

Verification Continuum™

ZeBu® Server User Guide

Version Q-2020.03-1, July 2020

SYNOPSYS®

Copyright Notice and Proprietary Information

©2020 Synopsys, Inc. All rights reserved. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at <http://www.synopsys.com/company/legal/trademarks-brands.html>.

All other product or company names may be trademarks of their respective owners.

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

www.synopsys.com

Contents

Preface.....	21
About This Book	21
Contents of This Book	21
Related Documentation	23
 1. ZeBu Server	 25
1.1. Overview Of ZeBu Server	26
1.2. Hardware Overview.....	27
1.2.1. Design Capacity	28
1.2.2. FPGAs in ZeBu Server	29
1.3. ZeBu Emulation Flow.....	31
1.4. Introduction to zCui	32
 2. Assembling a Design for Emulation	 33
2.1. Mapping Your Design and Test Environment	34
2.1.1. Importing a Design From a Simulation Environment	34
2.1.2. Preparing a DUT-Only Environment	35
2.2. Clock Modeling	35
2.3. Reset Modeling.....	36
2.4. Memory Modeling	36
2.4.1. Inference From a Design	36
2.4.2. Memory Model IP	40
2.5. Designating Signals for Access During Runtime	41
2.5.1. Reading Signals	41
2.5.2. Forcing and Injecting Values	41
2.6. Verilog System Tasks	42
2.6.1. \$memset() System Task.....	47
2.6.2. \$plusargs System Task.....	48
2.6.3. \$fopen and \$fclose System Tasks	50
2.6.4. \$finish System Task	52
2.7. Verilog Procedural Interface	53

3. Compilation.....	55
3.1. Overall Unified Compile Flow.....	56
3.1.1. Introduction to UTF.....	58
3.1.2. Basic UTF Commands.....	58
3.2. Word-Level Synthesis.....	59
3.2.1. WLS Logs	60
3.3. Compilation Script for VCS (UUM)	61
3.3.1. Setting Up VCS	61
3.3.2. Setting Up the synopsys_sim.setup File	61
3.3.3. Setting Up Compilation Commands Using UTF	62
3.3.4. Compilation Settings for DesignWare Blocks	63
3.4. Specifying the Hardware Architecture for Compilation	63
3.5. Grid/LSF Handy Commands.....	64
3.6. Gate-Level Netlist Compilation	65
3.7. Compiling a Project	67
3.7.1. Compiling With zCui in the Batch Mode	67
3.7.2. Compiling With zCui in the GUI Mode	68
3.8. Important Log Files	73
3.8.1. Analyzing Compilation Results Using zBatchExplorer.....	73
3.8.2. zTime	75
3.8.3. Gathering Information of a Compilation Database	76
4. Clock Delay Support in Verilog for ZeBu	79
4.1. Enabling Clock Delay Feature	80
4.2. Support of Delays With zceiClockPort.....	80
4.3. Supported Verilog Language Subset.....	81
4.4. Unsupported Constructs	83
4.5. clockDelayPort Macro	84
4.6. Configuring clockDelayPort Through designFeatures.....	86
4.7. Timescale and Precision	87
4.8. Compilation Results.....	87
4.9. Timestamp Clock	88
4.9.1. ZeBu Server 3.....	88
4.9.2. ZeBu Server 4.....	88
4.10. C_COSIM and Vertical Solution Transactors	89
4.11. zDPI and ZEMI-3	90
4.12. Waveform Dumping	90

4.13. Global Time	91
4.14. Tolerance Support.....	92
4.15. Limitations	95
5. Runtime	97
5.1. Controlling Runtime Parameters	98
5.1.1. Emulation Speed	98
5.1.2. Clock Definition.....	98
5.1.3. Memory Initialization	100
5.2. zRun.....	101
5.2.1. Common zRun Command-Line Options	101
5.2.2. Controlling Clocks.....	104
5.2.3. Managing Memory	109
5.3. Cycle-Based C/C++ Co-Simulation	113
5.4. Reading Signal Values During Runtime	116
5.4.1. C++ Example	116
5.4.2. Tcl Example	117
5.5. Changing the Signal Values During Runtime	117
5.5.1. Depositing a value on a signal value.....	117
5.5.2. Injecting a Value on a Signal.....	118
5.5.3. Forcing a Value on a Signal.....	119
5.5.4. Releasing a Forced Signal	120
5.6. Randomizing Signals and Memories During Runtime.....	120
5.6.1. C++ Syntax	121
5.6.2. Tcl Syntax	122
5.6.3. Memory Randomization	122
6. RunManager	125
6.1. Enabling RunManager	126
6.2. RunManager APIs.....	126
7. Using SystemVerilog Assertions	129
7.1. SVA Compilation.....	130
7.2. Runtime Usage	130
7.2.1. Controlling Assertions From the C++ Testbench.....	131
7.2.2. Controlling Assertions Using zRun	136
7.2.3. Controlling Assertions Using zRun Tcl Commands	137

7.2.4. Runtime Options for Assertion Control Tasks.....	138
7.3. Post Processing SVA.....	139
7.4. Supported SVA subset	139
8. DPI Synthesis Support in a Design.....	145
8.1. Compilation	146
8.2. Writing DPI Import Functions	148
8.2.1. C/C++ Language Compatibility.....	148
8.2.2. Header Files.....	148
8.3. ZeBu API to Control DPI Functions at Runtime.....	149
8.3.1. Clocking Mode.....	149
8.3.2. Enabling Synchronization of DPI Calls.....	151
8.3.3. Selecting Function Calls Only on Input Changes	151
8.3.4. Loading Dynamic Libraries	152
8.3.5. Starting DPI Call Processing	152
8.3.6. Stopping DPI Call Processing	154
8.3.7. zRun Tcl Commands to Control the DPI Functions.....	156
8.4. Processing DPI Function Calls Offline	157
8.4.1. Identifying the DPI Functions	157
8.4.2. Enabling the DPI Offline Feature During Emulation	157
8.4.3. Creating a Shared Library	164
8.4.4. Creating an Input File for the zdpiReport Tool	164
8.4.5. Using the zdpiReport Tool	164
8.5. Optimization Guidelines	168
8.5.1. Enhancing Data Transfer.....	168
8.5.2. Defining Sampling Clock Frequency.....	170
8.5.3. Investigating Emulation Slowdowns	170
8.6. Limitations	172
8.6.1. Multiple Clock Groups.....	172
8.6.2. Performance	172
8.6.3. zRun GUI	172
8.6.4. Synthesis	173
8.7. Example	173
8.7.1. Design	173
8.7.2. Implementation of DPI Functions	176
8.7.3. Controlling DPI Calls from the Testbench.....	177
8.7.4. Controlling the Calls from zRun.....	179

9. Power Aware Verification With ZeBu.....	183
9.1. UPF Commands Supported by ZeBu.....	184
9.2. ZeBu Power Aware Emulation Flow	187
9.2.1. UPF Support in Unified Compile	187
9.2.2. ZeBu Power Aware Compilation	190
9.2.3. Corruption in ZeBu Power Aware Emulation Flow.....	190
9.2.4. UPF File Example.....	191
9.2.5. Low Power Optimization and Reporting Commands	196
9.3. Emulation Runtime.....	199
9.3.1. C++ Interface	199
9.3.2. C Language Interface.....	213
9.4. Limitations	218
9.5. Investigating Power Bugs	219
9.5.1. Checking Isolation between Power Domains	219
9.5.2. Examining the Power State of Design Instances.....	220
9.5.3. Examining the State of Power Domains	220
9.5.4. Retention Control Signals	221
9.6. Troubleshooting	222
9.6.1. Incorrect Order when Calling PowerMgt::init	222
9.6.2. Deprecated Emulation Runtime Control Methods.....	223
9.6.3. Failure when Calling any Power Aware Method	223
9.6.4. Failure When Calling any Retention-Related Method	224
10. Compiling ZeBu Transactors.....	225
10.1. Instantiating a Transactor.....	226
10.2. Compiling a Transactor	229
10.2.1. Example of a .utf file	229
10.3. Compiling a Transactor Example	230
10.4. Guidelines for Mapping Transactors in a Multi-RTB Environment.	231
11. Custom DPI Based Transactors With ZEMI-3	233
11.1. Choosing a Transactor Architecture	234
11.1.1. Selecting Hardware or Software Part for Processing	234
11.1.2. Avoid Using an Unnecessary Large Bandwidth	234
11.1.3. Use Multi-Cycle Transactions	234
11.1.4. Use Modular Transactor Architecture	235
11.2. Data Exchange Between Hardware and Software	235

11.2.1.	Simulation Time Consumption	235
11.2.2.	Hardware -Initiated Transactions.....	235
11.2.3.	Software -Initiated Transactions.....	236
11.2.4.	Mixing Imports and Exports.....	237
11.2.5.	Functions and Tasks	239
11.2.6.	Managing the Call Order of Software Import Functions.....	240
11.2.7.	Synchronization Points.....	240
11.2.8.	Streaming Data	241
11.3.	Clocks.....	243
11.3.1.	Controlled Clocks	243
11.3.2.	Sampled Clocks	243
11.3.3.	Ensuring the Best Performance	243
11.4.	Advanced Features	244
11.4.1.	Back-to-Back Mode	244
11.4.2.	Protocol Mode	245
11.4.3.	Prefetch Calls	246
11.4.4.	Serializing Calls	247
11.4.5.	Lossy Calls.....	247
11.4.6.	Port Buffering	247
11.4.7.	Multi-Cycle Messaging (Port Multiplexing).....	247
11.4.8.	Automatic Compiler Optimization	247
11.5.	Writing the Hardware Part	248
11.5.1.	Imports and Exports in SystemVerilog Code	248
11.5.2.	Using Behavioral Coding Style	250
11.5.3.	Adding Delta Delays	253
11.5.4.	Supported System Calls.....	254
11.5.5.	Advanced Properties.....	255
11.6.	Writing the Software Part.....	259
11.6.1.	Calling Export Functions/Tasks in C	259
11.6.2.	Implementing Import Functions in C.....	262
11.6.3.	C Types Supported by ZEMI-3.....	262
11.6.4.	SystemVerilog C Functions Supported by ZEMI-3	263
11.6.5.	Controlling the Back-to-Back Mode From Your Testbench	265
11.7.	Compiling the Hardware Part of a ZEMI-3 Transactor	266
11.7.1.	Instantiating a ZEMI-3 Transactor in the Hardware Top	267
11.7.2.	Adding and Setting a ZEMI-3 Transactor in the UTF File.....	267
11.7.3.	Compiling a ZEMI-3 Transactor with zCui.....	271
11.7.4.	Results of the ZEMI-3 Transactor Pre-Processing	271
11.7.5.	Compiling Output Files for the Software Part of the Transactor	273
11.8.	Running Emulation of ZEMI-3 Transactors.....	275

11.8.1. Prerequisites	275
11.8.2. Preparing the Files Required for Runtime	276
11.8.3. Launching zEmiRun.....	277
11.9. Running Emulation of ZEMI-3 Transactors in a Heterogeneous Environment	286
11.9.1. Using a SystemC Testbench	286
11.9.2. Using ZCEI Transactors and ZEMI-3 Transactors	290
11.10. Performing Simulation of ZEMI-3 Transactors.....	302
11.10.1. Writing a Top-Level Wrapper.....	302
11.10.2. Using a Software Entry Point	304
11.10.3. Launching the Simulation.....	304
12. Hybrid Emulation.....	305
12.1. Hardware Top Preparation	306
12.1.1. AMBA Transactors	306
12.1.2. SideBand Signals.....	306
12.1.3. LPDDRx Shared Memory	306
12.2. UTF Commands for Hybrid Emulation	307
12.2.1. Black-Boxing a Design During Synthesis	307
12.2.2. Defining the zTLM_RESETn Signal	307
13. Design Partitioning With zManualPartitioner.....	309
13.1. User Interface	310
13.1.1. Launching zManualPartitioner	310
13.1.2. Loading a Netlist File.....	311
13.1.3. Main Window	314
13.1.4. Netlist Tree Pane	315
13.1.5. Partition Pane.....	318
13.1.6. Partition Interconnection Matrix Pane	320
13.1.7. Instance Interconnection Matrix Pane	322
13.1.8. Timing Analysis Pane	324
13.1.9. Configuration Window	327
13.1.10. Properties Window	329
13.1.11. Loading and Generating Mapping Files	330
13.2. Managing Black-boxes.....	331
13.2.1. Applying a Black Box Attribute.....	331
13.3. Moving Instances Between Partitions	332
13.3.1. Using an EDIF File	332

13.3.2. Using an Automatic Clustering Interconnection File.....	334
13.3.3. Performing Timing Analysis	336
13.4. Limitations	339
14. Runtime Performance Analysis With zTune.....	341
14.1. Data Collected	342
14.1.1. Controlled Clocks	342
14.1.2. Clock Frequency	342
14.1.3. Clock Stopping Activity	343
14.2. Commands/ API Description	343
14.3. Compile Time Options.....	344
14.4. zTune Runtime	344
14.4.1. zTune C++ API.....	344
14.4.2. zEmiRun	346
14.4.3. zRci	346
14.5. Post Processing	348
14.5.1. zTune GUI.....	348
14.5.2. UPF Event Support in zTune	358
14.5.3. System Stoppers Information	359
14.5.4. zTune in Command Line Mode	359
14.5.5. CSV Output.....	365
14.6. Limitations	366
15. Saving the Design State and Restarting from a Saved State	367
15.1. Save and Restore - Recommendations.....	368
15.2. C++ Method for Save and Restore	368
15.3. C++ Example.....	371
15.4. C Function for Save and Restore.....	371
16. Direct - ICE for ZeBu Server 3	373
16.1. Direct-ICE Architecture and Functional Description.....	374
16.1.1. Power Supply and Interface Voltage Level	375
16.2. Functional Description of Direct-ICE Clocks.....	377
16.2.1. Global Clocks	378
16.3. Functional Description of Direct-ICE I/O Signals.....	379
16.3.1. Direct-ICE Target Integration With the DUT	379
16.3.2. Connecting an External Direct-ICE Target to the DUT	379

