigger because events on this cycle are already served and all SDL events, including `TRIGGER`, `EXEC`, `DISPLAY`, and `TX` statements, that were served on the first run will be ignored on this cycle.

In IXCOM, if run stopped at a time unit, for example 10ns, on fifth FCLK cycle, then the next run will start from the same time unit and FCLK cycle.

SDL running under VVM and SDL running in the emulator might exhibit some differences in timing, depending on the settings of the xeset `stopDelay` command and the exact sequence being executed by SDL. The difference between the two can be anywhere between `0` and `2` cycles. For more details on the timing model of SDL, refer to Chapter 43, “Controlling the Run with SDL,” Also, refer to the description of the `xeset stopDelay` command in the *VXE Command Reference Manual*.

© 2021 All Rights Reserved.

Figure 45-8 Control Tab

Selecting Trigger Modes

The *SDL and Break Control* Group box allows you to select trigger modes for SDL instances. If no trigger mode is defined, the design runs without any trigger conditions set.

1. Select either *Enable* or *Disable* radio button to control the SDL Programs that have not been specifically enabled or disabled using the *Advanced SDL Control* window accessible from the *SDL* tab. You can also define, load, or enable the SDL programs using the *SDL* tab.

2. In the *Trigger Expression* field, specify the event or logic expression for triggering by expression. The syntax and semantics of the expressions are the same as SDL expressions.

© 2021 All Rights Reserved.

3. In the *Post Trigger Samples* field, specify a percentage to calculate the number of samples to capture after a trigger occurs. By default, it is set to 100%.

Figure 45-9 SDL & Break Control

Running Designs

Use the *Run Control* Group box to virtually run the design. In vvmDebug, running a design means playing back the design.

Figure 45-10 Run Control

- To run the design continuously without resetting the SDL state from the previous run, select the *Continue Run* check box.
- To go to a specific time point in the current VVM session, enter the time in the text box. The time unit is displayed adjacent to the text field.

  Or

- Select an existing bookmark from the list. A bookmark is a mark for a specific time point. For more information, see Creating Bookmarks.

  Alternatively, issue the following command:

  ```
  restart {<cycles>|<bookmark_name>}
  ```

  For example:

  To start running the design after 100 ns, run the following command:

  ```
  restart 100 ns
  ```

  To go to bookmark1, run the following command:

© 2021 All Rights Reserved.

```
restart bookmark1
```

The *Goto* drop-down list shows the bookmarks available in the *Bookmark* tab. If there is a filter in *Bookmark* tab, the list only shows the bookmarks which match the filter pattern.

Note: Selecting *Continue Run* starts data tracing after which you can upload the traced data. However, *Goto* or `restart` are used only to go to a specific time point and do not start data tracing.

Managing Captured Trace Data

The *Trace Data Capture* group box provides fields to select how captured trace data is to be dealt with.

Figure 45-11 Trace Data Capture

1. Specify the name of the trace file in the *Wave DB Name* field.
2. Select the *Probe SDL Nets* check box if the SDL signals need to be probed.
3. Turn on the *Auto Upload* option to automatically trace data, upload it and display the waveform after each run.
4. Select the *Auto Wave* check box to automatically display data in the Waveform Viewer.
5. *Set Trace Window*: You can either specify the memory size or set the start and end time for the trace window:

© 2021 All Rights Reserved.

- *Start/End Time*: Specify the start and end time or any existing bookmark name.
- *Trace Memory Size*: You can set the memory size in terms of percentage or specific unit such as fclk. The upload is done for the specified memory size ending at the time point or bookmark specified in *Run Control*.

6. Select the signals to be probed. See Selecting Signals to be probed.
7. Click *Upload* to manually upload the trace data to the Waveform Viewer.

Note: To delete the trace data from the current trace file, click *Clear Trace*.

The *Session Status* group box provides information about the current VVM session such as the session name, intervals, current time, current cycle, and so on.

Selecting Signals to be probed

1. Open the *DUV Design Browser* window to select the signals to be probed.
2. Select the required signals.
3. Select *AutoUpload* and then click *Probe*.

You can then view the waveforms in SimVision for the requested signals and time period.

Creating Bookmarks

Bookmarks are used to mark specific time points. You can use bookmarks as parameters to other commands. For example, you create two bookmarks bm1 for 250 fclk and bm2 for 500 fclk. You can use these bookmarks to upload waveform from 250 fclk to 500 fclk, as follows:

```
database -upload -start bm1 -end bm2
```

Bookmarks can be created either through the command line or using the *Bookmark* tab in xeDebug or vvmDebug.

To create bookmarks through the command line, issue the following command:

```
bm -add <bookmark_name> [<time_specification>] [-replace]
```

Note: For more information about the `bm` command and its options, refer to the `bm` command description in the *Run-time Commands* chapter of the *VXE Command Reference Manual.*

To create bookmarks through the *Bookmark* tab:

1. Go to the *Bookmark* tab.
2. Specify a name for the bookmark in the *Bookmark* field.

© 2021 All Rights Reserved.

3. Specify time for the bookmark in the *Time* field. If no time unit is entered, the current time unit is used as default.
4. Click to add a new bookmark. If the bookmark name already exists, click *Replace* to replace it.

Figure 45-12 Bookmark Tab

To filter bookmarks on the basis of a specific pattern:

1. Enter a pattern in the *Filter* field. A "*" lists all the bookmarks.
2. Click to apply the filter.

To save bookmarks in a file:

1. Enter the bookmark file name in the *Bookmark File* field.
2. Click *Save* to save all bookmarks into the specified file.

To load bookmarks from a file:

1. Click to open the file selection dialog and select the required file.
2. Click *Load* to load the bookmarks from the specified file.

The bookmarks are saved automatically in the design directory. If you try to add a bookmark that already exists, an error message will be displayed. To overwrite existing bookmarks, select the *Replace* checkbox.

© 2021 All Rights Reserved.

Creating SDL Programs

The *SDL* tab contains the *SDL Editor* group box that is used for creating and editing SDL (State Description Language) programs. The SDL programs are saved to and restored from files by using the *Save* and *Load* buttons. The *SDL* tab in the vvmDebug GUI is same as in the xeDebug GUI. For more information, see the SDL Tab section in Appendix 42, "Using the xeDebug GUI".

VVM SDL Support Matrix

The following table lists the supported and unsupported options for the `sdl` run time command. For information on how to create SDL programs, see Chapter 43, "Controlling the Run with SDL". For a description of the `sdl` command and its options, refer to the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Table 45-1 VVM SDL Support Matrix

Option	Supported / Not Supported
`-addProbes`	Supported
`-autoProbe`	Supported
`-disable`	Supported
	Note: Disabling is always instant and does not require the `-now` option.
`-display`	Not Supported
`-enable`	Supported
	Note: Enabling is always instant and does not require the `-now` option.
`-expression`	Supported
`-execOvf`	Not Supported
`-getActiveInstances`	Supported
`-getAssertions`	Not Supported
`-getFiles`	Supported
`-getInstances`	Supported
`-getLabels`	Not Supported

© 2021 All Rights Reserved.