16.3.3. Connecting an Internal Target to the DUT	381
16.3.4. Delays	381
16.3.5. Direct-ICE Sampling Data	382
16.4. Physical Description	383
16.4.1. Connectors	384
16.4.2. LEDs.....	384
16.4.3. Direct-ICE Interface Connectors and FPGAs on a Direct-ICE Module	384
16.4.4. Direct-ICE Banks	385
16.4.5. Direct-ICE Data Connectors Mapping	387
16.4.6. Direct-ICE LEDs.....	387
16.4.7. Direct-ICE Connectors Physical Description.....	388
16.4.8. Direct-ICE Clocks Physical Description	389
16.4.9. Direct-ICE Pinouts	390
16.4.10. Data and Clock Timings for Direct-ICE Interface	394
16.5. Writing the Direct-ICE Compilation Script	398
16.5.1. Direct-ICE Compilation Script Commands.....	398
16.5.2. Direct-ICE Compilation Script options	400
16.5.3. Configuring the Direct-ICE Interface.....	405
16.5.4. General Rules for Signal Connections.....	412
16.5.5. Connecting Clocks	414
16.5.6. Connecting I/O Signals.....	417
16.5.7. Direct-ICE Declaring Signal Connections for an Internal Target.....	419
16.6. Compiling for Direct-ICE.....	426
16.7. Runtime With Direct-ICE	446
16.7.1. Specific Information in the ZeBu Runtime Database.....	446
16.7.2. Dynamic-Probes on the Direct-ICE interface	446
16.7.3. Delay Activation and Programming.....	449
16.7.4. Control of the Direct-ICE Shielding	450
16.7.5. Sampling Feature	451
16.8. Mechanical Data	454
17. Smart Z-ICE Interface for ZeBu Server 3	457
17.1. Connection Principle	459
17.1.1. Single-Unit Configuration	459
17.1.2. Multi-Unit Configuration.....	460
17.2. Physical Description of Smart Z-ICE Interface.....	462
17.3. Smart Z-ICE HE10 Adapter	465
17.4. Electrical Characteristics	468

17.5. Connecting a Pod to the Smart Z-ICE Connector	469
17.6. Driver Declaration	469
17.6.1. Smart Z-ICE Driver Instantiation.....	469
17.6.2. Declaring the Type of Physical Interface	472
17.6.3. Voltage Declaration	472
17.6.4. Data Declaration.....	473
17.6.5. Pin Declarations on Smart Z-ICE Connector	473
17.6.6. Pin Declarations With Smart Z-ICE HE10 Adapter	474
17.6.7. Ambiguous Pin Declarations	474
17.6.8. Clock Pin as an Input.....	475
17.6.9. Declaration for Synchronous Clock Handling With Smart Z-ICE.....	475
17.7. Remapping for Runtime.....	476
17.7.1. Declaration of the Connector in the designFeatures File.....	476
17.7.2. Declaration of the Connector Using an Environment Variable.....	477
17.7.3. Disabling Smart Z-ICE ports at Runtime.....	477
17.8. Examples.....	478
17.8.1. Mono-Directional Interface: Example 1	478
17.8.2. Mono-Directional Interface: Example 2	478
17.8.3. Example of Bi-Directional Interface	479
17.8.4. Example of SMART_ZICE_ZSE in UC Flow.....	480
18. Smart Z-ICE Support on ZeBu Server 4	481
18.1. Physical Description of Smart Z-ICE Interface.....	482
18.2. Instantiating Smart Z-ICE	484
18.3. Using the Smart Z-ICE During Emulation	486
18.3.1. Mapping Smart Z-ICE Using the Environment Variable	486
18.3.2. Mapping Smart Z-ICE Using the designFeatures Command	487
18.3.3. Disabling Smart Z-ICE ports at Runtime.....	487
19. Appendix.....	489
19.1. designFeatures File	490
19.1.1. Syntax Rules	490
19.1.2. Location of the Calibration Files	491
19.1.3. Declaring the Process Name	491
19.1.4. Parameters for Design Clocks	495
19.1.5. Parameters for Transactors.....	504
19.1.6. Initializing Memories	505
19.1.7. Programming the driverClk Reset Signal.....	506

19.1.8. Programming the DUT Reset	506
19.1.9. Declaration for Smart Z-ICE	506
19.2. Runtime Clock file	507
19.2.1. Clock File Content.....	507
19.2.2. Clock File Syntax	507
19.3. Memory Content File	510
19.3.1. ZeBu-Proprietary Binary Format	510
19.3.2. Memory Binary Format for Shorter Load Time.....	510
19.3.3. Memory Content File Text Format.....	511
20. zFmCheck	515
20.1. Preparation	516
20.2. Formal Equivalence with Formality	516
20.3. Formality Results	517
20.4. Constrained Random Testbench Simulation with VCS.....	518
20.5. Simulation Results	519
20.6. Advanced Simulation Options.....	520
20.7. Using zFmCheck for Large Design	520
20.8. Best Practices and Further Helpful Uses.....	523
20.8.1. Organize Your Results	523
20.8.2. Using -norm to Keep Simulation Data	524
20.8.3. Using DVE to Debug a Simulation Failure	525
20.8.4. Using Verdi to Debug a Simulation Failure	525
20.8.5. Using -save du to Launch Formality GUI	525
20.8.6. Why Use -scm Option?	526
20.8.7. Checking Status of a zFmCheck run	526
20.9. zFmCheck Use Model	526

List of Figures

ZeBu Server-2: 2-slot Front and Rear Panels.....	27
ZeBu Server-3: 5-slot Front and Rear Panels.....	28
Architecture of an FPGA Module in ZeBu Server 3	30
Architecture of ZeBu Server 4	30
ZeBu Emulation Flow.....	31
zCui Default View	32
Importing a DUT and Test Environment From a Simulation Environment	34
Preparing a DUT-Only Environment.....	35
ZeBu Unified Compile Flow.....	57
zCui Options.....	68
zCui GUI.....	69
Project Messages Panel.....	69
Launching Compilation in zCui	70
Viewing the VCS Task in zCui	70
Tasks to Job Queue Panel	71
Viewing the Verdi Task in zCui	72
Viewing the Verdi Task in zCui Log.....	72
Opening a Directory using zBatchExplorer	74
Selecting Log File and zCui Global File	74
Checking for Multiplexer/Hops on a Critical Routing Path	76
System Verilog Assertion Panel.....	136
UPF Flow in UC	187
Front-End Compilation Flow	189
Corruption in ZeBu Power Aware Emulation Flow	191
ZEBU_POWER_ON Signal in zSelectProbes.....	220
Retention Strategy Instances in zSelectProbes	221
Retention Strategy Signals in zSelectProbes	222
Hardware-Initiated Transaction	236
Software-Initiated Transactions	237

Calling an Export From an Import (Context Import)	238
Calling an Import From an Export	238
Streaming Import	241
Streaming Export	242
Back-to-Back Mode	245
Prefetch Mode	246
Compiling the Hardware Part of a ZEMI-3 Transactor	266
ZEMI-3 Runtime Environment	275
ZeBu Hybrid Emulation Platform	305
ManualPartitioner Welcome Screen.....	310
Loading an Automatic Clustering Interconnection File	312
EDIF File loaded in zManualPartitioner.....	314
Automatic Clustering Interconnection File loaded in zManualPartitioner.....	314
Netlist Tree Pane (EDIF File).....	315
Netlist Tree Pane (Automatic Clustering Interconnection File)	317
Partition Pane (An EDIF File)	319
Partition Pane (An Automatic Clustering Interconnection File)	319
View Menu	320
Partition Interconnection Matrix Pane (EDIF File)	321
Partition Interconnection Matrix Pane (Automatic Clustering Interconnection File)	321
Instance Interconnection Matrix Pane (An EDIF File)	322
Instance Interconnection Matrix Pane (Automatic Clustering Interconnection File)	324
Timing Analysis/Clock Path View	325
Configuration Window.....	327
Example of Sample Configuration File Loaded.....	328
Modifying Filling Rates in Configuration Window	329
Applying a Black Box Condition	331
Removing a Black Box Condition	332
Moving Instances in the Partition Pane Using an EDIF File	333
Result of Moving Instances in the Partition Pane Using an EDIF File....	334
Moving Instances in the Partition Pane Using an Automatic Clustering	

Interconnection File	335
Result of Moving Instances in the Partition Pane Using an Automatic Clustering Interconnection File	336
Element Information Window	337
Element Modified in the Timing Analysis Pane	338
Timing Analysis Pane Refreshed	339
Frequency Monitoring	342
Monitoring Clock Stopping Activity.....	343
Data Analysis in zTune	348
zTune Initial GUI.....	350
Overall View.....	351
Process ID.....	352
Threaded Bars.....	353
Thread View	353
Software Profile Configuration Page	354
Selecting a Range in Thread View	355
Function View	355
Caller/Callee View	356
Call Tree View	356
Markers View	358
UPF Events	358
Architecture of the Direct-ICE Interface with a 9F/ICE Module.....	374
Direct-ICE Global Output Clocks Connection	378
ZeBu-Centric I/O Convention	379
Connecting an External Direct-ICE Target to the DUT.....	380
Connecting an Internal Target to the DUT	381
Direct-ICE Interface Connectors	383
Direct-ICE Interface Connectors Versus FPGAs with a 9F/ICE FPGA Module	385
Direct-ICE Connectors with Bank Distribution for one 9F/ICE Module..	386
Direct-ICE I/Os Mapping	387
LEDs on the ZeBu Server 3 Direct-ICE Interface board	388
12-pin Connectors Pinout.....	393
Pinout for Data Connectors	393

Conditions for Timing Measurements With Output Clocks	395
Definition of Timings for Input Data.....	395
Definition of Timings for Output Data.....	396
Delay Options for I/O Pad.....	409
Using the -pin Option Alone.....	413
Using the -pin Option with -target_path	414
Interconnecting Direct-ICE Target Pins.....	421
zSelectProbes Display for Direct-ICE Debugging Elements.....	446
Dynamic-Probes on Mono-directional Signals (pin location label)	447
Dynamic-Probes on Bi-directional Signals (pin location label)	448
Dynamic-Probes on Mono-directional Signals (design object name)	448
Dynamic-Probes on Bi-directional Signals (Design Object Name)	449
Delays Activation and Programming	450
Direct-ICE Shielding Control.....	451
Sampling on Mono-directional Signals.....	452
Sampling on Bi-directional Signals	453
Mechanical Position for Direct-ICE Connectors.....	454
Smart Z-ICE Connectors on the Rear Panel of a Two-slot Unit.....	457
Smart Z-ICE Connectors on the Rear Panel of a Five-Slot Unit	458
Connection Principle for a Two-Slot Unit	459
Connection Principle for a Five-Slot Unit	460
Connection Principle in a Multi-Unit Configuration	461
Smart Z-ICE Connectors on the Two-Slot ZeBu Server Unit	462
Smart Z-ICE Connectors on the Five-Slot ZeBu Server Unit	463
Pin Numbering on a Smart Z-ICE Connector.....	464
Pin Allocation on a Smart Z-ICE Connector.....	464
ZeBu Server Smart Z-ICE HE10 Adapter	465
Pin Allocation on the Smart Z-ICE HE10 Adapter.....	467
Pin Numbering on the Smart Z-ICE HE10 Adapter	467
Connection to FICE FPGA	468
Smart Z-ICE Port Connectors on ZeBu Server 4 Control Interface	482
Pin Allocation in the ZeBu Server 4 Smart Z-ICE HE10 Connector	483

List of Tables

Design Capacity for ZeBu Server	28
FPGA Type and FPGA Module Types in ZeBu Server	29
Supported Verilog System-Tasks.....	42
UTF Help Commands	58
Basic UTF Commands	58
WLS Logs	60
Sun Grid Engine and Platform LSF Recommendation.....	64
ClockDelayPort Macro	84
APIs Supported with RunManager.....	126
zRun Tcl Commands	137
Supported SystemVerilog SVA Subset	139
zdpiReport Mandatory Parameters	165
zdpiReport Options	165
Mapping Between SystemVerilog Types with C Types	169
Supported UPF Commands for ZeBu	184
config_upf Command Options	197
C++ API to Control Power Aware Verification: PowerMgt class	201
C++ API / C API Equivalence.....	213
Supported Sub-Set of Behavioral Code	250
Advanced Properties	255
Mapping Between SystemVerilog and C Language Types	262
Supported SystemVerilog DPI Utility Functions.....	263
Back-to-Back Mode Functions	265
ZEMI-3 Options available in the UTF File	268
Log File Elements	272
zEmiRun Options.....	278
ZEMI3Xtor Class Methods.....	291
Additional Files to Modify Partitioning Results	313
Column Description in the Netlist Tree Pane (EDIF File)	316

Available Icons in the Netlist Tree Pane (EDIF File)	317
Netlist Tree Pane Column Description (Automatic Clustering Interconnection File).....	318
Available Options in the Timing Analysis Pane	326
Available Options in the Properties Window.....	329
Arguments Used with report_hardware	362
Optional Arguments Used with zTune -h	364
Supported Standard for Direct-ICE Interface	376
Range and Step Values for Delay Programming at Runtime	382
References for ERNI Connectors.....	389
Pinout for 12V Power Supply Connectors J122 and J123.....	390
Pinout for Global Clocks Connector (J121)	391
Pinout for VCCIO_2 Connector (J120)	391
Pinout for 12-pin Connectors.....	392
Generic Pinout Information for Data Connectors	394
Timing for Data Inputs	396
Timing for Data Outputs	397
Maximum Skew for Output Clocks	397
Global Commands (zice).....	399
Clock Commands (zice_clock)	399
Data Commands (zice_io)	400
Strictly Global Options (zice only)	401
Overridable Default Options (zice and zice_clock / zice_io).....	402
Strictly Clock and/or Data Options (zice_clock / zice_io)	404
Range and Step Values for Delay Programming at Runtime	409
Pin Allocation on a Smart Z-ICE Connector	463
Pin Allocation on Smart Z-ICE HE10 Adapter Connectors.....	466
ZS4 Smart Z-ICE HE10 Connector Pins for Data.....	483
ZS4 Smart Z-ICE HE10 Connector Pins for VCC and GND	484
Description of ports of SMART_ZICE_ZSE Module.....	485

About This Book

The ZeBu® Server User Guide provides a complete reference to the Synopsys emulator system, ZeBu Server. Using this guide, you can understand ZeBu Server, its features, its supported components, and emulate your design with ZeBu Server.

Contents of This Book

The ZeBu® Server User’s Guide has the following chapters:

Chapter	Describes...
ZeBu Server	Introduces ZeBu Server and its features.
Assembling a Design for Emulation	Describes how to prepare your design for emulation before compilation.
Compilation	Describes how to compile your design in ZeBu.
Clock Delay Support in Verilog for ZeBu	Describes how to use the Clock Delay feature to enable simulation style support of native clock generation in the design.
Runtime	Describes how to control ZeBu runtime.
RunManager	Describes how to control the emulation runtime to overcome the limitation of the C-COSIM and ZEBU::Clock object.
Using SystemVerilog Assertions	Describes SystemVerilog assertions for functional verification.
DPI Synthesis Support in a Design	Describes DPI imported function calls supported in ZeBu.
Power Aware Verification With ZeBu	Describes Power Aware Verification with ZeBu.
Compiling ZeBu Transactors	Introduces ZeBu transactors and how to compile them in Unified Compile flow.

Chapter	Describes...
<i>Custom DPI Based Transactors With ZEMI-3</i>	Describes ZEMI-3 that enables writing transactors for functional testing of a design.
<i>Hybrid Emulation</i>	Introduces Hybrid Emulation.
<i>Design Partitioning With zManualPartitioner</i>	Describes zManualPartitioner , a graphical interface to help with manual partitioning.
<i>Runtime Performance Analysis With zTune</i>	Describes emulation performance analysis using zTune .
<i>Saving the Design State and Restarting from a Saved State</i>	Describes the save and restore feature that allows you to save and restore the state of the entire hardware DUT and the software testbench environment during the emulation runtime.
<i>Direct - ICE for ZeBu Server 3</i>	Describes the usage of Direct-ICE.
<i>Smart Z-ICE Interface for ZeBu Server 3</i>	Describes the usage of Smart Z-ICE.
<i>Smart Z-ICE Support on ZeBu Server 4</i>	Describes the usage of Smart Z-ICE on ZeBu Server 4.
<i>Appendix</i>	Describes the input files for compilation and runtime operations specific to the ZeBu environment.
<i>zFmCheck</i>	Describes about the usage of zFmCheck .

Related Documentation

Document Name	Description
<i>ZeBu Server 4 Site Planning Guide</i>	Describes planning for ZeBu Server 4 hardware installation.
<i>ZeBu Server 3 Site Planning Guide</i>	Describes planning for ZeBu Server 3 hardware installation.
<i>ZeBu Server Site Administration Guide</i>	Provides information on administration tasks for ZeBu Server 3 and ZeBu Server 4. It includes software installation.
<i>ZeBu Server Getting Started Guide</i>	Provides brief information on using ZeBu Server.
<i>ZeBu Server User Guide</i>	Provides detailed information on using ZeBu Server.
<i>ZeBu Server Debug Guide</i>	Provides information on tools you can use for debugging.
<i>ZeBu Server Debug Methodology Guide</i>	Provides debug methodologies that you can use for debugging.
<i>ZeBu Server Unified Command-Line User Guide</i>	Provides the usage of Unified Command-Line Interface (UCLI) for debugging your design.
<i>ZeBu Server Functional Coverage User Guide</i>	Describes collecting functional coverage in emulation. For VCS and Verdi, see the following: <ul style="list-style-type: none">- Coverage Technology User Guide- Coverage Technology Reference Guide- Verification Planner User Guide- Verdi Coverage User Guide and Tutorial For SystemVerilog, see the following: <ul style="list-style-type: none">- SystemVerilog LRM (2017)
<i>ZeBu Server Power Estimation User Guide</i>	Provides the power estimation flow and the tools required to estimate the power on a System on a Chip (SoC) in emulation. For SpyGlass, see the following: <ul style="list-style-type: none">- SpyGlass Power Estimation and Rules Reference- SpyGlass Power Estimation Methodology Guide
<i>ZeBu Verdi Integration Guide</i>	Provides Verdi features that you can use with ZeBu. This document is available in the Verdi documentation set.
<i>ZeBu Server LCA Features Guide</i>	Provides a list of LCA features available with ZeBu Server.
<i>ZeBu Server Release Notes</i>	Provides enhancements and limitations for a specific release.

1 ZeBu Server

This section presents an overview and the features of the Synopsys emulator, ZeBu Server. This section describes the following ZeBu Server topics:

- *Overview Of ZeBu Server*
- *Hardware Overview*
- *ZeBu Emulation Flow*
- *Introduction to zCui*

1.1 Overview Of ZeBu Server

ZeBu Server is a very high-capacity emulator system with easy setup and debugging capabilities. It is available in two slot and five slot to handle designs up to 60– 300 million ASIC-equivalent gates. ZeBu Server allows you to connect two users in a two-slot system and up to five users in a five-slot system.

ZeBu Server is also expandable in a multi-unit environment to accommodate up to 49 users (by combining 10 units of a five-slot system) to handle designs up to three billion ASIC-equivalent gates.

Features

The salient features of ZeBu Server are as follows:

- ZeBu Server supports various software and hardware debugging modes. It can handle the most challenging verification problems that can occur in the systems during the design cycle.
- ZeBu Server is connected to a host PC through a Peripheral Component Interconnect (PCI) Express board. In multi-user environments, different hosts can be connected to each unit of ZeBu Server.
- ZeBu Server provides the following two additional interfaces for connecting a Design Under Test (DUT) to a software debugger or a target system:
 - The Direct In-Circuit Emulation (ICE) interface connects the DUT to a target system or a hardware core through 1,200 data pins and two dedicated clock pins.
 - The Smart Z-ICE interface (64 data pins and four clock pins for a two-slot system; 80 data pins and five clock pins on a five-slot system) provides support for standard software debuggers using JTAG cables.

1.2 Hardware Overview

As already mentioned, ZeBu Server is available in both two-slot and five-slot systems.

Note

A two-slot ZeBu Server is not intended to be part of a multi-unit environment.

The two-slot ZeBu Server consists of the following elements:

- Two slots to plug-in up to two FPGA modules.
- A rear panel equipped with the following components:
 - A PCIe interface: Four connectors to link up to four PCs.
 - The Smart Z-ICE interface: Four connectors with 64 data pins and four clock pins.

The following figure displays the front and rear panel of a two-slot ZeBu Server, respectively.



FIGURE 1. ZeBu Server-2: 2-slot Front and Rear Panels

A five-slot ZeBu Server can be a standalone or a part of a multi-unit environment. The five-slot ZeBu Server consists of the following elements:

- Five slots to plug-in up to five FPGA modules.
- A backplane equipped with the following components:
 - The PCIe interface (10 connectors to link up to five PCs, four units, and a hub).
 - The Smart Z-ICE interface (Five connectors with 80 data pins and five clock pins).

The following figure displays the front and rear panel of a five-slot ZeBu Server, respectively.



FIGURE 2. ZeBu Server-3: 5-slot Front and Rear Panels

For detailed information about the ZeBu Server hardware and software prerequisites for installing ZeBu Server, see *ZeBu Server Installation Manual*.

This section consists of the following sub-sections:

- *Design Capacity*
- *FPGAs in ZeBu Server*

1.2.1 Design Capacity

The following table lists the design capacity for each ZeBu Server unit.

TABLE 1 Design Capacity for ZeBu Server

FPGA Module	Design FPGAs	Design Capacity (ASIC Gates)	DDR3 Memory	ICE Pins
9F	9	60M	4x 512MB 2 X 8 GB	N/A
9F/ICE	9	60M	4x 512 MB 2 X 8 GB	568

1.2.2 FPGAs in ZeBu Server

ZeBu Server is a scalable system that maps a DUT into Xilinx Virtex-7 LX2000 FPGA.

Note

Interface FPGAs (IF) is available only in ZeBu Server 3.

The following table lists FPGA module types supported in ZeBu Server.

TABLE 2 FPGA Type and FPGA Module Types in ZeBu Server

FPGAs Type	FPGA Module	Description
Xilinx Virtex-7 LX2000	9F	<ul style="list-style-type: none"> ● Four FPGAs with 512 MB DDR3 memory for each FPGA ● One FPGA with 2x 8 GB DDR3 memory ● Four FPGAs with no additional memory ● One FPGA dedicated to the RTB
	9F/ICE	<ul style="list-style-type: none"> ● Four FPGAs with 512 MB DDR3 memory for each FPGA ● One FPGA with 2x 8 GB DDR3 memory ● Four FPGAs with no additional memory ● One FPGA dedicated to the RTB ● Direct In-Circuit Emulation (ICE) interface (568 data pins and dedicated clock pins) to connect a target system

The following figure displays the architecture of an FPGA module within ZeBu Server 3.

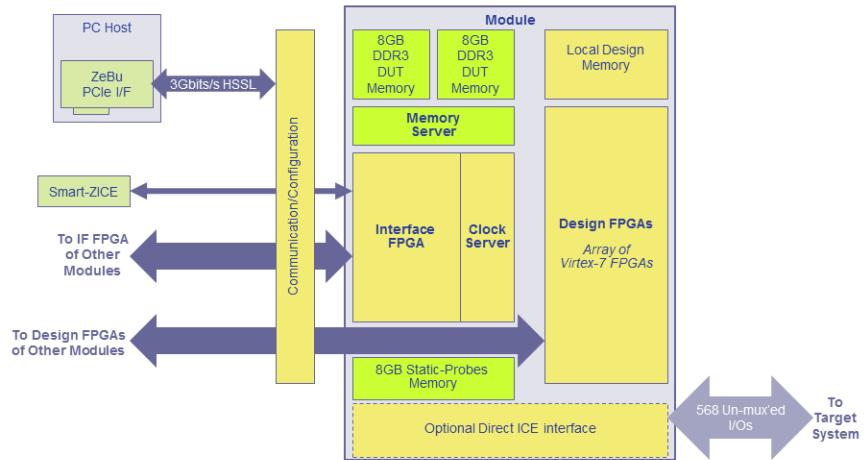


FIGURE 3. Architecture of an FPGA Module in ZeBu Server 3

The following figure displays the architecture of ZeBu Server 4.

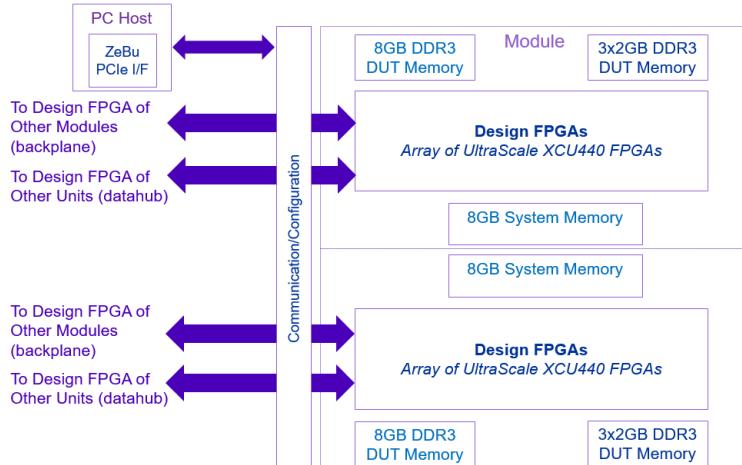


FIGURE 4. Architecture of ZeBu Server 4

1.3 ZeBu Emulation Flow

In ZeBu emulation, design files and project files are compiled using VCS.

The emulation provides the following files for runtime:

- FPGA bitstream files (Downloaded into the FPGAs)
- Runtime data (Used by the host computer)

The following figure displays the overall emulation flow in ZeBu.

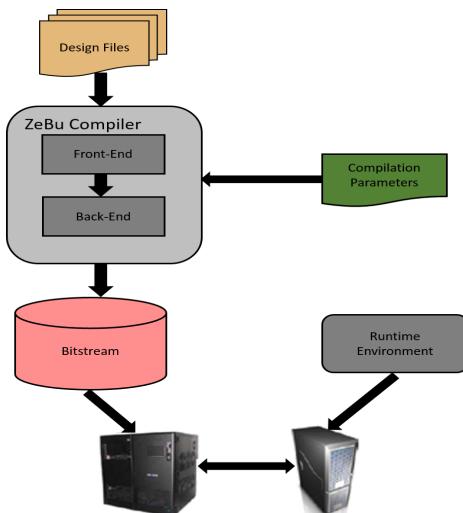


FIGURE 5. ZeBu Emulation Flow

The ZeBu emulation flow consists of the following steps:

- *Assembling a Design for Emulation*
- *Compilation*
- *Runtime*

1.4 Introduction to zCui

To compile a design, ZeBu offers a graphical interface, **zCui**, to configure the compiler and analyze the compilation results. The following figure displays the default view of **zCui**.

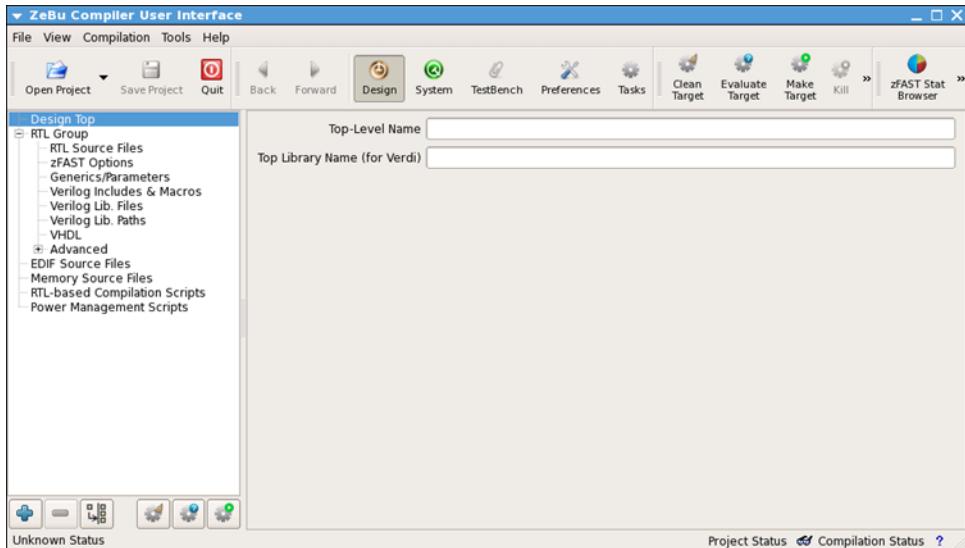


FIGURE 6. zCui Default View

The following operations are available in **zCui**:

- **Clean Target** (): Removes the resultant files of the previous compilation but retains their log files.
- **Evaluate Target** (): Checks the status of the compilation target.
- **Make Target** (): Starts the compilation.