Table 45-1 VVM SDL Support Matrix

Option	Supported / Not Supported
`-getNets`	Supported
`-getStatus`	Supported
`-getTrigger`	Supported
`-haltOnTrigger`	Not Supported
`-isActive`	Supported
`-isExecOvf`	Supported; always returns 0, which indicates no execution overflow.
`-isDispOvf`	Supported; always returns 0, which indicates no display buffer overflow.
`-isTrigger`	Partially Supported: Not Supported Options: ■ `-instance` ■ `-label`
`-ldcnt1/-ldcnt2`	Not Supported
`-load`	Supported
`-log`	Not Supported
`-simBreakpoints`	Not Supported
`-onlyReport`	Not Supported
`-pp`	Supported
`-report`	Supported
`-restoreState`	Not Supported
`-saveState`	Not Supported
`-scope`	Supported
`-setFile`	Supported
`-simTime`	Supported
`-stop`	Not Supported
`-traceClear`	Not Supported

© 2021 All Rights Reserved.

Table 45-1 VVM SDL Support Matrix

Option	Supported / Not Supported
`-traceDump`	Not Supported
`-traceOn`	Not Supported
`-tx`	Not Supported
`-verify`	Supported
`-when`	Supported

Creating Probes

The term probe refers to the collection of probed objects grouped under a single probe name. The *Probe* tab in the vvmDebug GUI is also same as in the xeDebug GUI. You can create a probe on either a specific signal (or assertion), a specific scope (that is, all objects under the specified design hierarchy). The *Upload* and *Display* buttons are used to upload the data into the waveform database and display the probes in the specified waveform browser. If required, the probe lists can be saved and imported using the *Save* and *Import* buttons. For more information, see the Probe Tab section in Appendix 42, “Using the xeDebug GUI”.

Note: Probes can also be defined using the *DUV Design Browser* window. This window is also same as in xeDebug. For information on how to use the DUV Design Browser, see the DUV Design Browser section in Appendix 42, “Using the xeDebug GUI”.

© 2021 All Rights Reserved.

Figure 45-13 Probe Tab

Controlling Memory Dump Operations

If the memory replay is enabled in the online mode while creating VVM sessions, you can recreate the memory contents at any time point and dump the memory in the offline session.

You must first define the time point where the memory contents need to be dumped. For example, use `restart` or `run` to get to a specific time point and then dump the memory. The *Memory* tab in the vvmDebug GUI is used to control the memory dump operations. You can specify a file format for the output file in which the data should be dumped.

© 2021 All Rights Reserved.

Important

The memory replay is enabled by specifying the *-memoryReplay* option in the VVM prepare session. By default, all the memories are probed after this option is enabled. You can specify memory instances for fast replay. This is a benefit when the memory contents will be recreated later in the offline session. For more information, see the points 4 and 5 in section Creating VVM Sessions.

Figure 45-14 Memory Tab

To dump memories:

1. Go to the *Memory* tab.
2. In the *Memory Location* field, select whether the edit operation should apply to all memories, tb (testbench) memories only, or dut memories only.

© 2021 All Rights Reserved.

3. Specify the memory instance to be operated upon in the *Memory Instance* field.
4. Specify a memory content file for a specific memory instance in the *File* field.
5. Specify a memory format. This field is a listbox. Click the down-arrow on the right of the field to view and select an available format. The available formats are:
   - `%h`: Xcelium format, radix hex
   - `%b`: Xcelium format, radix bins
   - `%o`: Xcelium format, radix oct
   - `%readmemb`: readmemb format
   - `%readmemh`: readmemh format
   - `%pd_b`: Palladium ASCII format, radix bin
   - `%pd_o`: Palladium ASCII format, radix oct
   - `%pd_d`: Palladium ASCII format, radix dec
   - `%pd_h`: Palladium ASCII format, radix hex
   - `%pd_memtran`: memtran format
   - `%pd_raw`: raw format
   - `%pd_raw2`: raw2 format
6. Specify the range of memories to be read in the *Start Address* and *To* fields. Enter hexadecimal values. If these fields are left blank, the entire memory is read.
7. Select the *Dump* option from the *Memory Action* drop-down list for the selected memories. Use the button to commit the memory contents file to the selected memory instance.
8. Click *Save* to bring up the dialog box that enables you to save the memory list to a file.
9. Click *Import* to bring up the dialog box that enables you to load memory data from a previously-saved file.
10. Click *Clear* or *Clear All* to clear either the selected memory or the entire list of memories, respectively.

Note: For more information about the `memory -dump` command and options, refer to the `memory` command description in the *Run-time Commands* chapter of the *VXE Command Reference Manual*.

© 2021 All Rights Reserved.

To dump multiple memories at the same cycle and to avoid multiple memory replay for the same cycles, add the `-bg` option to each `memory -dump` command. The memory dump is not done until the `memory -fg` command is issued. This method is faster than replaying the memories separately.

For example,

```
memory -dump -bg %pd_h M1 -file M1.dump
memory -dump -bg %pd_h M2 -file M2.dump
memory -fg
```

If, however, the `memory -dump -bg` command is followed by the `memory -dump` command without the `-fg` option, the system automatically triggers the `memory -fg` command to complete the memory dump.

In the following example, the `memory -fg` command will be triggered after the `memory -dump` command.

```
memory -dump -bg %pd_h M1 -file M1.dump
memory -dump -bg %pd_h M2 -file M2.dump
memory -dump %pd_h M3 -file M3.dump
```

Important

The memory contents should be recreated only when the memory is stable. For example, the memory contents should not be recreated at a write-enable edge. If the memory is not stable, the results from vvmDebug might not match the results from online execution. This behavior is applicable to both SA and ICE modes.

Debugging RTL Files and Gate Level Netlists

The *RTL/Gate* tab allows you to debug Verilog and VHDL-based RTL files and gate-level netlists. It can be used in conjunction with the RTL Design Viewer for working with RTL files, or the Schematic Viewer for working with gate-level netlists.

This tab controls the import of gate-level netlists, or RTL files. Imported files can then be displayed in the Schematic Viewer or RTL Viewer.

Note: For more information, see the RTL/Gate Tab section in Appendix 42, “Using the xeDebug GUI”.

© 2021 All Rights Reserved.

Figure 45-15 RTL/Gate Tab

Defining Fast Probes

A VVM session contains essential fullvision data. You can use the `fvCompute` command to do preprocessing on the fullvision data for fast interactive waveform generation.

Note: For information about the `fvCompute` command and the steps required for waveform generation, refer to the `fvCompute` command description in the *Programs and Utilities* chapter of the *VXE Command Reference Manual*.

Preprocessing might take time if the session captures a very long time window. If preprocessing has not been performed, the first time waveform generation requires the fullvision computation and is slow. When running SDL programs or searching for SDL trigger, the SDL simulator needs the waveform for the involved signals. You can define these signals as fast probes to bypass the fullvision computation. The fast probes have to be defined in the

© 2021 All Rights Reserved.

online session before the VVM session is captured. The waveform for fast probes is directly captured in the VVM sessions.

For example, 100 million cycles are captured in the VVM session and need long time for preprocessing. You can define some critical signals as fast probes. Then, in the offline session, you can search for the trigger point with these fast probes. After a trigger point has been identified, you can dump waveform for 1 million cycles around this trigger point. So, you would need to do fullvision on a small time window which is only 1% of the whole session. The fast probes should be added before executing the `vvm -prepare` command.

To define fast probes, use the `-fast` option with the `probe` commands.

For example,

```
probe -create -fast
probe -addinst -fast
```

To view a list of fast probes, run the following command:

```
probe -list -fast
```

To save fast probes, run the following command:

```
probe -save -fast
```

To automatically add SDL related signals as fast probes, run the following command:

```
sdl -addprobes -fast <sdl-filename>
```

Note: The fast probes can be also be used in regular runs. Also, there is a limit to the number of fast probe signals that can be defined like there is a limit to the number of DYNP probes that can be defined when using the DYNP mode. The limit is based on the compile parameters (for example, number of domains) used.