For information about compiling a design using **zCui**, see [Compiling a Project](#).

— 2 Assembling a Design for Emulation

This section describes how to assemble your design for emulation before compilation. This section discusses the following topics.

- *Mapping Your Design and Test Environment*
- *Clock Modeling*
- *Reset Modeling*
- *Memory Modeling*
- *Designating Signals for Access During Runtime*
- *Verilog System Tasks*
- *Verilog Procedural Interface*

2.1 Mapping Your Design and Test Environment

To map your design and test environment to emulation, you must do one of the following depending on the design and the testbench:

- When a simulation environment is available for your design, you can retain the same architecture with the existing top-level module. For more information, see [Importing a Design From a Simulation Environment](#).
- When the DUT and testbench are available as separate elements, you need to create an additional top-level wrapper. For more information, see [Preparing a DUT-Only Environment](#).

2.1.1 Importing a Design From a Simulation Environment

To import a DUT and its test environment from a simulation environment, retain the existing top module and perform the following steps:

1. Instantiate clock-generation primitives to connect to DUT clocks.
2. Instantiate transactors that interact with the DUT.

The following figure displays a clock generator and transactor instantiated in a top module.

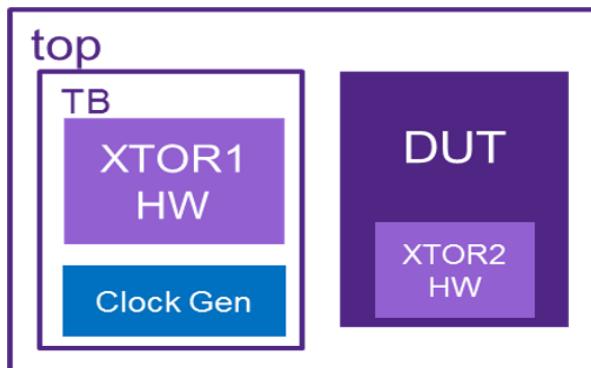


FIGURE 7. Importing a DUT and Test Environment From a Simulation Environment

2.1.2 Preparing a DUT-Only Environment

When the DUT and testbench are available as separate elements, you need to create an additional Verilog top-level wrapper that connects the design with your verification environment (testbench or transactors). Then, instantiate the clock-generation primitives in this wrapper.

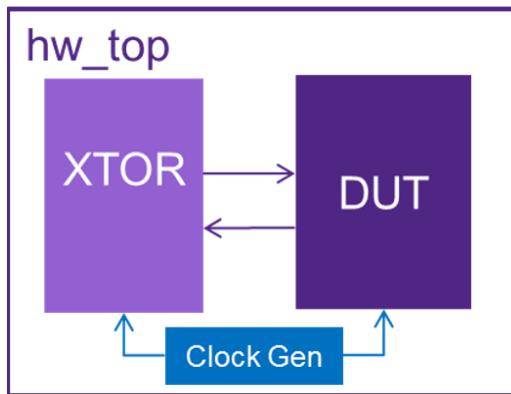


FIGURE 8. Preparing a DUT-Only Environment

2.2 Clock Modeling

Primary clocks are modeled with a dedicated clock generation primitive that can be instantiated anywhere in the HDL. You must instantiate at least one clock-generation primitive and can use up to 16 clock-generation primitives.

A clock generator is an instance of the `zceiClockPort` primitive.

For example,

```
zceiClockPort ClockPort (.cclock(clk));
```

where,

- `cclock`: A clock signal connected to `cclock` pin is called a controlled clock.

The runtime attributes of controlled clocks are accessible through the `designFeatures` file. For more details about the `designFeatures` file, see [Controlling Runtime Parameters](#).

2.3 Reset Modeling

The reset signal can also be directly programmed in the testbench environment by declaring the signal as controlled by software or forceable to accept a user-defined value at runtime.

For more information about forcing reset, see [Forcing and Injecting Values](#).

Based on your testbench, the reset signal can be provided as follows:

- If you are using **zRun** to control your testbench, **zRun** should control the reset signal.
- If you have a cycle-based testbench, the tesbench should provide the DUT reset signal.
- If you have a transaction-based testbench, the transactor should provide the DUT reset signal.

Note

Do not use the reset outputs of zceiClockPort.

2.4 Memory Modeling

ZeBu Server instantiates SRAM-type memory models that which are generated by the memory generator of the ZeBu compiler. These models apply to the memory of both design and testbench. You can model memory in the following ways:

Describe the memory array declaration in HDL and memory accesses in process blocks.

Instantiate the available memory model Intellectual Property (IP).

This section consists of the following sub-sections:

- [Inference From a Design](#)
- [Memory Model IP](#)

2.4.1 Inference From a Design

The ZeBu compilation handles Verilog/VHDL synthesizable memory that can be implemented in the following ways:

- Small-size memory (up to 1 KB) implemented using registers or Lookup Table (LUT) RAMs (also known as Distributed RAMs) (1KB - 10 KB) in Xilinx Virtex FPGAs.
- Medium-size memory (1 KB - 500 KB) implemented using Block RAMs (BRAM) in Xilinx Virtex FPGAs.
- Large-size memory (over 500 KB) implemented using on-board memories (called ZRM memories).

To change the implementation of these memories, update the thresholds using memories and `memory_preferences` Unified Tcl Format (UTF) commands.

2.4.1.1 Guidelines for Performance

Memory performance depends on memory ports inference.

Following are some of the coding style guidelines and examples:

- Multi-ports and number of instances

It is recommended to prevent inference of memory with high numbers of ports. Therefore, the code must be reviewed to understand the cause of the ports and change coding style.

The number of instances has a direct impact on the number of ports. As ZeBu has a limited number of physical memory banks, multiple instances may be mapped onto the same physical memory bank. However, it increases the number of ports on the physical memory.

For example, one 32-bit wide memory is usually better than four 8-bit wide memories.

- Read ports: Recommended coding style

- Continuously read ports

It is important to control the read with an enable to avoid continuous access. Otherwise, the design clocks are stopped on every memory active clock edge, thus impacting runtime performance.

- Asynchronous read ports

It is recommended to avoid asynchronous read ports when possible as they have a longer output delay, try to re-model with a synchronous port.

The following are some of the user code transformation examples:

Example 1:

Both reads and writes to the memories must happen inside sequential always blocks to read and write synchronous ports to the memories.

Original RTL

```
always @ (posedge clk)
begin
    read_address_reg      <= read_address;
end
assign read_data = mem[read_address_reg]; // Read to memory
outside clocking block.
```

Modified RTL

```
always @ (posedge clk)
begin
    read_data      <= mem[read_address];
end
```

Example 2:

The following code is implemented as an asynchronous read port if the read output is initialized within the reset block.

Original RTL

```
always @(posedge clk or negedge reset_n)
begin
    if (~reset_n) begin
        rd_data <= {((DIR_BITS+PAGE_BITS)+4){1'b0}};
    end
    else begin
        rd_data <= mem[address];
    end
end
```

Modified RTL

```
always @(posedge clk or negedge reset_n)
begin
    if (~reset_n) begin
        rd_reset <= 1'b1;
    end
    else begin
        rd_data_tmp <= mem[address];
        rd_reset <= 1'b0;
    end
    end
    assign rd_data = (rd_reset) ? {((DIR_BITS+PAGE_BITS)+4){1'b0}}
: rd_data_tmp;
```

- Write ports: Recommended coding style
 - Asynchronous write ports

Avoid when possible as they may be slower.

Read-modify-write ports

If the read occurs in an always block and the data modification happens outside the always block based on byte enable, the read and write are implemented as separate ports. If the read, modification, and write back take place in one always block, it is implemented as a single port. The following example can be used to implement read-modify-write using a single port memory.

```
always @ (posedge clk) begin
    wmask = 0;
    for (i=0; (i < SMEM_DW) ;i = (i+1)) begin //SMEM_DW-Data width
        in bits

        if(byten[(i >> 3)]) begin
            wmask[i] = 1'b1;
        end
    end
    if (wr) begin
        shmem[waddr] <=(shmem[waddr] & ~wmask) | (wdata & wmask);
    end
end
```

2.4.2 Memory Model IP

You can instantiate some specific memory implementations available as Memory IPs or transactors.

- Complex memory models (for example, DDRx/GDDRx SDRAM and NAND/NOR Flash) are available as separate IPs with dedicated documentation.
- Ultra-large memory, which exceeds the memory capacity of ZeBu Server, can use the memory of the host PC through one of the dedicated transactors (for example, SRAMSW and ZLPDDR4_Xtor).

Designating Signals for Access During Runtime

- Shared memory uses the memory of the host PC and it allows the software side to bypass the security and access the memory for better performance (for example, SHARED_SRAM).

For memory model IPs, the top-level wrapper is used to get access to the memory interface. In addition, you must use the `load_edif` command to load the EDIF description of the memory model.

For transactors, you must use the path defined in `$ZEBU_IP_ROOT` or use the `transactors` UFT command to specify the list of transactors and their respective paths.

2.5 Designating Signals for Access During Runtime

This section describes how signals can be accessed during runtime.

2.5.1 Reading Signals

It is possible to read any sequential signal at runtime using the Xilinx scan-chain feature and dynamic-probes.

The following UFT command allows the testbench to access any signal at runtime:

```
probe_signals -type dynamic -rtlname <net_declaration>
```

Runtime control for the signals is possible through the C/C++ API described in the `Board.hh` or `ZEBU_Board.h` header files.

2.5.2 Forcing and Injecting Values

Forcing a signal means to set a user-defined value at runtime until it is explicitly released.

Injecting a signal means to set a user-defined value at runtime. However, the value is overwritten by the value defined by the design during the next write operation.

To force or inject a value on a signal, use the following UFT commands:

```
zforce [options]  
zinject [options]
```

These commands can be used for any type of signal in the design, such as, undriven signals, combinational signals, or signals driven by registers and latches. Both commands support pattern matching declaration with the `-fnmatch` option

To view the UFT help commands, use the following commands:

```
vcs -help utf+zforce  
vcs -help utf+zinject
```

Runtime control of the signals designated by `zforce` and `zinject` commands is also possible using C/C++ APIs.

2.6 Verilog System Tasks

Verilog System tasks are used to generate input and output during simulation. ZeBu supports Verilog system-tasks present in the definition of a DUT and ZEMI-3 transactors.

For information about ZEMI-3, see [Custom DPI Based Transactors With ZEMI-3](#).

The following table lists the Verilog system-tasks supported in ZeBu.

TABLE 3 Supported Verilog System-Tasks

Task	Supported in ZEMI-3	Supported in Design	Default
Simulation Tasks			
\$finish (always block)	No	Yes	No
Simulation Time Functions			
\$realtime	No	No	-

TABLE 3 Supported Verilog System-Tasks

Task	Supported in ZEMI-3	Supported in Design	Default
\$time	Yes	Yes	No (clock_dela y)
Timescale Tasks			
\$timeformat	No	No	-
Conversion Functions			
\$bitstoreal, \$realtobits	No	No	-
\$signed, \$unsigned	Yes	Yes	Yes
\$cast (statically determined)	Partial	Partial	Yes
Array Query Functions			
\$unpacked_dimensions, \$dimensions, \$left, \$right, \$low, \$high, \$size, \$increment	Yes	Yes	Yes
Math Functions			
\$clog2, \$asi, \$ln, \$acos \$log10, \$atan, \$atan2, \$tanh \$exp, \$sqrt, \$hypot, \$pow \$sinh, \$floor, \$cosh, \$ceil \$sin, \$asinh, \$cos, \$acosh \$tan, \$atanh	No	No	-
Bit Vector System Functions			
\$countbits	No	No	-
\$countones, \$onehot, \$onehot0	Yes	Yes	Yes
\$isunkown (in SVA)	Partial	Partial	-
Severity Tasks			
\$fatal, \$error, \$warning, \$info	No	No	-
Elaboration Tasks			
\$fatal, \$error, \$warning, \$info	Yes	Yes	Yes

TABLE 3 Supported Verilog System-Tasks

Task	Supported in ZEMI-3	Supported in Design	Default
Assertion Control Tasks			
\$asserton, \$assertoff, \$assertkill	Yes	Yes	Yes
Sampled Value System Functions			
\$rose, \$fell, \$stable, \$changed, \$past	Yes	Yes	Yes
Coverage Control Functions			
\$coverage_control, \$coverage_get_ max, \$coverage_get, \$coverage_merge, \$coverage_save, \$get_coverage, \$set_coverage_db_name, \$load_coverage_db	No	No	-
Probabilistic Distribution Functions			
\$random(\$memset)	Partial	Partial	-
\$dist_chi_square, \$dist_erlang, \$dist_t, \$dist_exponential, \$dist_normal, \$dist_poisson, \$dist_uniform	No	No	-
Stochastic Analysis Tasks and Functions			
\$q_initialize, \$q_add \$q_full, \$q_exam, \$q_remove	No	No	-
PLA Modeling Tasks			

TABLE 3 Supported Verilog System-Tasks

Task	Supported in ZEMI-3	Supported in Design	Default
\$async\$and\$array, \$async\$and\$plane, \$async\$nand\$array, \$async\$or\$array, \$async\$nand\$plane, \$sync\$or\$array, \$async\$or\$plane, \$async\$nor\$array, \$async\$nor\$plane, \$sync\$and\$array, \$sync\$and\$plane, \$sync\$nand\$array, \$sync\$nand\$plane, \$sync\$or\$plane, \$sync\$nor\$array, \$sync\$nor\$plane	No	No	-
Miscellaneous Tasks and Functions			
\$system	No	No	-
Display Tasks			
\$display, \$displayb, \$displayh, \$displayo, \$write, \$writeb, \$writeh, \$writeo	Yes	Yes	No
\$strobe, \$monitorh	No	No	-
\$monitor (initial block)	No	Yes	Yes
File I/O Tasks and Functions			
\$fopen, \$fclose, fdisplay, \$fdisplayb, \$fdisplayh, \$fdisplayo, \$fwrite, \$fwriteb, \$fwriteh, \$fwriteo	Yes	Yes	No
\$sformat	No	No	-
Memory Load Tasks			
\$readmemb, \$readmemh	Yes	Yes	Yes

TABLE 3 Supported Verilog System-Tasks

Task	Supported in ZEMI-3	Supported in Design	Default
Memory Dump Tasks			
\$writememb, \$writememh	No	No	-
\$memset	Yes	Yes	Yes
Command Line Input			
\$value\$plusargs (always block), \$test\$plusargs(always block)	No	Yes	No
VCD Tasks			
\$dumpvars, \$dumports	Partial	Partial	-
\$dumpfile, \$dumpoff, \$dumpon, \$dumpall, \$dumplimit, \$dumpflash, \$dumpportsoff, \$dumpportson, \$dumpportsall, \$dumpportslimit, \$dumpportsflush	No	No	-

The following example displays how to use these system tasks:

```
initial $readmemh("mem1_init_content.txt",top.dut.mem1);

always @(error_detected)
  if(error_detected)
    $display("[ERROR] Error #%0d detected",error_code);
```

If the synthesis of these tasks is not enabled by default, the following UFT commands must be added:

```
dpi_synthesis -enable ALL
system_tasks -task "\$<task>" -enable
```

Verilog System Tasks

Some of the Verilog System tasks can be enabled by other UFT command as indicated in the preceding table.