Analyzing Power Consumption in VVM Sessions

You can perform DPA operations in VVM sessions in the offline mode. You must have the special license to perform DPA operations. The *Dynamic Power Analysis* (the power tower icon on the toolbar) invokes the Dynamic Power Analysis (DPA) menus.

In the xeDebug tool, the default trace window for performing DPA operations is the time range of the `.phy` file. However, in the vvmDebug tool, you must specify an explicit trace window before you start DPA operations. The time range of the VVM sessions is generally large. Therefore, to avoid performing DPA operations on the complete VVM sessions, the default trace window is not defined.

1. In the vvmDebug GUI, click the power tower icon on the toolbar to open the DPA window. The DPA window in the vvmDebug GUI is same as in the xeDebug GUI.

© 2021 All Rights Reserved.

2. In the *DPA Probe* tab, specify the probes to be used for DPA.
3. In the *TCA Preparation*, *TCF*, *SAIF*, and the *PowerProfile* tabs, specify the time range for performing DPA in the *Start Time* and *End Time* textboxes. This is equivalent to the following command:

   ```
   dpa - tracewindow -start <xxx> -end <yyy>
   ```

   Note: You can also use pre-defined bookmarks instead of specifying the time range.

   For example, `dpa -tracewindow -start bm1 and -end bm2`

For more information on DPA, refer to Chapter 32, “Analyzing Power Consumption of the Designs”

Run-Time Commands Support for VVM

This section provides information about the support of XEL run-time commands for VVM. For syntactical details of the listed commands, refer to the *Run-Time Commands* chapter of the *VXE Command Reference Manual*.

Table 45-2 Palladium VVM Commands Support Matrix

Command	Supported / Not Supported
assertion	Not Supported
bm	Supported
clockConfig	Not Supported
configPM	Not Supported
convertRadix	Supported

© 2021 All Rights Reserved.

Table 45-2 Palladium VVM Commands Support Matrix

Command	Supported / Not Supported
database	Partially Supported Not Supported Options: ■ -listphyfile ■ -prepareOffline ■ -captureMode ■ -continuousupload ■ -phy ■ -disable ■ -enable
debug	Supported
deposit	Not Supported
download	Not Supported
dpa	Partially Supported Not Supported Options: ■ -phyfile ■ -listphyfile ■ -sparse_sampling
force	Not Supported
genPowerProfile	Supported
getCableStatus	Not Supported
getCycleNum	Supported
getHostStatus	Not Supported
getPMStatus	Not Supported
getTimes	Supported
hdlConfig	Not Supported
hdlRunLogFile	Not Supported

© 2021 All Rights Reserved.

Table 45-2 Palladium VVM Commands Support Matrix

Command	Supported / Not Supported
host	Not Supported
infiniTrace	Not Supported
xrun	Not Supported
lp	Not Supported
memory	Partially Supported Not Supported Options: ■ -load ■ -set ■ -reset ■ -setvalue ■ -getFileOffset ■ -sparse_mem_info **Note:** The -foreground option is supported only in the ICE mode.
xmsim	Not Supported
probe	Partially Supported Not Supported: -addPowerDomain
probe (for CPF)	Not Supported
probe (for 1801)	Not Supported
release	Not Supported
reset	Not Supported
resetCycleNum	Not Supported
restart	Supported **Note:** In vvmDebug, the restart command is used to go to a specified time point.
resultWaveform	Not Supported

© 2021 All Rights Reserved.

Table 45-2 Palladium VVM Commands Support Matrix

Command	Supported / Not Supported
`resume`	Not Supported
`run` (in ICE mode)	Partially Supported Not Supported: ■ `-wait` ■ `-nowait`
`run` (in SA mode)	Partially Supported Not Supported: ■ `-wait` ■ `-nowait` ■ `-ignorestop` ■ `-swap`
`save`	Partially Supported Not Supported: ■ `-commands` ■ `-overwrite` ■ `-simulation` ■ `-uservars` **Note:** In VVM, only the `-environment` option is supported to save the last debug environment.
`scope`	Supported
`sdl`	Partially Supported **Note:** For a complete list of supported and unsupported options, see VVM SDL Support Matrix.

© 2021 All Rights Reserved.

Table 45-2 Palladium VVM Commands Support Matrix

Command	Supported / Not Supported
stop	Partially Supported Not Supported: `-emulation` **Note:** As vvmDebug is used for offline debugging without connection to the emulator, the `stop` command in vvmDebug only stops the current run and does not stop the design clocks.
suspend	Not Supported
symbol	Supported
tca	Supported
userData	Supported
value	Supported **Note:** In vvmDebug, the `value` command returns the current state of the specified objects only after the `run` command finishes execution.
vector	Not Supported
vvm	Supported
waitWhileBusy	Supported **Note:** In vvmDebug, this command is used to wait until the trigger happens.
xeset	Partially Supported Not Supported: `-appendTrace`
xegui	Supported

© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

46

Embedded Software Debugging with Palladium Z1

This chapter provides information about support of the Embedded Software Debugging (ESWD) with Palladium Z1.

This chapter includes the following sections:

- Introduction
- Advantages of Embedded Software Debugging
- Hardware and Software Co-Verification with ESWD
- Offline Debugging Preparation
- Generating ESWD Database
- Offline Debugging With Indago Debug Analyzer

Introduction

Embedded Software Debugger (ESWD) is an embedded software debug solution targeted at debugging embedded software executing on RTL processor models in RTL platforms, running either using the simulation or emulation platforms. It enables you to trace the execution of the embedded software, synchronized with hardware traces, such that you can examine the state of the embedded software — be it variable values, stack, registers — at any given point in time.

Currently, ESWD supports only offline debugging. In VXE, the emulator software tools, xeDebug and vvmDebug facilitate offline debugging with ESWD and emulator data.

The ESWD database is generated from ARM tarmac logs that enable you to debug the source code in *Indago Debug Analyzer* (IDA). This debug view includes variables and functions to enable you to step through the code. You can correlate the source view with time accuracy, with hardware data available from the emulator.

© 2021 All Rights Reserved.

For information on how the ESWD databases are generated from the ARM tarmac logs and for more details on how to configure and use ESWD, refer to *Embedded Software Debugger User Guide* in the *Xcelium Documentation Set*.

The `TARMAC` logs generated prior to modification match the logs generated after running in the emulator. ARM `TARMAC` trace file(s) are produced directly by the ARM processors during a simulation or emulation session, one for each processor core for which tracing is enabled.The trace files contain the processor and embedded software trace information, with an associated time (or clock sequence).

ESWD requires IXCOM compilation and the front-end debugger for its first release is IDA 15.1. IDA does not support Verdi. However, the emulator might invoke xeDebug or vvmDebug with Verdi. Interactive debugging mode is not supported with ESWD.

Advantages of Embedded Software Debugging

ESWD enables you to debug the software-level functionality with the ability to get signal resolution from the emulation. In addition, you can get hardware-level analysis in a software perspective, for example, power consumption based on signal toggling in the context of software functions.

Hardware and Software Co-Verification with ESWD

The following diagram depicts the hardware and software co-verification with ESWD.

- The `TARMAC` code must be first synthesized into the emulator.
- The simulation is run and the `TARMAC` logs are generated.
- After simulation, in a post-processing step, the `TARMAC` logs, and the compiled image of the software are converted by ESWD into an ESWD database, `esw.db`. You also have an option to get an IDA and waveform database generated through ESWD.

  The waveform database contains software information such as functions, variables, and parameters. During offline debug, you can view and step through the software. In addition, you might invoke xeDebug or vvmDebug to generate more hardware data at specific time points relative to the software.

© 2021 All Rights Reserved.

Figure 46-1 Hardware and Software Co-verification with ESWD

Offline Debugging Preparation

In offline debugging, you do not occupy the emulator. However, you must generate the `TARMAC` logs and create an ESWD database. To test any changes in the software, you must re-run.

Offline debugging when used with IDA means that the Xcelium simulator, *xmsim*, is not connected. However, the simulation or emulation run is not changed. You can still have the emulator connected and downloaded, so that you can get additional data. In order to maintain the validity of the `TARMAC` logs, if the design is downloaded on the emulator, it must be running with InfiniTrace observe. Alternatively, you can run with xeDebug in the offline mode, which means that the design is not downloaded on the emulator.

© 2021 All Rights Reserved.

You must first run online with `TARMAC` file generation enabled. During the online run, you must also run with any necessary xeDebug commands to prepare the hardware data for the offline session.

For more information, refer to section, Debugging in Offline Session.

You can debug offline in several ways:

- Using vvmDebug
- Using xeDebug in Offline Mode
- Using InfiniTrace Sessions

Using vvmDebug

If you intend to use vvmDebug, you must open a VVM prepare session online. The VVM prepare session automatically logs all hardware traces. You can also enable memory replay in the offline session, which enables you to recreate memory contents at any point in simulated time in the offline session. To bypass FullVision computation, define the signals as fast probes in the online session. vvmDebug enables you to use SDL to search for the hardware conditions of interest before uploading data. It also allows for full memory replay, if you specify the `-memoryReplay` option during the prepare session.

Note: vvmDebug requires a separate license and at least Z1hardware. It is also slower during the prepare session than InfiniTrace because it has to upload the DCC data when DCC is full.

For information on how to create a VVM Prepare session, refer to section, Creating VVM Sessions. For information on how to define fast probes, refer to section, Defining Fast Probes.

Using xeDebug in Offline Mode

If you intend to use xeDebug in the offline mode, you must create a `.phy` file. You also have the option to create fast probes. For fastest performance on the emulator, use the `database -prepareOffline` command and defer the FullVision computation and waveform generation until running the offline session. Offline debugging in xeDebug can be done without a separate license and on all Z1 hardware. However, it does not support memory replay and SDL operations.

© 2021 All Rights Reserved.

Using InfiniTrace Sessions

To use InfiniTrace, you must open an InfiniTrace session. An InfiniTrace session is faster during the online session than opening a VVM prepare session, but requires the emulator during the InfiniTrace observe session. InfiniTrace prepare runs faster than vvmDebug prepare. It also does not require a special license and runs on all Z1 hardware. To generate additional hardware data, the InfiniTrace observe session must occupy the emulator. The time to upload DCC is additional time cost on top of the InfiniTrace prepare session.

You should not upload DCC to maximize performance. Open the prepare session to get all of the other data (besides `TARMAC` probes) during the observe session.

For more information on the InfiniTrace sessions, refer to section InfiniTrace Sessions.

Generating ESWD Database

The tarmac files are generated at end of the online session.

The ESWD database includes the Indago Debug Analyzer (IDA) database and a Waveform Trace database. After finishing the prepare session, an IDA database is generated which logs the previously identified functions or variables.

For more information, refer to *Embedded Software Debugger User Guide* in the *Xcelium Documentation Set*.

Offline Debugging With Indago Debug Analyzer

Use the Indago Debug Analyzer to bring up the IDA database and trace the source code. The Debug Analyzer also brings up SimVision and displays the waveform database.

The Debug Analyzer is a post-process, graphical debugger. That is, after you have identified a failing test, you run the simulation one more time to gather data about both the DUT and the testbench. Once the simulation data is stored in a database, you can load the data into the Debug Analyzer for analysis. Debug Analyzer works with the Xcelium simulator, *xmsim*, and SimVision to run the simulation and gather debugging data.

For more information on debugging with Indago Debug Analyzer, refer to *Indago Debug Analyzer App User Guide.*

Note: The Debug Analyzer is based on the Xcelium simulator and Specman. So, the environment must be configured for running these tools. For information on setting up the environment, refer to the *Installation and Configuration Information* and the *Specman*

© 2021 All Rights Reserved.

Configuration Guide.The following figure shows the Debug Analyzer window and SimiVision waveform.

Figure 46-2 Indago Debug Analyzer

The Debug Analyzer shows the user software, disassembly code around the current debug location, current software function of each core, variables, and breakpoints. In the waveform, you can see the functions with a `enter` or `exit` value to indicate if the core is in that function at that time and variable values.

© 2021 All Rights Reserved.

The Palladium plug-in buttons are available in the SimVision waveform, which enable you to work with the xeDebug and vvmDebug GUI. For example, using the Design Browser in the vvmDebug GUI, you might select the desired signals and add them to the waveform.

For more information on how to debug designs using vvmDebug, refer to chapter, Chapter 45, "Debugging Designs using vvmDebug". For information about using the xeDebug GUI, refer to chapter, Chapter 42, "Using the xeDebug GUI".

During the offline debugging session, do as follows:

1. Invoke the Debug Analyzer. You can invoke the Debug Analyzer from a script or from the UNIX command prompt using the following command:

   ```
   indago [-help] [-db=db-name] [-restore_file save-file] [-sim_env]
   ```

   If you do not specify a database name, the Debug Analyzer looks in the current working directory for a database with the default name, `ida.db`. The Debug Analyzer automatically opens the SimVision waveform viewer. The SimVision waveform viewer displays the Palladium toolbar. The other SimVision viewers that display the Palladium toolbar are the Design Browser, Source Browser, and the Schematic Browser.

   For more information on the `indago` command and options, refer to the *Indago Debug Analyzer App User Guide.*

   The Palladium Z1 toolbar includes the following buttons:

   - PZ1 icon – Opens the vvmDebug GUI.
   - *Probe* and *Display* buttons – Probes signals selected in the SimVision Design Browser and displays them in the waveform. To do this, you should have a snapshot loaded.
   - *Upload* button – Sends `database -upload` command to vvmDebug to generate emulator data to display on the waveform.
   - *Toggle H/W Data Folding* button – Collapses or expands the gaps between hardware data. This is useful for hiding long stretches of time that have no hardware data.
   - *Toggle Hide Folds* button – Shows collapsed data folds without or with orange bars.
   - *Dynamic Power Analysis* button (the power tower icon) – Brings up the DPA GUI and generates a DPA waveform. For more information, refer to section, Analyzing Power Consumption Using xeDebug GUI.

2. From the UNIX command prompt, invoke the xeDebug or vvmDebug GUI.

   To invoke the xeDebug GUI, issue the following command:

© 2021 All Rights Reserved.

```
xeDebug -gui [-shm|-fsdb|-vcd] \
             [-init <tcl_file>] ... \
             [-log <name>] [-append_log] [-append_key] \
             [--xmsim <option> ... [--]] [script [<arg> ...]]
```

To invoke the vvmDebug GUI, issue the following command:

```
vvmDebug -gui
```

For information about the `xeDebug` and `vvmDebug` command options, refer to the *Programs and Utilities* chapter of the *VXE Command Reference Manual*.