The \$display task is very helpful for error handling. However, new verification methodologies are more assertion based. For more information about assertions, see [Using SystemVerilog Assertions](#).

The zDPI support must be enabled at runtime when emulating \$display tasks. To enable zDPI support, use the following:

```
Board *z = Board::open("zcui.work/zebu.work");
// ... other stuff
CCall::Start(z);
```

For details, see [DPI Synthesis Support in a Design](#).

2.6.1 \$memset () System Task

The Verilog System task, \$memset (), can be used to fill the memory bits (all or within an address range) with binary values or random 0/1. The randomization is specified using the system function call \$random, which is a placeholder. The random memory content is defined during compilation and remains static at runtime.

The following is an example of how to use the \$memset () task.

```
always begin
if(power_off_condition) begin
    $memset(0, uBLK0.mem0);
    $memset(0, uBLK0.mem1);
    ...
    $memset(0, uBLK23.mem386);
end
and
```

```
always begin
  if(power_off_condition) begin
    $memset($random, uBLK0.mem0);
    $memset($random, uBLK0.mem1);
    ...
    $memset($random, uBLK23.mem386);
  end
```

Note

The random values assigned for a compilation cannot be modified; successive runs do not change the randomized content of memories set by calling \$memset.

You can verify the vcs_simu_memset_*.mem files generated in the VCS dump directory for emulation flow. The \$display task can be used with the memory name to check that a certain binary value was set. The waveform can also display memory values.

2.6.2 \$plusargs System Task

The \$test\$plusargs and \$value\$plusargs Verilog system tasks search the list of \$plusargs for a user-specified plusarg_string. The string is specified in the argument to the system function as a string or an integral variable that is interpreted as a string.

For more information about the \$test\$plusargs and \$value\$plusargs tasks, see the IEEE Standard Language Reference Manual (1800-2012 section 21.6).

You must specify the following line in the designFeatures file to load values into \$plusargs variables during runtime:

```
$plusargs_file = <filename>;
```

where <filename> is the name of the file containing the variables (filename can be a full path to the file or a relative path).

2.6.2.1 Example

```
% cat designFeatures  
$plusargs_file = "../plusargs_values.txt";  
% cat designFeatures  
$plusargs_file = "/work/uc_plusargs/rt_values.txt";
```

The format of plusargs_file is as follows:

- +<VAR_NAME>=<VALUE>: for variable inside \$value\$plusargs
- +<VAR_NAME>: for variable inside \$test\$plusargs

The strings in plusargs_file are separated by a space.

- <VAR_NAME> is a variable name, specified in the first argument of \$plusargs.
- <VALUE> is a loaded value for this variable. It should be an integer or environment variable.

If a string in plusargs_file has an incorrect format, an error is displayed and line is skipped.

2.6.2.2 Limitations of \$plusargs

- Calls to \$value\$plusargs/\$test\$plusargs are supported only at DUT, not in transactors.
- Second argument of \$value\$plusargs must be an int/integer. Other types are currently not supported.

Example

```
$value$plusargs("count=%d", count) // supported case  
$value$plusargs("filename=%s", filename) // unsupported case
```

- First argument of \$value\$plusargs/\$test\$plusargs must be literal constant string. Other types are not supported.

Example

```
$test$plusargs("enable_diag") // supported case  
string en = "enable_diag";  
$test$plusargs(en) //unsupported case  
  
$value$plusargs("count=%d", count) // supported case  
string str = "count=%d";  
$value$plusargs(str, count) //unsupported case
```

For the unsupported cases, a warning message, “ZEBUUC-PLUSARGS-*”, is displayed and the functions calls are skipped.

2.6.3 \$fopen and \$fclose System Tasks

The Verilog system tasks \$fopen and \$fclose are supported in both DUT and ZEMI-3 transactors.

For more information about the \$fopen and \$fclose tasks, see the IEEE Standard Language Reference Manual (1800-2012 section 21.3.1).

To enable the support of \$fopen and \$fclose, the following commands must be added to the UTF file:

```
system_tasks -task "\$fopen" -enable  
system_tasks -task "\$fclose" -enable
```

2.6.3.1 Example

```

module dut(input clk, output reg [31:0] counter);
    integer dut_fd;
    bit      isopen = 0;
    always @ (posedge clk) begin
        counter <= counter+1;
    end
    always @ (posedge clk)
    begin
        if (counter == 1) begin
            dut_fd = $fopen("dut_file.log", "w");
            isopen = 1;
        end
        if (counter == 100) begin
            isopen = 0;
            $fclose(dut_fd);
        end
        if (isopen) begin
            $fdisplay(dut_fd, "DUT_COUNTER %d\n", counter);
        end
    end
endmodule // dut

```

2.6.3.2 Limitations of \$fopen and \$fclose tasks

\$fopen and \$fclose tasks support the following:

- The arguments of \$fopen (file name and mode) must be constant strings. The following table lists examples for supported and unsupported scenarios.

Supported Format	Unsupported Format
<pre>int fd; fd = \$fopen("my.log", "w")</pre>	<pre>wire[8*80-1:0] file_name; assign file_name = "my.log"; fd = \$fopen(file_name, "r");</pre>

- `$fopen/$fclose` are supported only inside always block.
- `$fopen/$fclose` are supported in initial blocks by enabling `clock_delay` in UFT and add "#0" inside the initial block.

2.6.4 \$finish System Task

The Verilog system task `$finish` is supported inside the DUT and can be used to exit the emulation run. To exit emulation through `$finish`, a runtime API call with a function name as argument is required. The content of this function must be defined in a Tcl script and have no arguments. This is called when `$finish` is invoked, and can be used for a user callback just before exiting emulation run.

The `$finish` task is only supported in the following scenarios:

- Calls to `$finish` must appear only in the DUT.
- `$finish` is supported only with `zRun` standalone mode.
- `$finish` is always supported only inside blocks.

The callback function is provided using the command
`"ZEBU_Finish_SetCallback <function_name>"`.

Note

There can be only one Tcl function inside the DUT and the function definition must appear before calling the function.

2.6.4.1 Example

```
% cat designFeatures
$nbProcess = 0;
zRun command:
zRun -zebu.work zcui.work/zebu.work -do run.tcl -nogui -
debugFullMode
% cat run.tcl
proc do_finish {} {
    global zfinished
    puts "#### Finish called ####"
    set zfinished 1
    ZEBU_CCall_flush
    ZEBU_CCall_stop
    ZEBU_close

}
ZEBU_Finish_setCallback do_finish
```

2.7 Verilog Procedural Interface

vpi_get_time() is supported with clocks as follows:

- When clock delay support is used, it returns the simulation time at the precision derived from settings for the design clocks.
- When ZCEI clocks are used, it returns the tick count (every posedge of driverClk on which a design clock changes value).

3 Compilation

This chapter describes how to compile your design in ZeBu.

It discusses the following topics:

- *Overall Unified Compile Flow*
- *Word-Level Synthesis*
- *Compilation Script for VCS (UUM)*
- *Specifying the Hardware Architecture for Compilation*
- *Grid/LSF Handy Commands*
- *Gate-Level Netlist Compilation*
- *Compiling a Project*
- *Important Log Files*

3.1 Overall Unified Compile Flow

With the Unified Compile flow, the design and its test environment are compiled together by the VCS™ compiler, which is directed by a ZeBu project file written in Unified Tcl Format (UTF) file.

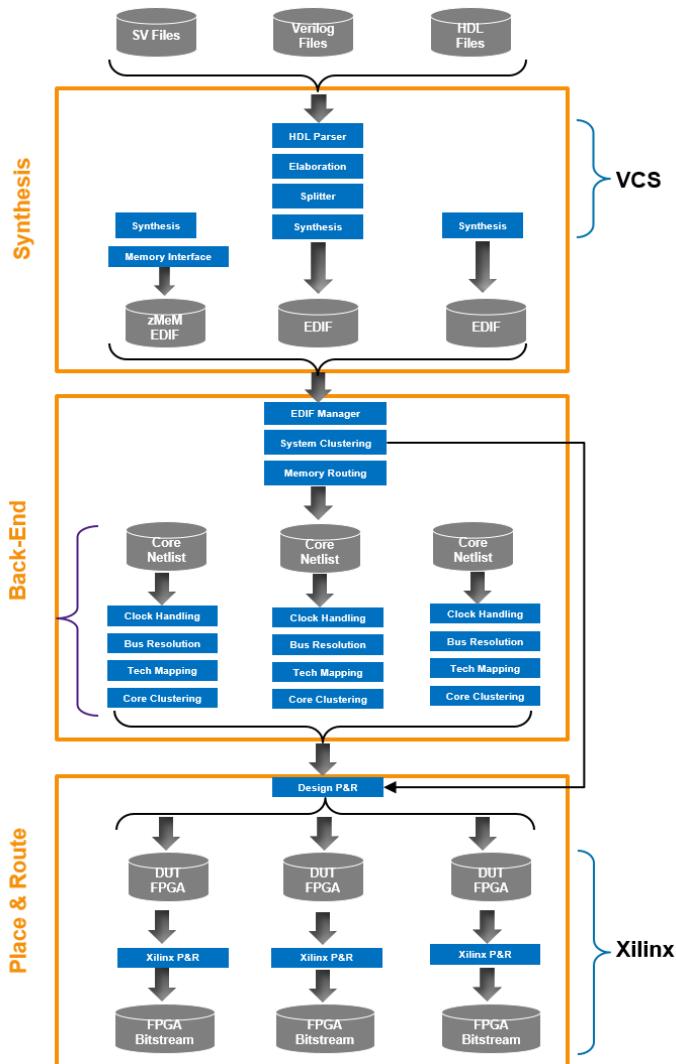
The UTF project file (for example, `project.utf`) is the main input file to the ZeBu compiler. It contains all the information required for successful compilation, including the VCS compilation command and the ZeBu back-end compilation.

When compiling a project file, VCS parses the HDL and elaborates the design. VCS compilation scripts can be reused from an existing simulation environment to reduce the design bring-up time.

The entire compilation flow is controlled by **zCui**, a Graphical User Interface (GUI), for ZeBu compilation. **zCui** launches VCS for front-end synthesis, followed by back-end compilation, and Xilinx FPGA Place and Route.

The following figure illustrates the ZeBu compilation flow.

Overall Unified Compile Flow

**FIGURE 9.** ZeBu Unified Compile Flow

This section consists of the following sub-sections:

- *Introduction to UTF*
- *Basic UTF Commands*

3.1.1 Introduction to UTF

The UTF file is a single project file that provides all inputs required by the ZeBu compiler. It contains Tool Command Language (Tcl) commands that control the compilation.

Various categories of commands co-exist in the UTF file. Some of these categories are mandatory based on the overall emulation strategy.

The following table describes the UTF command categories.

To view the UTF help commands, use the commands listed in the following table.

TABLE 4 UTF Help Commands

Commands	Description
vcs -help utf+all	Displays the list of commands.
vcs -help utf+<command>	Displays the detailed help for <command>.
vcs -help utf+*	Displays the detailed help for all commands.

3.1.2 Basic UTF Commands

The following table lists some of the basic UTF commands.

TABLE 5 Basic UTF Commands

Commands	Description
architecture_file -filename {<path_to_zse_configuration.tcl> }	Specifies the hardware architecture file.
grid_task_association -task {<zebu_task_name>} -queue {<given_queue_name>} grid_cmd -queue {<given_queue_name>} -submit {<submit_command>} -njobs <int>	Sets up the job queuing system.

TABLE 5 Basic UFT Commands

Commands	Description
vcs_exec_command{<script_name_with_path>}	Specifies the VCS command for a design.
set_hwtop -module <design_top_module_name>	Specifies the top level of a design.

3.2 Word-Level Synthesis

For ZS3, Word-Level Synthesis (WLS) is enabled by default. WLS is the new synthesis tool implemented by VCS™ compiler. The WLS supports the following features:

- Formal model construction
 - Fully-synthesized hierarchical design view of the HDL
 - Infers Memories, Registers, and Latches
- Word-level model
 - Faster and less memory footprint than bit-level synthesis
- Built into VCS
- Supports constructs such as XMRs, SVA, UDP, and so on
- Supports multiple output formats such as SCM, HNL, DFG, SV, and so on
- Handles Verilog/SV and VHDL in a unified manner

The following are the benefits of using Word-Level Synthesis for compilation:

- Improves the front-end compilation performance
 - Faster and lower memory foot-print
 - Global optimization at word-level
 - Disk space savings in front-end compilation
 - Weight-based multi-processing
 - Efficient handling of gate-level netlists
- Enhances the quality of the compilation
 - Symbolic methods for state inference
 - Better memory inference/synthesis

- Improved SVA compiler
- Better integration with Formality™
- Eliminates the dependency on third-party software
- Provides support for VCS congruency and full design optimization at word-level data model

3.2.1 WLS Logs

The following table lists the various logs created by WLS.

TABLE 6 WLS Logs

WLS Stages	Logs
VCS simulation & WLS synthesis	See “vcs_splitter_VCS_Task_Builder.log” File can be found in: zcui.work/zCui/ log/ vcs_splitter_VCS_Task_Builder.log
WLS synthesis options used during the compilation are captured and can be leveraged during failure analysis.	See “simon.xml”
Each Synthesis bundle generate separate log file with prefix “Bundle_” & suffix equivalent to bundle number	See Log file for “0” bundle File can be found in: zcui.work/design/ synth_Default RTL_Group/ Bundle_0.log
Bundle script provides details about input files part of a bundle such as primary modules, library files & black boxes	See Bundle scripts

3.3 Compilation Script for VCS (UUM)

This section describes the steps required to compile the emulation environment (DUT and transactors hardware part) using the VCS Unified Use Model (UUM). The following are the compilation steps:

- [*Setting Up VCS*](#)
- [*Setting Up the synopsys_sim.setup File*](#)
- [*Setting Up Compilation Commands Using UTF*](#)
- [*Compilation Settings for DesignWare Blocks*](#)

3.3.1 Setting Up VCS

Perform the following steps to set up VCS:

1. Point the `$VCS_HOME` environment variable to the VCS installation path. Also, the `$PATH` environment variable should contain `$VCS_HOME/bin`.
2. Set the license file using one of the two environment variables `$LM_LICENSE_FILE` or `$SNPSLMD_LICENSE_FILE`.
3. Use the following VCS command to check VCS setup, version, and the platform:

```
% vcs -full64 -id
```

3.3.2 Setting Up the `synopsys_sim.setup` File

In case of multiple designs or VHDL/VHDL-MX designs, a `synopsys_sim.setup` file must be used. The `synopsys_sim.setup` file maps logical library names (using `-work` option in analyze commands) into the physical library directory (the UNIX directory for the analyzed content of respective logical library).

VCS looks for the `synopsys_sim.setup` file at the following locations (from the highest to the lowest precedence):

1. `% setenv SYNOPSYS_SIM_SETUP <file_path>`
2. `synopsys_sim.setup` in current directory
3. `synopsys_sim.setup` in the home directory
4. Installation directory (`$VCS_HOME/bin/synopsys_sim.setup`)

3.3.3 Setting Up Compilation Commands Using UUF

A compilation script (or command line) must be provided to the UUF command `vcs_exec_command` for parsing and elaborating a design.

The script should consist of the following two commands:

- Analyze commands for parsing HDL files (`vlogan` for Verilog/SystemVerilog and `vhdlan` for VHDL), for example:

```
% vlogan -full64 [vlogan_opts] file1.v file2.v -work IP1  
% vlogan -full64 [vlogan_opts] -f filelist.f -work IP2  
% vhdlan -full64 [vhdlan_opts] file3.vhd -work my_VH_lib  
% vhdlan -full64 [vhdlan_opts] file4.vhd file5.vhd
```

When there are no dependencies across commands, multiple analyze commands can be invoked to reduce the compile time.

- An elaboration command (`vcs`) for building the design hierarchy from the library files generated during the analysis stage:

```
% vcs -full64 [elab_opts] [libname.]top_unit
```

where,

- `top_unit`: Specifies the top module name or the top-level `v2k` configuration.
- `libname`: Specifies the library name of the top-level module and configuration (optional, if not specified the top-level module and configuration is searched in the `synopsys_sim.setup` file as per the given order).

For more information about VCS-MX setup and compilation commands, see options in the VCS Documentation using the following command:

```
% vcs -full64 -doc
```

Or

Access the *VCS MX/VCS MXi User Guide* from SolvNet by performing the following steps:

1. Log in to the SolvNet online support site using your SolvNet account (<https://solvnet.synopsys.com/>).

2. Click the **Documentation** tab and select **VCS®** or **VCS®MX**.

3.3.4 Compilation Settings for DesignWare Blocks

To set the compilation settings for DesignWare Building Blocks (DWBB), perform the following steps:

1. Set the \$SYNOPSYS environment variable to the Synopsys synthesis (DesignCompiler) installation.
2. Map the logical libraries to appropriate files in the synopsys_sim.setup file.
3. In your VCS script, add the commands for the following steps:
 - a. Create the work directories for each library, as shown in the following example:

```
mkdir dware
```
 - b. Add the analysis commands (vhdlan/vlogan) for the DWBB component source files (encrypted VHDL and Verilog files), as shown in the following example:

```
vhdlan -full164 -work dware $SYNOPSYS/dw/fpga_ip/fv/  
dw_foundation/dware_comp.vhd.e
```

```
vhdlan -full164 -work dware $SYNOPSYS/dw/fpga_ip/fv/  
dw_foundation/dware.vhd.e
```

If your design targets low power, use specific min-power libraries.

For details on examples, see the [article](#) on SolvNetPlus.

3.4 Specifying the Hardware Architecture for Compilation

The hardware architecture file, `zse_configuration.tcl`, is generated by the ZeBu installation process. This file identifies the ZeBu hardware architecture and is available in the `$ZEBU_SYSTEM_DIR/config/` directory.

For example,

```
architecure_file -filename <path> {/u/tools/ZEBU_SYSTEM/  
zs3_config/zse_configuration.tcl}
```

3.5 Grid/LSF Handy Commands

When compiling on a compute farm, it is recommended to use different remote commands through an external task scheduler such as Sun™ Grid Engine or Platform LSF® to do load balancing of the farm. If no remote command is declared, the ZeBu compiler is launched on the system from which **zCui** is launched. The following table lists recommendations for Sun Grid Engine and Platform LSF:

TABLE 7 Sun Grid Engine and Platform LSF Recommendation

Sun Grid Engine/Platform	Typical Remote Command	Typical Kill Command
Sun Grid Engine	qrsh	Qdel
Platform LSF with blocking jobs and a non-interactive queue	bsub -K	Bkill
Platform LSF with blocking jobs and an interactive queue	bsub -Is	Bkill

Additional options may be necessary for these commands to match the configuration constraints, in particular to target only compilation hosts with 64-bit operating systems and to manage priorities.

In the UTF script, the following command is used to setup the job queuing system:

```
grid_task_association -task {<zebu_task_name>} -queue
{<given_queue_name>}
grid_cmd -queue {<given_queue_name>} -submit {<submit_command>} -
njobs <int>
```

The `njobs` (number of jobs) parameter must be set to an appropriate value that takes into account:

- The number of processors in the farm (which is the maximum number of jobs for efficiency purposes).
- The acceptable load on the computer that runs the load sharing tool.

The `<given_queue_name>` can be:

- Zebu

Gate-Level Netlist Compilation

- ZebuAlternateSynthesis
- ZebuHeavy
- ZebuIse
- ZebuLight
- ZebuRelaunchedIse
- ZebuRelaunchedIseLevel2
- ZebuSuperHeavy
- ZebuSynthesis

For example,

- To set the default queue (minimal setting to compile on a farm), use the following command:

```
grid_cmd -queue {DEFAULT_QUEUE} -submit {<submit_command>} -jobs 0
```

Typical <submit_command> with qrsh is as follows:

```
qrsh -P zebu_compile -cwd -V -now no -verbose
```

3.6 Gate-Level Netlist Compilation

ZeBu supports gate level design synthesis. In large designs, the hierarchical level is long and it may impact design performance. To minimize the hierarchical level, inlining can be set for the modules that adhere to the following conditions during compilation:

- You can specify a list of modules to be inlined or inlining limit
- Modules can be automatically detected for inlining if:
 - Modules declared between `celldefine` and `endcelldefine`
 - Modules loaded from a library with `-y` or `-v` options for VCS

Inlining commands are available as part of the optimization `UTF` command as follows:

- `-inline <module_list>`: Performs module inlining for a list of modules. Also, it can also fetch pattern as follows:
 - * matches everything

- ? matches any single character
- [seq] matches any character in seq
- -inline_disable <module_list>: Disables inline on list of modules
- -auto_inline_limit <limit>: Auto inlines all modules with cost <= limit
 - The cost of module is cost of its instances and the number of gates/continuous assign/UDPs/regs
- -auto_inline_cost_refine <bool>: Sets optimization
 - NOTE:** *It does not add simple buffers towards inlining cost.*
- -auto_inline_params <seq_modules>=<bool>: Enables/disables auto inling parameters.
 - seq_modules enables inlining of sequential blocks
 - Default value is false.

For more options on inlining, use vcs -full64 -help utf+optimization.

3.7 Compiling a Project

zCui controls the entire compilation flow. It launches the necessary tools for compilation. **zCui** supports the following modes:

- Batch mode
- GUI mode

3.7.1 Compiling With **zCui** in the Batch Mode

To run **zCui** in the batch mode with an existing UTF project file, use the following command:

```
zCui -c -n -u <project_name>.utf [-w <zCui_uc_work_dir>]
```

where,

- **-c** launches compilation.
- **-n** runs **zCui** in the batch mode (no GUI).
- **<project_name>.utf** is the existing UTF project file containing information to compile a design for ZeBu using Unified Compile flow.
- **<zCui_uc_work_dir>** is the optional working directory for compilation; by default, this is **./zCui.work**.

After compilation, you can explore the compilation status with **zBatchExplorer**, a graphical tool similar to the Tasks workspace in **zCui** GUI. For more information, see [Analyzing Compilation Results Using zBatchExplorer](#).

3.7.2 Compiling With zCui in the GUI Mode

When running **zCui** in the GUI mode, launch compilation from the **Compilation** menu or from the corresponding toolbar as shown in the following figure:

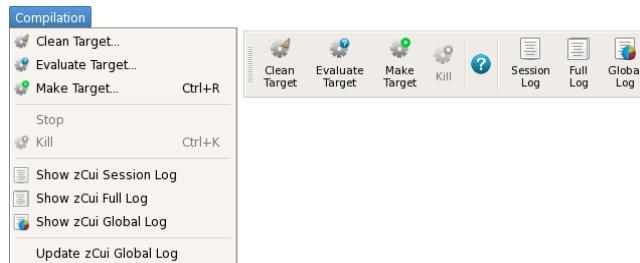


FIGURE 10. zCui Options

The following operations are available in the GUI mode:

- **Clean Target** (): Removes the resultant files of the previous compilation but retains their log files.
- **Evaluate Target** (): Checks the status of the compilation target (requires the same write access to compile a design).
- **Make Target** (): Starts the compilation of the target (launches only the tasks for which some dependencies are modified).

You can use the following command to launch **zCui** to monitor and manage the compile process with an existing UTF project file:

```
zCui -u <project_name>.utf [-w <zgui_uc_work_dir>]
```

where, `<project_name>.utf` is the UTF project file containing information to compile the design for ZeBu.

While using **zCui** to compile using a UTF file, it is not possible to save the compilation settings in the UTF file when they are modified in the GUI. Such modifications are only applicable for the next compilation within the GUI, for test purposes. Modification of the UTF file must be done in the original UTF project file.

Compiling a Project

The GUI displays the **Design** workspace, with only the **Unified Compile Flow Entry** panel, which displays the VCS command line from your UTF file as displayed in the following figure:

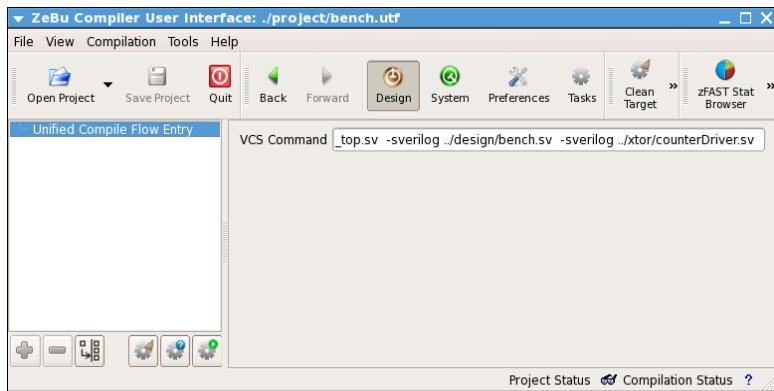


FIGURE 11. zCui GUI

If any issues are found while checking the content of your UTF file, **zCui** opens with the **Report Message** panel (**Preferences** workspace). You can also view this panel when the UTF file is loaded correctly in **zCui**, see the following figure:

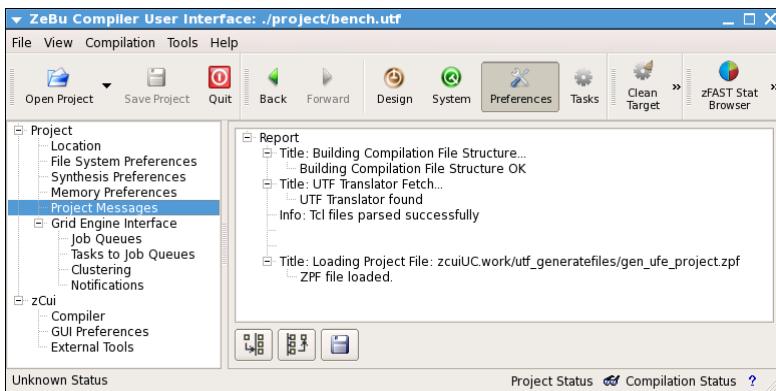


FIGURE 12. Project Messages Panel

If you modify some compilation settings in **zCui**, the modified settings are applied on the next compilation.

Launch your compilation using **Make Target** as displayed in the following figure:



FIGURE 13. Launching Compilation in zCui

In the **Compilation Chooser** dialog (see *Figure 13*), you must select one of the following:

- If you select **Design**, **zCui** launches front-end compilation.
- If you select **Default** in **Backends**, **zCui** launches all the compilation tasks, in both front-end and back-end compilation, if necessary.

During compilation, VCS is listed as a separate task in the **Tasks** workspace as displayed in the following figure:

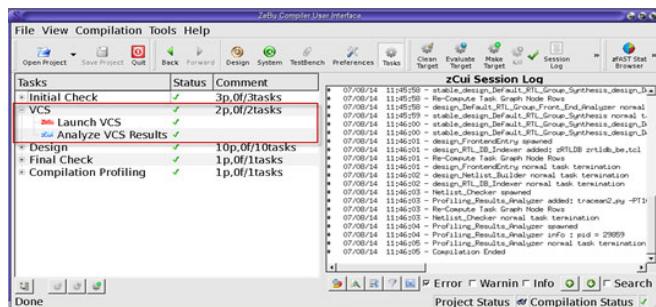


FIGURE 14. Viewing the VCS Task in zCui

3.7.2.1 Verdi Database Tasks in zCui

zCui controls construction of the Verdi database in parallel to synthesis. Therefore, the time consumed by Verdi's database construction is not on the critical path.