3. Click the PZ1 icon to connect to the xeDebug or vvmDebug GUI. If the xeDebug or vvmDebug GUI starts in the same design directory before the Debug Analyzer, the Debug Analyzer connects to it automatically.

4. If you are using the vvmDebug GUI, select the previously prepared VVM session. If you are using xeDebug in the offline mode, open the `.phy` file. If you are using InfiniTrace, download the design into the emulator and open the InfiniTrace *Observe* session.

   See Opening VVM Sessions for more information.

5. Open a new database in the xeDebug or vvmDebug GUI for hardware probes. This database is opened automatically in SimVision.

6. Open the xeDebug or vvmDebug *DUV Design Browse*r window. Select signals of interest and use the *Probe* and *Display* button to probe these signals. The signals appear on the waveform. The *DUV Design Browser* window is same in both xeDebug and vvmDebug.

   For information on how to use the *DUV Design Browser*, see the DUV Design Browser section in Appendix 42, “Using the xeDebug GUI”.

7. Use the *Upload* button to generate data for the signals. The data is uploaded upto the current debug location marker.

   - vvmDebug goes to the specified time point of the debug location marker.
   - If you are using InfiniTrace, xeDebug goes to the same time point as the debug location marker before upload.
   - If you are using xeDebug in the offline mode, a `Pd_Marker` is created and the time between this marker and the current simulation time specifies the trace window before upload.

   Note: Any signal gaps are automatically collapsed, even if the waveform is already displayed. Going forward in time accumulates the data. Going backward in time overwrites the previous data.

8. You might step through the code in the *IDA Source Code Viewer*. The debug location marker on the waveform updates automatically to track the position in time. If you move

© 2021 All Rights Reserved.

in time in vvmDebug or during the *Observe* session, the time in Debug Analyzer is also updated.

Viewing Modports in Indago

You can generate waveforms in Indago using Palladium generated SHM waveform files. Indago expects Palladium generated SHM waveform files to use the original System Verilog naming notation.

For SV interface, the interface array must be a scope with the following naming convention:

`<interface>[<num>]`

where, the interface modport must be a scope under interface.

The SV interface array might follow the naming style as either `<interface>_ <num>_` , or `<interface>[<num>]`. However, for Indago, you must use the following: `<interface>[<num>]`.

To get the naming style as `<interface>[<num>]`, use the `-rtlNameForGenerate` option of the `ixcom` command while compiling your design. Without the `-rtlNameForGenerate` option, the interface array has the `<interface>_<num>_` naming style.

For more information on this option, refer to Enabling Run-time Access of For-generate Blocks and Instance Arrays Using Original RTL Names and the *Command and System Tasks for SA Mode* chapter of the *VXE Command Reference Manual*.

Start Indago by loading the SHM waveform files with the original HDL generated LWD database. Execute the following command to launch Indago:

```
indago -lwd <lwd> -db <shmFIleName>
```

Where,

- `<lwd>` specifies the name of the LWD database.
- `<shmFileName>` refers to the name of the SHM file.

For example, `indago -lwd ./lwd/xcelium.d -db trace.shm`

If RTL is as follows:

```
interface Intf(input clk);
   bit  [7:0] pingC;
   bit  [7:0] pingD;
   modport MC(input clk, output pingC, output pingD);
   modport MR(input clk, input pingC, input pingD);
endinterface // Intf
module Top(output [3:0] [7:0] ctr, input clk);
```


© 2021 All Rights Reserved.

```
    Intf i(clk);
    Mid  m0(i, 0, ctr[0]);
    Mid  m1(i, 1, ctr[1]);
    Mid m2(i, 2, ctr[2]);
    Mid  m3(i, 3, ctr[3]);
    INCR  INCR_inst(i);
endmodule // Top
```

Xcelium 18.9 Indago Design Hierarchy panel shows the modport `MC/MR` under interface scope `Top.i`. The interface `i` is also shown in the *Signals* panel for the instance scope `Top.INCR_inst, Top.m0, …., Top.m3.`

© 2021
All Rights Reserved.

47

Using Novas Tools

In both SA and ICE modes, Palladium Z1 enables you to view waveforms of FSDB format using standalone Novas tools, such as Verdi and Siloti. Before using these tools, you need to integrate them with the xeDebug tool as described in the following section.

The integrated standalone Novas tools support the following:

- RTL Verilog, VHDL, and mixed-HDL designs
- Gate-level designs (for backward compatibility)
- Verigen's trace output Verilog files (for backward compatibility)
- FSDB waveform database for both RTL and Gate-level designs without setting the verilog delimiter rule or any additional compiler options

Supported FSDB Versions

Palladium Z1 supports FSDB versions 5.1, 5.7, and 5.8 in both SA and ICE (online and offline) modes. The default FSDB file format is 5.1.

Multiple FSDB file generation, waveform streaming, and modular compilation is supported for all the supported versions.

Integrating Novas Tools with xeDebug

Before opening the waveform database generated using xeDebug, determine the installation of the Novas tools (Verdi or Siloti). Set the Novas license file and other Novas-related environment variables. In addition, specify the executable name (Verdi or Siloti) by setting the `NOVAS_EXE` environment variable.

© 2021 All Rights Reserved.

Setting the Environment Variables for Running Novas Tools

Set the required environment variables for generating the required FSDB format and to use Verdi3 tools to read FSDB format as follows:

```
# To generate FSDB dump
setenv NOVAS_HOME <Verdi3_installed_path>
setenv VERDI_HOME $NOVAS_HOME
setenv TURBO_ETC_DIR $NOVAS_HOME/etc/access


# Set Palladium schematic primitives
setenv TURBO_LIBS qds_u


# set Palladium Z1 schematic symbols path for 64bit design
setenv TURBO_LIBPATHS $QTHOME/share/vxe/gift/novas_schematic/64
setenv NOVAS_LIBPATHS $QTHOME/share/vxe/gift/novas_schematic/64


# Set Novas tool's license file
setenv LM_LICENSE_FILE $NOVAS_HOME/license/license.dat:$LM_LICENSE_FILE
set path = ($NOVAS_HOME/bin $path)


# Set NOVAS_EXE for xeDebug
setenv NOVAS_EXE $NOVAS_HOME/bin/verdi           //(or siloti)
setenv NOVAS_VERICOM $NOVAS_HOME/bin/vericom
```

Note:

- Use the `xeDebug -fsdb -fsdbVersion` <*number*> command in the xeDebug command line. You might also use the `XE_FSDB_VERSION` <*number*> environment variable to choose the required FSDB format.

  ```
  setenv XE_FSDB_VERSION 57 # for fsdb version 5.7
  setenv XE_FSDB_VERSION 58 # for fsdb version 5.8
  ```

  If the `XE_FSDB_VERSION` variable is set, then the `-fsdbVersion` option used at xeDebug command line will be ignored.

- You must add the following two additional shared libraries to the `LD_LIBRARY_PATH` variable:

  ```
  $NOVAS_HOME/share/PLI/IUS/LINUX64
  $NOVAS_HOME/share/PLI/IUS/LINUX64/boot
  ```

  For example:

  ```
  setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:$NOVAS_HOME/share/PLI/IUS/
  LINUX64:$NOVAS_HOME/share/PLI/IUS/LINUX64/boot
  ```

  If the `LD_LIBRARY_PATH` variable is not set as shown above, the FSDB file generation fails in SA mode.

© 2021 All Rights Reserved.

- To enable the xeDebug GUI to launch newer versions of Verdi3 released after 2015, you need to set the `NOVAS_VERDI3x` environment variable. To use an older version of Verdi3 (released in 2014), use the `NOVAS_VERDI3` environment variable.