In the **zCui** compilation log, the following new tasks are visible:

- **Prepare Verdi Compilation:** It is internal to **zCui** and is very short. It prepares the environment for launching the database construction.
- **Launch Verdi:** It constructs the Verdi database in batch mode.

The Verdi task is registered in **zCui** with the "**Verdi_Compilation**" task class. By default, it is associated to the job queue "**ZebuSuperHeavy**" because it must access the entire design. If you want to change the job queue in UTF, you must add the following command:

```
grid_task_association -task {Verdi_Compilation} -queue {newQueue}
```

where, **newQueue** is another existing job queue or a new job queue created by
`grid_cmd -queue {newQueue} -submit {...} -delete {...} -njobs ...`

A new row is added in the **Tasks to Job Queue** panel and another row is added in the **Job Queue** panel for Verdi.

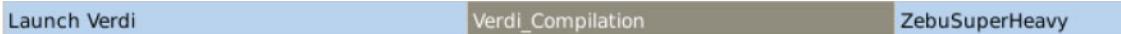


FIGURE 15. Tasks to Job Queue Panel

During compilation, Verdi is listed as a separate task in the **Tasks workspace** as displayed in the following figure:

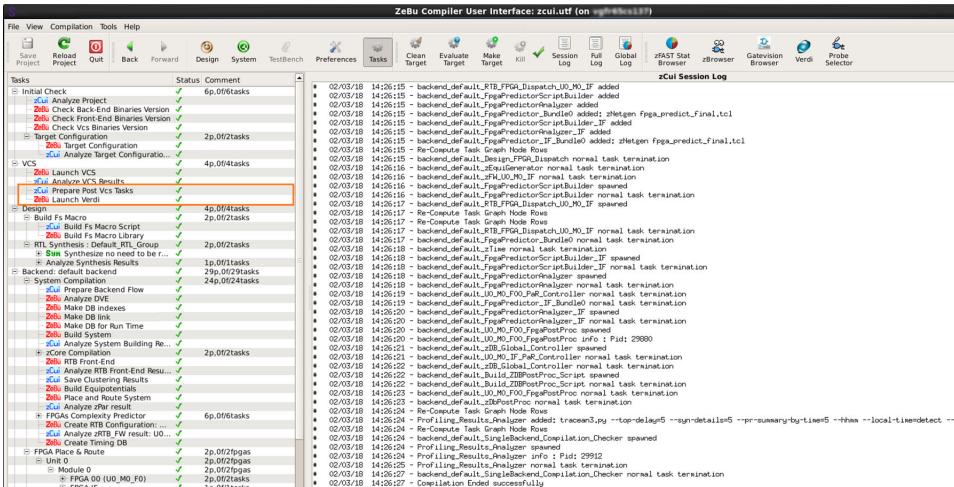


FIGURE 16. Viewing the Verdi Task in zCui

The following figure shows the Verdi tasks in the **zCui** log.

```
# 27/02/18 09:00:23 - vcs_splitter_VCS_Task_Builder spawned
# 27/02/18 09:00:23 - vcs_splitter_VCS_Task_Builder info : Pid: 15245
# 27/02/18 09:00:27 - vcs_splitter_VCS_Task_Builder normal task termination
# 27/02/18 09:00:27 - Verdi_Task_Builder spawned
# 27/02/18 09:00:27 - Verdi_Compilation added: # 27/02/18 09:00:27 - Verdi_Task_Builder
normal task termination File vcs_splitter/kdb_postelab.csh is found. Start Verdi compilation
# 27/02/18 09:00:27 - Verdi_Task_Builder info : File vcs_splitter/kdb_postelab.csh is found.
Start Verdi compilation
# 27/02/18 09:00:27 - VCS_Task_Analyzer spawned
# 27/02/18 09:00:27 - design_Default_RTL_GroupBundle_0_Synthesis added: zFe compile_hdr.tcl
script/Bundle_0_synp.tcl -log Bundle_0.log -zlog 1
# 27/02/18 09:00:27 - design_Default_RTL_GroupBundle_0_Synthesis_Bundle_0_analyzer added
# 27/02/18 09:00:27 - Re-Compute Task Graph Node Rows
# 27/02/18 09:00:27 - VCS_Task_Analyzer normal task termination
# 27/02/18 09:00:28 - Verdi_Compilation spawned
# 27/02/18 09:00:28 - Verdi_Compilation info : Pid: 15261
# 27/02/18 09:00:28 - design_Default_RTL_GroupBundle_0_Synthesis spawned
# 27/02/18 09:00:28 - design_Default_RTL_GroupBundle_0_Synthesis info : Pid: 15262
# 27/02/18 09:00:30 - Verdi_Compilation normal_task_termination
# 27/02/18 09:00:33 - design_Default_RTL_GroupBundle_0_Synthesis normal task termination
```

FIGURE 17. Viewing the Verdi Task in zCui Log

3.8 Important Log Files

This section consists of the following sub-sections:

- *Analyzing Compilation Results Using zBatchExplorer*
- *zTime*
- *Gathering Information of a Compilation Database*

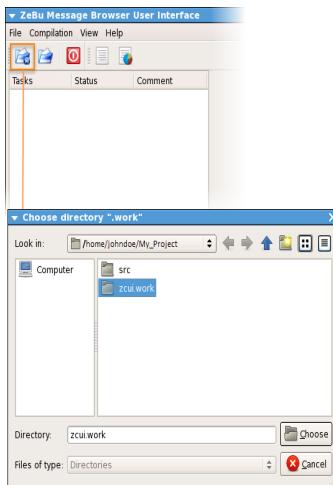
3.8.1 Analyzing Compilation Results Using zBatchExplorer

zBatchExplorer is a GUI that displays the task tree (similar to the **Tasks** workspace in **zCui**) after compilation. It analyzes log files of all the tasks, **zCui** full log, and global log.

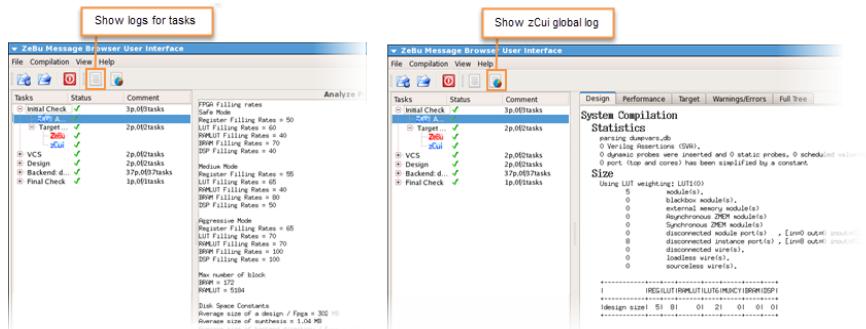
To launch **zBatchExplorer**, use the following command:

```
$ zBatchExplorer
```

You can choose a working directory (for example, `zcui.work`) to open with the **zBatchExplorer** GUI as displayed in the following figure:

**FIGURE 18.** Opening a Directory using zBatchExplorer

Once the task tree is displayed, you can choose between the log file of a task or zCui global log as displayed in following figure:

**FIGURE 19.** Selecting Log File and zCui Global File

Important Log Files

3.8.2 zTime

zTime is a Static Timing Analysis tool used during the compilation process.

The **zTime** results are accessible:

- In **zCui**, when displaying the log for the **zTime** step or within the Performance tab of the global log view.
- In **zBatchExplorer**, under the **Performance** tab of the global log view.
- As the `zTime.log` file in the backend directory.
- As the `zTime.html` web pages in the backend directory.

The following summary section is present at the end of the `zTime.log` file:

```
#-----#
Longest inter-fpga filter path delay is : 5 ns
Critical routing data path delay : 205 ns
  - Constant part      : 116 ns
  - Multiplexed part   : 89 ns
Xclock frequency is : 450 MHz
Longest memory period is : 140 ns
Driver clock frequency is limited by routing data paths
The theoretical frequency using default settings and ignoring
clock skew is 4884 Khz
#-----#
```

You can calculate the theoretical frequency estimation of the `driverClock`, also called the **zTime** estimation, using the following formula:

$$\text{TheoreticalFrequency} = \frac{1}{\text{Max}(\text{CriticalRoutingDataPathDelay}, \text{LongestMemoryPeriod})}$$

The `zebu.work/zTime.log` file contains summary tables of detected memories and critical routing paths. These tabular views are sorted in descending order of the delay value.

You can also see critical routing paths in detail with the static HTML web pages accessible from `zcui.work/zebu.work/zTime.html`. The detailed path is displayed in the following tabular view. In the following example, the signal has a multiplexing of 10 when it traverses from `F02` to `F00`.

Fpga	Delay	Arrival	X	zCore	Port	
U0/M0/I/F	10 ns	10 ns			RVALID_VP	RVALID_VP
	5 ns					
U0/M0/F02	15 ns	30 ns				
	43 ns		10			
U0/M0/F00	16 ns	89 ns		ZCORE_TOP	U0_M0_F0/RVALID_VP	RVALID_VP
	10 ns	99 ns		ZCORE_TOP	U0_M0_F0/RREADY_VP	RREADY_VP
	37 ns		8			
U0/M0/F02	15 ns	151 ns				
	5 ns					
U0/M0/I/F	10 ns	166 ns			RREADY_VP	RREADY_VP

FIGURE 20. Checking for Multiplexer/Hops on a Critical Routing Path

From the detailed reports, you can find out which paths or elements limit the maximum speed of ZeBu.

3.8.3 Gathering Information of a Compilation Database

To obtain partial or complete information of a compilation database, use the `zRscManager` command as follows:

To obtain partial information:

```
$ zRscManager -nc -compilationstatus ../zcui.work
```

Design information	
Status	: PASS
Platform	: ZS4
Release	: Q-2020.03
Number of modules	: 0.5
Driver clock frequency	: 10000 KHz

To obtain complete information:

Important Log Files

```
$ zRscManager -nc -compilationstatus ../zcui.work -compilationmetrics
-mappingsinformation
```

Design information				
Status	:	PASS		
Platform	:	ZS4		
Release	:	Q-2020.03		
Number of modules	:	0.5		
Driver clock frequency	:	10000 KHz		
Design Metrics				
REG	:	35972		
LUT	:	22387		
RAMLUT	:	64		
LUT6	:	5784		
MUXCY	:	56		
BRAM	:	48		
DSP	:	0		
Compilation Metrics				
Compile time:				
Wallclock time	:	3377 s	(56m:17s)	
Without grid delay	:	3236 s	(53m:56s)	
Compilation total cumulative time per queue:				
ZebuIse	:	3149 s	(52m:29s)	
ZebuSynthesis	:	10 s		
Zebu	:	280 s	(04m:40s)	
Compile time breakdown (wallclock time (s) / CPU time (s) / Memory (KB))				
VCS	:	14	11	551696
Synthesis	:	10	6	383136
RTLDB indexer	:	4	0	44000

System build :	16	12	2512640
Cores build :	55	51	4594256
System P&R :	10	6	1370016
FPGA P&R :	3149	3146	47363840
Postprocess database :	6	1	268416
Mappings information			
nbModuleMappings :	32		
nbUsedResources :	1		
(U0.HM0)			
(U1.HM0)			
(U2.HM0)			
(U3.HM0)			

For more information on this command, use the following command:

```
zRscManager -nc -help compilationstatus
```

4 Clock Delay Support in Verilog for ZeBu

The Clock Delay feature enables simulation style support of native clock generation in the design (using `#delay`). With this feature, all waveforms produced by ZeBu are time-based (instead of cycle-based).

When using with Run Manager, this feature allows you to drive emulation by time instead of by cycles.

This section describes the following sub-topics.

- [*Enabling Clock Delay Feature*](#)
- [*Support of Delays With `zceiClockPort`*](#)
- [*Supported Verilog Language Subset*](#)
- [*Unsupported Constructs*](#)
- [*Timescale and Precision*](#)
- [*Compilation Results*](#)
- [*`clockDelayPort` Macro*](#)
- [*Configuring `clockDelayPort` Through `designFeatures`*](#)
- [*Timestamp Clock*](#)
- [*`C_COSIM` and Vertical Solution Transactors*](#)
- [*`zDPI` and ZEMI-3*](#)
- [*Waveform Dumping*](#)
- [*Global Time*](#)
- [*Tolerance Support*](#)
- [*Limitations*](#)

4.1 Enabling Clock Delay Feature

To enable the clock delay, the following UTF command must be added.

```
clock_delay <args>
```

The following options are available:

- To enable the delay synthesis on a specific module, specify the `-module {...}` option or add an attribute `* (*ZebuClockDelay*)` on the module.
You can also specify both the options to enable the clock delay functionality on the union of the two sets.
If you have not specified these options, you **MUST** specify `zceiClocks`, which is emulated using the clock delay infrastructure to obtain time-annotated waveforms.
- To enable auto-tolerance computed at compile time, set `-auto_tolerance` to `true`.

To obtain a detailed list of options available with the `clock_delay` command, use
`vcs -help utf+clock_delay`.

4.2 Support of Delays With `zceiClockPort`

You can enable the `clock_delay` feature in a design without `#delay`. In this case, the configurations of the clocks in the `designFeatures` must be changed to configure the duty high/duty low duration.

When the `clock_delay` feature is enabled, the waveforms of the generated clocks have timestamps according to the clock shapes defined in `designFeatures`.

When the `clock_delay` feature is enabled in the presence of `zceiClockPorts`, the following limitations apply:

- In ZeBu Server 3, a single clock group is available. However, you can still use JTAG because it does not require controlled clocks.
- In ZeBu Server 4, multiple clock groups can be used. However, there is still only one time group.
- Only 15 `zceiClockPorts` can be specified for ZeBu Server 3.

Supported Verilog Language Subset

- Tolerance is not available.

`zceiClocks` in the presence of `clock_delay` can be configured through `designFeatures` using time or frequency. This is an example specification of `zceiClocks` with time:

```
$U0.M0.top.clk.Mode = "delay-controlled";
$U0.M0.top.clk.DelayUnit = "ps";
$U0.M0.top.clk.DutyLo = 1;
$U0.M0.top.clk.DutyHi = 1;
$U0.M0.top.clk.Phase = 0;
```

This is an example specification of `zceiClocks` with frequency:

```
$U0.M0.my_clock.Mode = "delay-controlled";
$U0.M0.my_clock.Frequency = my_realFreq; # a frequency in kHz
```

4.3 Supported Verilog Language Subset

The following supported constructs can be used in transactors and DUT.