Tracing and Uploading Waveforms

Palladium Z1 supports waveforms of SST2 (also known as SHM) and FSDB formats. By default, waveforms are generated in SST2 format.

The traced waveform database can be viewed by uploading it to the appropriate waveform viewer that is selected based on the waveform format.

You can choose to view the waveforms in the FSDB format by specifying the `-fsdb` option of the `xeDebug` command at the time of invoking xeDebug. After invoking xeDebug, you cannot change the waveform file format. The FSDB format is supported to enable easy debug of Palladium-generated databases when using Novas debugger tools like Verdi and Siloti.

```
xeDebug -fsdb -gui &
```

To use a specific FSDB format, specify the `-fsdbVersion` option with the `xeDebug -fsdb` command. The `-fsdbVersion` option is supported in both SA and ICE modes. The default FSDB version is 5.1.

```
xeDebug -fsdb -fsdbVersion <n>
```

Where <*n*> can be 51, 57, or 58.

For example:

```
xeDebug -fsdb -fsdbVersion 57
```

xeDebug shows the FSDB version number in the `xe.msg` file and the FSDB writer log file.

For FSDB format waveforms, you need to use the Novas waveform tools, such as Siloti and Verdi. To open the waveforms in Verdi waveform tool installed on your machine, use the following set of commands:

To view the waveforms, you must use the current version or a higher version of Verdi. For example, for FSDB 5.7 file, use Verdi 5.7 or higher version.

From the UNIX/Linux shell:

```
%         verdi -nWave -showtrigger -ssf <fsdb_database_name> &
%         verdi -nWave -showtrigger &
```

From XE command-line when running under the xeDebug script mode:

```
XE>       exec verdi -nWave -showtrigger -ssf <fsdb_database_name> &
XE>       exec verdi -nWave -showtrigger &
```


© 2021 All Rights Reserved.

If *<fsdb_database_name>* is not specified on the command line, the FSDB format waveforms can be opened from the GUI.

When running xeDebug GUI mode, the waveform can be displayed by clicking the waveform icon at the top of the GUI. The waveform can also be opened automatically if the *Auto Wave* option in the *Control* tab is turned on.

Note: If you zoom Novas nWave to 100% while the xeDebug is either generating the waveform data or sending the commands to Novas nWave, you might see the nWave waveform displayed incorrectly.

Changing WaveExtendLastTick Setting

The waveform setting needs to be changed for all Verdi3 versions as follows:

- On the nWave window, click *Tools > Preferences > Waveform > View Options > Waveform Pane > General*. Deselect the *Extend the Last Tic* check box. Refer to Figure 47-1 on page 2147.

  Or,

- Edit the `novas.rc` file and set `WaveExtendLastTick` to `FALSE` as follows:

  ```
  WaveExtendLastTick = FALSE
  ```


© 2021 All Rights Reserved.

Figure 47-1 Extend the Last Tic

When Verdi3 is launched, the *Extend the Last Tic* check box is selected or deselected automatically as follows:

1. The directory from where you launch Verdi has the `novas.rc` file that contains the following waveform setting:

   ```
   WaveExtendLastTick = FALSE
   ```

   Or,

   ```
   WaveExtendLastTick = TRUE
   ```

   If `WaveExtendLastTick` is set to `TRUE`, the *Extend the Last Tic* check box is selected, which might display a flat waveform.

2. If Verdi is launched in a directory that has no `novas.rc` file, a message is displayed as follows and Verdi tries to read the `novas.rc` file available in the home directory:

   ```
   rc = /home/<username>/novas.rc
   ```

   This file affects the `Extend the Last Tic` setting as mentioned above.

If no `novas.rc` is available in the home directory, the *Extend the Last Tic* check box is selected by default. You need to deselect the check box to eliminate the flat waveform problem.

© 2021 All Rights Reserved.

Functions Supported by Novas Tools

After integrating the standalone Novas tools with the xeDebug tool, following debug functions are supported:

- The Novas Browser/Schematic can be opened from the xeDebug tool by using the xeDebug icons or RTL/Gate tab.
- The drag and drop functionality is supported among the Novas Source Code Browser, Design Browser, Schematic, and Waveform tools.

  Note: Drag and drop between the xeDebug tool and Novas tools is not supported.
- Back annotation can be turned on in all Novas Browsers.

  Note: Back annotation for flattened verilog files (the Verigen's net trace output) can be supported automatically if it is opened from xeDebug RTL/Gate GUI. If the flattened verilog file needs to be opened using UNIX command manually, use `-play` option to include the name translation function.

  Example for the flattened verilog file, `trace_back_20.v`

  ```
  % verdi trace_back_20.v -play
  <VXE_INSTALL_HOME>/share/vxe/etc/qel/tdname.tcl &
  ```
- The Novas nWave waveform viewer can be integrated into the xeDebug ICE and Vector Debug modes.
- Result waveform databases are automatically compared with the stimulus database and the result mismatches are displayed.
- Following xeDebug run-time functions are available in the EMUTools pull-down menus of the Novas Source/Schematic/Waveform windows:

© 2021 All Rights Reserved.

Figure 47-2 Novas Tool (Verdi) Source Code Browser with xeDebug Functions

- Novas Source Browser - nTrace (refer to Figure 47-2):
  - addProbe
  - DisplaySelected->nWave
  - Switch Annotation TB/DUV
  - addBreakpoint
  - delBreakpoint
  - delAllBreakpoint
  - Testbench Selection...
  - DUV Probe Selection...

© 2021 All Rights Reserved.

- OpenXeDebug
- CloseXeDebug

Figure 47-3 Novas Tool (Verdi) Schematic Browser with xeDebug Functions

- Novas Schematic Browser - nSchema (refer to Figure 47-3):
  - Switch Annotation TB/DUV
  - AddProbe
  - Upload DUV Probe
  - DisplaySelected->nWave
  - addNotFound
  - Testbench Selection...
  - DUV Probe Selection...

© 2021 All Rights Reserved.

- ❍ ShowXeDebug
- ❍ ExitXeDebug

Figure 47-4 Novas Tool (Verdi) nWave Browser with xeDebug Functions

- ❑ Novas Waveform Browser - nWave (refer to Figure 47-4):
  - ❍ About nWave EMU tools
  - ❍ Upload
  - ❍ Hide xeDebug
  - ❍ FileNameOn
  - ❍ FileNameOff
  - ❍ ShowXeDebug
  - ❍ Exit Verdi
  - ❍ ExitXeDebug

© 2021 All Rights Reserved.

There are two database in IXCOM Flow. One database, named `<database>.sw.fsdb` is for dumping all the testbench signals and can include DUT signals. The other database, named `<database>.hw.fsdb` is for DUT signals only. If you have added the DUT signals into testbench database before swapping-in to hardware, then after the swapping-in to hardware, the DUT signals in `<database>.sw.fsdb` do not change (you have to ignore the value in `<database>.sw.fsdb` and read the value from DUT database). After swapping-out, the DUT database `<database>.hw.fsdb)` stops dumping.

Figure 47-5 RTL/Gate Tab in the xeDebug Tool

- RTL and Gate-level schematic databases can be debugged using the *RTL/Gate* tab of the xeDebug tool (refer to Figure 47-5). The tab provides the following buttons for debugging such databases:
  - Probe & Display Selected Nets
  - Probe Selected Insts
  - Probe Selected As Cone

© 2021 All Rights Reserved.

Before you start debugging an RTL schematic database, you must open it using either of the following ways:

- Select the check box adjacent to the *Command Options File* field within the *RTL Schematic Viewer* section and specify the name of the input command file, that is, `turbo.f` file. Then click the *Open RTL Viewer* button.
- Specify the name of the schematic OH database in the *Schematic DB* textbox. Then click the *Open RTL Viewer* button.

  Note: The RTL schematic database must be created before running the xeDebug tool.

To open a gate-level schematic database for debugging, use either of the following ways:

- Select the *Use Existing DB* option button and then specify the name of the required gate-level schematic OH database in the *Schematic DB* textbox.
- Select the *Use Existing Verilog File* option button and then specify the name of the required verilog input file in the *Verilog File* textbox.
- Select the *Generate Schematic DB* and/or *Generate Verilog File* option buttons. Then, select the *Instance* option button to create and view instance dump of whole or partial design; or select the *TraceNet* option button to create and view flattened trace results.

© 2021 All Rights Reserved.

© 2021 All Rights Reserved.

Index

Symbols

+dut 634
+sv 1210
+tfReadOnly 795, 1179
+vmgr 959, 1267
$axis_a2m 1177
$axis_exec 767, 1181
$axis_m2a 1177
$axis_rwrite 774, 1176
$display 767, 1182
$export_frcrel 795, 1179
$export_read 795, 1179
$monitor 767, 1182
$random 1174

A

acquiring samples 2045
adding VHDL operators 539
aggregates
    VHDL 543
Algorithmic State Machine (ASM) 581
algorithms
    cycle-fixed 582
always blocks 477, 483
    combinational 484, 488
    sequential 484, 486
-arch
    vhshell 105
architecture
    bodies 528
arithmetic operations
    VHDL 563
array types
    VHDL 535
arrays
    multiple dimensional 536
    unconstrained 535
assertion statements
    VHDL 549
assigning
    continuous 482
    tristate 482
    wires 482
asynchronous counters
    VHDL 575
asynchronous Mealy models 579
asynchronous reset
    registers 484
    VHDL 583
asynchronous set
    registers 486
asynchronous set/reset signal 569, 573
ATG pin assignment 329
attributes
    VHDL 542
axis_tbcall 1716

B

-b
    vhdel 98
behavioral descriptions
    Verilog 476
binary memory file format 91
bind 1217
bit strings 541
block statements
    VHDL 554
blocks
    always 477, 483
    combinational always 484, 488
    sequential always 484, 486
bp (.bp) file 157
buses
    VHDL 578

C

-c
    vhdel 98
capturing samples
    trace memory 1799
case statements
    VHDL 546
CBC vectors 94
character
    strings 541

© 2021 All Rights Reserved.

character literals
VHDL 541
CLI
See Command Line Interface (CLI).
clocked process
VHDL 572
code writing considerations 309
combinational
always blocks 484, 488
logic 483
Command Line Interface (CLI) 56
-comp
vhshell 104
compiling
designs 607
component instantiations
VHDL 554
concurrent cover statement 1216
concurrent statements
signal assignment 552
VHDL 550
Conditional Acquisition Methods 1981
conditional signal assignment statements
VHDL 552
constants
VHDL 529
context
DPI keyword 1403
continuous assignment 482
control files
pulse 85
counters
decrementing 2043
incrementing 2043
creating a working library 244
creating additional working libraries 244
cross-module reference (XMR) 1170, 1176
C-side Interface 1341
cursor
in the debugging environment 56
Cycle-By-Cycle (CBC) 94
cycle-fixed algorithms
VHDL 582

D

-d
vhmake 103
data objects
VHDL 529
data types
VHDL 532
Debugger
file suffixes 1968
Debugger
file suffixes 1967
decrementing counters 2043
defparam statement 483
delays
modules 482
descriptions
external bit out (XBO) configuration 85
design clock
specifying 1172
design directory 243
design import preparation 243
design library
creating 310
design placement
forcing 157
designs
modeling 483
units 528
Verilog 475
digital system 528
domains
fitting designs into 112
DPI
overview 1392
using 1403
drag and drop signals 1892
drive strengths
modules 482
drivers
signals 568
DUT (Device Under Test) 52
dynamic target 49

E

-e
vhdel 98
editing
SDL programs 2001
entity declaration 528
enumeration literals
VHDL 541
enumeration types
VHDL 534
exit statements

© 2021 All Rights Reserved.

VHDL 549
export
DPI keyword 1403

F

-f
vhshell 105
files
<cell>.et3confg 110
formats 83
memory pattern 87
flipflops
VHDL 572, 573
for loop statements
VHDL 547
for statement
VHDL 555
formats
files 83
forms
to save and reload data 1966
function calls
VHDL 544
functional coverage 1215
functions
Verilog 477
VHDL 550, 557

G

-g
vhshell 105
generate statements
VHDL 555
generic constants 528
VHDL 531
Global Definition Section 2006
global expressions
filtering 1966
Greatest Common Divider (GCD) 581, 582
group
in the debugging environment 56

H

-h
vhls 99
vhmake 103
hard IP mapping 327
HDL
import process 246 to 252
Verilog 476
VHDL designs 249 to 250
HDL-ICE Compiler 246
high-impedance signals 577

I

ICE mode
acquiring samples 2045
ICE Prepare 329
identifiers
VHDL 541
IEEE Standard Logic Package 561
if statements
VHDL 546, 556
imporitng a Verilog design 246
import
DPI keyword 1403
full 246
import functions and tasks 1396
importing
memory cell instances 87
importing a VHDL design 249
importing designs 308
include 618
incrementing counters 2043
indexed names
VHDL 542
inferences
latches 488
registers 484
InfiniTrace 1791
initial block 1170
Instance Definition section 2007
instances
parameters 483
integer types
VHDL 535
IP
physical 327
IP Builder 328
ixc_pulse 786
ixc_pulse#(0) 786
ixc_pulse#(1) 787
ixvcg.log 640

© 2021 All Rights Reserved.

K

keywords
  Verilog 479

L

-l
  vhls 99
LA mode 49
latches 572
  inferences 488
  VHDL 572
-libmaprc 98, 100, 618
libmaprc file 617
libxsc 644
literals
  VHDL operands 540
loading counter values 2042
-loadsc 642
logic
  combinational 483
  latches 488
  sequential 483
Logic Analyzer
  triggering 2044
logical library 617
logical operators
  VHDL 538
loop statements
  VHDL 547
loop variables
  VHDL 532
loops
  asynchronous 321
  manually breaking 322

M

Mealy models
  VHDL 579
memories
  file format 90
memory
  trace 1799
memory cells 308
memory instances
  bit number of a terminal 89
  cell definitions 87
  cell name patterns 87
  custom ASIC library 87
  literal value 90
  memory patterns 89
  pattern name rules 88
  pin name definitions 87
  port 89
  port numbering 89
  reference value 90
  single-port memory cells 89
  supported terminal types 89
memory pattern file 87
memTran memory format 92, 401
message URL ../vxecmdref/IXCOM_CmdRef.html#ixcom 1735, 1838
mixed HDL
  SystemC simulation flow 641
modeling
  designs 483
modules
  delays 482
  drive strengths 482
  parameters 483
monitor code
  dealing with 1180
  marking 766, 1181
  testbench 1170
Moore models
  VHDL 578
multiple assignment
  wires 482
multiple dimensional arrays
  VHDL 536
Multiple SDL Instances 1979
multiplying VHDL operators 539

N

NCSC_EXPORT_MODULE 642, 644
  treatment of commas 643
NCSC_EXPORT_MODULE macro
  for topmost modules 643
Netlist Importer
  memory pattern files 87
netlists
  synthesis 253
next statements
  VHDL 548

© 2021 All Rights Reserved.

null statements
VHDL 549
numeric literals
VHDL 541
NUMERIC_BIT
VHDL 562
NUMERIC_STD
VHDL 562

O

offline debug 56, 1819
operands
VHDL 540 to 544
operators
adding 539
logical 538
multiplying 539
overloading 540
relational 538
unary 539
VHDL 537
output enable signal 577
overloading VHDL operators 540

P

-p
vhdel 98
package bodies
VHDL 560
package declarations
VHDL 560
packages
VHDL 559
parameters
defining 483
instances 483
modules 483
pattern file 87
cell names 90
literal set 88
positional set 88
syntax 88
terminal properties 89
values 89
wildcards 89
physical directory 617
ports
VHDL 530
predefined VHDL operators 537, 539
Preferences
Filter tab 1882
GUI Variables tab 1884
Message tab 1881
prepare
ICE 329
primitives
memory 391
Print-on-Change (POC) 94
procedures
VHDL 550, 557
process statements
VHDL 551
Programming Language Interface
(PLI) 1178
programs
SDL 2001
pure
DPI keyword 1403

Q

qualified expressions
VHDL 543
qualifying variables 489

R

race condition
testbench 1175
random.hex 1175
rccgen.log 640
record types
VHDL 536
registers
asynchronous reset 484
asynchronous set 486
inferences 484
VHDL 574
relational operators
VHDL 538
reset
asynchronous 484, 583
synchronous 583
reset signal
asynchronous 569, 573
synchronous 573

© 2021 All Rights Reserved.

resolution functions
VHDL 558

S

-s
vhls 100
samples
number to trace 1799
SC_CTOR 645
SC_CTOR() 644
SC_FOREIGN_MODULE 645
sc_foreign_module 642, 644, 645
sc_main() 643
SC_MODULE 642
sc_module_name 644
sclan 642
SDL
decrement statements 2043
editing 2001
goto statements 2042
increment statements 2043
load statements 2042
start statements 2043
stop statements 2044
syntax 2005
trigger statements 2044
selective signal assignment statements
VHDL 553
sequential
always blocks 484, 486
logic 483
sequential logic 483
sequential statements
VHDL 544 to 548
set signal
asynchronous 569, 573
synchronous 573
signal assignment statements
VHDL 545
signals
asynchronous set/reset 569, 573
high-impedance 577
multiple drivers 568
output enable 577
synchronous set/reset 573
VHDL 530
slice names
VHDL 542
STANDARD package
VHDL 561
statements
defparam 483
unsupported 481
static target 49
STB mode 49
stopping SDL 2044
string literals
VHDL 541
structural descriptions
Verilog 476
subprograms
VHDL 550, 557
subtypes
VHDL 537
Support for Encryption in IXCOM 806
synchronous counters
VHDL 576
synchronous reset
VHDL 583
synchronous set/reset signal 573
synthesis
emulation 258
netlists 253
tristate logic 490
Verilog 253, 475
synthesis directives
VHDL 566
system tasks
Verilog 479, 481
SystemC
case sensitivity in VHDL, Verilog, and SystemC 643
comma separators in SystemC 643
cross-boundary port symbolic names 643
exporting SystemC modules 643
HDL instantiated in SystemC 644
HDL wrapper class from sc_foreign_module 644
instantiated module types SystemC declarations 643
libxsc SystemC library extension 644
mixed HDL simulation flow 641
NCSC_EXPORT_MODULE macro 643
sc_foreign_module class 644
SC_FOREIGN_MODULE macro 644
sc_module class 644
sc_module_name constructor argument 643
scshell utility 644

© 2021 All Rights Reserved.

SystemC instantiated in HDL 643
templatized module names
restriction 643
VHDL instantiated in SystemC 644
wrapper class
writing 644
SystemVerilog Direct Programming
Interface (DPI) 1391

T

target 49
target system 49
tbcall_region 772
testbench
synchronous 1173
writing guidelines 1169
Testbench Callback 768
automated method 769
TEXTIO package
VHDL 561
timing constraints
IP 327
trace memory
capturing samples 1799
traced signals
uploading 54
tracing
number of samples 1799
triggering
Logic Analyzer 2044
tristate
assigning 482
tristate buffers
VHDL 577
tristate gates
VHDL 577
tristate logic
synthesis 490
type conversion
VHDL 544

U

-u
vhshell 105
-ua 634
unary operators
VHDL 539
unconstrained arrays
VHDL 535
undriven wires 482
uploading
traced signals 54
Using the ncpli2tab Utility 1378

V

-v
vhdel 98
variable assignment statements
VHDL 546
variables
qualifying 489
VHDL 530
VD mode 49
VDL
slice names 542
vectors
CBC 94
Verilog
behavioral descriptions 476
descriptions 476
designs 475
functions 477
HDL 476
keywords 479
mixed descriptions 476, 477
structural descriptions 476
system tasks 479, 481
Verilog import 246
veriuser.c 1403
vhan.log 630, 631
vhdel 98
VHDL
aggregates 543
arithmetic operations 563
array types 535
assertion statements 549
asynchronous counters 575
asynchronous reset 583
attributes 542
block statements 554
buses 578
case statements 546
character literals 541
clocked process 572
component instantiations 554
concurrent statements 550, 552

© 2021 All Rights Reserved.

conditional signal assignment statements 552
constants 529
cycle-fixed algorithms 582
data object definitions 532
data objects 529
enumeration literals 541
enumeration types 534
exit statements 549
flipflops 572, 573
for loop statements 547
for statement 555
function calls 544
functions 550, 557
generate statements 555
generic constants 531
identifiers 541
if statements 546, 556
indexed names 542
integer types 535
latches 572
literal operands 540
logical operators 538
loop statements 547
loop variables 532
Mealy models 579
Moore models 578
multiple dimensional arrays 536
next statements 548
null statements 549
numeric literals 541
NUMERIC_BIT 562
NUMERIC_STD 562
operands 540 to 544
operators 537
package bodies 560
package declarations 560
packages 559
ports 530
predefined data types 532 to 534
predefined operators 537, 539
procedures 550, 557
process statements 551
qualified expressions 543
record types 536
registers 574
resolution functions 558
selective signal assignment statements 553
sequential statements 544 to 548
signal assignment statements 545
signals 530
STANDARD package 561
string literals 541
subprograms 550, 557
subtypes 537
synchronous counters 576
synchronous reset 583
synthesis directives 566
TEXTIO package 561
tristate buffers 577
tristate gates 577
type conversion 544
unconstrained arrays 535
variable assignment statements 546
variables 530
wait descriptions 583
wait statements 545
while loop statements 548
working library 247, 249
VHDL import 246
VHDL operators
adding 539
multiplying 539
overloading 540
relational 538
unary 539
vhelab 626
vhlib 620
vhls 99
vhshell 104

W

wait descriptions
VHDL 583
wait statements
VHDL 545
while loop statements
VHDL 548
wires
multiple assignment 482
undriven 482
-work
vhdel 98
vhls 99
vhmake 103
vhshell 105
working library 243
VHDL 247, 249

© 2021 All Rights Reserved.

X

Xcelium 617
XEL
    invoking 78
-xmvhdl
    vhshell 106
xrun 643

© 2021 All Rights Reserved.