- #delays in procedural blocks

```
always
#10 clk = ~clk;
```

- Variable (non-constant) delays

```
always begin
  var=var+1;
  #var clk = ~clk
end
```

Note

Variables can be written by the design or through zInject/zForce.

■ Delays in initial blocks

```
initial begin
    reset = 0;
    #1000;
    reset =1;
end
```

■ Mix of delays and events

```
always @(posedge event);
#1000;
sig = ~sig
end
```

■ Delays in tasks

```
task mytask()
begin
    reset = 0;
    #del;
    reset = 1;
end
endtask
```

■ \$time

```
always @(posedge clk)
if (event_occured)
data_of_event = $time;
```

Unsupported Constructs

- \$display %d and %t with \$time

```
always @(posedge clk)
    always @(posedge clk)
        if (contd)
            $display("cond occurred at %t", $time);
```

- Intra assignment delays

```
assign clk = #10 ~clk;
```

- Inertial delay in both gate primitives and continuous assignments is supported. It is supported using the "-gate_delay true" option.

```
assign #(1) a = b;
buf #(1) inst(o, i);
buf #(1, 2) inst(o, i);
buf #(1,2,3) inst(o, i);
```

NOTE: Only single input gates like buf and not are currently supported.

4.4 Unsupported Constructs

The following constructs are not supported.

- Continuous assignment delays

```
assign #10 a = b;
-$timeformat
-wire #delay w;
-AND #delay inst(...);
```

4.5 clockDelayPort Macro

clockDelayPort is simpler mechanism to generate clocks and reset using delays.

The `clockDelayPort` macro enables you to easily define a clock. It is an alternative to the Verilog `#delay` syntax.

The `clockDelayPort` macro can be used without naming the module in the `"clock_delay"` command. However, you must use the `clock_delay` command to enable the clock delay feature.

The `clockDelayPort` macro configures the clock frequency in the middle of an emulation run with runtime ZeBu APIs. It also provides the benefit of clock tolerance when changing clock frequencies.

The following table provides a list of equivalent code to when using this macro.

TABLE 8 ClockDelayPort Macro

Existing Method	Equivalent ClockDelayPort Macro
<pre>initial begin reset <= 1'b0; #2 reset <= 1'b1 end always begin #5 clk1 <=1'b1 #9 clk1 <=1'b0 end</pre>	<pre>clockDelayPort #(5,9,0,0,2) def_clk1(clk1, reset)</pre>
<pre>always begin #7 clk2 <=1'b1 #11 clk2 <=1'b0 end</pre>	<pre>clockDelayPort #(7,11,0) def_clk2(clk2)</pre>

The APIs listed in the following table allows you to reconfigure these clocks and resets at runtime.

clockDelayPort Macro

C++ API	C API	Description
<code>ClockDelayPort::doNewReset</code>	<code>ZEBU_ClockDelayPort_doNewReset</code>	Takes the hierarchical name of the <code>clockDelayPort</code> instance (example, "hw_top.cdp1") and the delay at which to toggle the reset. The "reset" port of this instance toggles at the designated delay.
<code>ClockDelayPort::reconfigureClockShape</code>	<code>ZEBU_ClockDelayPort_reconfigureClockShape</code>	Takes the hierarchical name of the <code>clockDelayPort</code> instance (e.g. "hw_top.cdp1") and the new duty lo, duty hi, phase and whether immediate reconfiguration is required. The "clock" port of the instance starts behaving accordingly.
<code>ClockDelayPort::disableClock</code>	<code>ZEBU_ClockDelayPort_disableClock</code>	Takes the hierarchical name of the <code>clockDelayPort</code> instance (e.g. "hw_top.cdp1"). This stops the "clock" port of this. But, it does not have any effect on any other clocks in the system. Emulation continues.
<code>ClockDelayPort::Configuration getConfig</code>	<code>ZEBU_ClockDelayPort_Configuration getConfig();</code>	Returns the current delay settings of particular clock delay port from hardware.

4.6 Configuring clockDelayPort Through designFeatures

After defining a clock using `clockDelayPort` in the design, and compiling it through **zCui**, you might want to specify the clock duration and shape differently. This can be done using C/C++/**zRci** APIs during the run or using `designFeatures` when the run starts.

Note *ClockDelayPorts have a default time precision of "ps". However, it can be changed using an option as follows:*

```
clock_delay -clockdelayport_timescale "1fs"
```

For a given `clockDelayPort` instance in the design whose hierarchical path was "`zebuAutoTB.dut1.cdp1`", use the following code to re-configure `clockDelayPort` using `designFeatures`:

```
$zebuAutoTB.dut1.cdp1.Mode = "clock-delay-port";
$zebuAutoTB.dut1.cdp1.DutyLo = 10;
$zebuAutoTB.dut1.cdp1.DutyHi = 10;
$zebuAutoTB.dut1.cdp1.Phase = 0;
$zebuAutoTB.dut1.cdp1.DelayUnit = "ps";
$zebuAutoTB.dut1.cdp1.ResetValue = 0;
$zebuAutoTB.dut1.cdp1.ResetDuration = 10;
```

where,

- `DutyLo` changes the duration for which the clock remains at `LOW`.
- `DutyHi` changes the duration for which the clock remains at `HIGH`.
- `Phase` is between 0 and (`DutyHi + DutyLo`) and decides how long the clock stays at `LOW` in the first cycle or should start `HIGH`.
- `DelayUnit` specifies the unit of arguments given in `designFeatures` only and is internally converted to "ps".
- `ResetValue` specifies if the reset is active low (0) or active high (1).

- `ResetDuration` changes the duration of the reset.

4.7 Timescale and Precision

Each Verilog file has a Verilog timescale. The timescales are resolved by the compiled design to compute the smallest precision required and the largest delay . This resolution applies to emulation and simulation.

`Board::getClockDelayPrecisionUnit` is the runtime API that can give a precision at runtime.

Computed timescale precision is the unit used internally after compiling all delays. The hardware precision is the maximum number of bits that can be used to encode delay in the hardware. As all delays are normalized to the timescale precision, the maximum delay depends on the hardware precision of the ZeBu clock generator. This precision can be 24 or 32 bits. The hardware precision can be set using the UFT command `clock_config -accuracy <24 | 32>`. The default precision is 24 bits.

In case, the required hardware precision exceeds the ZeBu clock generator precision, an error is issued by VCS. You can force the global timescale/precision on the VCS command line, if necessary.

If a variable delay is used and the size of the variable is greater than the clock accuracy, VCS issues a warning that delay value may overflow, and only the number of bits equal to precision are considered in the hardware.

4.8 Compilation Results

To find all synthesized delays, run the following command:

```
grep -A 4 ZEBU-CLK-DELAY zcui.work/zCui/log/
vcs_splitter_VCS_Task_Builder.log
```

This command provides the file name, line, if the delay is synthesized or not, and a snippet of the Verilog code.

```
Note-[ZEBU-CLK-DELAY] Found delay control  
design.sv, 4  
Module 'hw_top' has delay control.  
Following delay control will be synthesized.  
#(10) clk = (~clk);
```

Automatic Tolerance Value can be viewed using the
`designFeatures.<host>.help` option.

4.9 Timestamp Clock

4.9.1 ZeBu Server 3

When the `clock_delay` feature is enabled, a `zceiClockPort` (name of the timestamp) is automatically created. This clock is configured to be a dual edge clock (that is, both edges can be controlled by `C_COSIM`). This clock has an edge for each delay. When this clock is stopped, the time is also stopped.

Note

Only one clock group is supported in ZeBu Server 3. However, JTAG can be used as a clock because it does not require a controlled clock.

4.9.2 ZeBu Server 4

There is no Timestamp clock in ZeBu Server 4. A composite clock (pseudo-clock) called "tickClk" is created in the place of timestamp.

Note

The time scheduler only controls a default time group. All other clock groups in ZeBu Server 4 are not affected by stopping the time scheduler.

4.10 C_COSIM and Vertical Solution Transactors

The default controlled clock for C_COSIM and Vertical Solution Transactors is "timestamp". You may choose to override it with a different ZCEI clock by creating zceiClockPort and specifying the "defparam" to connect it to C_COSIM.

Note

General

- *Do not use defparam to connect zceiClockControl instance(s) in Vertical Solution transactors when clock_delay (#delay or clockDelayPort macro) is used for clock generation and no zceiClockPort is instantiated in the design.*
- *The zceiClockControl instance(s) in the Vertical Solution transactors are automatically handled by the tool.*
- *This is also applicable for C_COSIM transactor.*

In ZeBu Server 3

- *C_COSIM is supported without any zceiClock. The "timestamp" clock becomes the C_COSIM clock. However, the replacement clock, "tickClk", is not considered as the C_COSIM clock.*

In ZeBu Server 4

- *C_COSIM is only supported with the presence of zceiClock.*
-

4.11 zDPI and ZEMI-3

When the `clock_delay` and the `ZEMI-3 -timestamp true` (UTF option) are used, the `svGetTimeFromScope` function returns the `timestamp` of the imported call.

Example: zDPI or ZEMI-3 import function

```
extern "C" void time_tracker()
{
    uint64_t clock_time;
    clock_time=svGetTimeFromScope(svGetScope());
    printf("    TIMESTAMP %d \n",clock_time);
}
```

4.12 Waveform Dumping

With the Clock Delay feature, it is not necessary to provide a sampling clock for FWC/QiWC/readback waveforms. The waveforms are annotated with time when you view them in Verdi waveform viewer.

4.13 Global Time

You can obtain the current global time in the software by using RunManager::getGlobalTime or by calling the following C++ or C API:

```
/*
 * \brief Type for returning the current value of time
 *   timeUnit is explained by example here:
 *   If timeUnit is -9, it refers to "1ns"
 *   If timeUnit is -8, it refers to "10ns"
 *   The range of values is from 2 to -15
 */
typedef struct _ZEBU_CurrentTimeType
{
    unsigned long long int highWord;
    unsigned long long int lowWord;
    ZEBU_TimeUnit timeUnit;

#ifndef __cplusplus
    _ZEBU_CurrentTimeType()
        : highWord(0), lowWord(0), timeUnit(ZEBU_InvalidTimeUnit) {
    }
#endif
} ZEBU_CurrentTimeType;

ZEBU_CurrentTimeType Board::getCurrentTime() const
throw(std::exception);

/*! \brief Get the current time in 128 bits and the unit of time
 \param board handler to a \c ZEBU_Board
```

```
\retval ZEBU_CurrentTimeType containing 2 64-bit words for time  
(high and low) and unit of time  
*/  
  
ZEBU_CurrentTimeType ZEBU_Board_getCurrentTime(ZEBU_Board *board);
```

4.14 Tolerance Support

Clock tolerance is a performance feature where the clock event scheduler tries to collapse discrete clock events into a single driverClk for improving throughput. It is assumed that user understand clocks undergoing tolerance have separate domains. User should ensure that data is not pipelined (written in one clock domain and read in another clock domain) because if such events are combined, it can cause an unexpected behavior. Therefore, design should always be brought up first without clock-tolerance and it should later be enabled to test possibility of improving throughput.

ZeBu clock event scheduler allows a tolerance of “minimum delay in the system - 1”. This is also the default tolerance value when auto-tolerance is used. You can reduce tolerance to avoid certain clocks from collapsing events with others, if required.

Usage of clock tolerance can be set at compile time using the following flag:

```
clock_delay -module -auto_tolerance true
```

To disable auto-tolerance {} at runtime, add this to designFeatures:

```
$rtlClockToleranceValue = 0;
```

Tolerance advances time of T + tolerance value instead of T, and all events in the time window [T:T+tolerance] occur at the same driverClock. This reduces the number of driverClk transitions; but reduces accuracy.

Example

Six always blocks (A0-A6) using the same delay (always #delay clk=~clk;)

- A0 #13

Tolerance Support

- A1 #15
- A2 #17
- A3 #10
- A4 #23
- A5 #28

In standard scheduler, the following sequence (in red the minimum winner) is present:

A0	13	3	13	11	9	6	3	13
A1	15	5	2	15	13	10	7	4
A2	17	7	4	2	17	14	11	8
A3	10	10	7	5	3	10	7	4
A4	23	13	10	8	6	3	23	19
A5	28	18	15	13	11	8	5	2
decrement	10	3	2	2	3	3	3	2

The preceding scenario with auto-tolerance of 9 is as follows:

A0	13	$13 - 19 + 13 = 7$	$7 - 10 + 7 = 4$
A1	15	$15 - 19 + 15 = 11$	$11 - 10 = 1$
A2	17	$17 - 19 + 17 = 15$	$15 - 10 = 5$
A3	10	$10 - 19 + 10 = 1$	$1 - 10 + 10 = 1$
A4	23	$23 - 19 = 4$	$4 - 10 + 23 = 17$
A5	28	$28 - 19 = 7$	$7 - 10 + 28 = 25$
Min delay	10	1	1
Min Delay + Tolerance	$10+9=19$	$1+9=10$	$1+9=10$

Where,

- **Yellow cell:** Clock has an event, and second event is scheduled, remaining delay is 7.

- **Red cell:** Minimum delay.
- **No color cell:** Clock still has a pending delay and no event in this driver clock cycle.
- **Grey row:** The scenario with auto-tolerance 9 (since the smallest delay of a clock is 10) is as follows:
 - Delays between min and min+9 are granted.
 - Minimum delays are marked in red and additional delays granted are marked in yellow.

With the tolerance run data, it is clear that there are multiple always blocks granted for every driver clock cycle and that A5 sequence is able to loop in three driverclock cycles compared to eight in the regular scheduler.

Note that, tolerance is currently not supported in presence of variable delays.

4.15 Limitations

Following are the limitations of the clock delay feature.

- Generic control of time from software is not supported.
For now, a ZEMI-3 transactor must control the time.
- Only 15 zceiClockPorts are available when `clock_delay` is enabled.
NOTE: *This is not applicable to ZeBu Server 4.*
- Tolerance is not supported when clock-delays are used with zceiClockPorts in the ZeBu Server 3 flow .
- A single clock group is ONLY supported for ZeBu Server 3.
However, JTAG can be used in ZeBu Server (ZSE) flow because it does not require a controlled clock.
- Tolerance is not supported with variable delays.

5 Runtime

During runtime, a design is loaded into FPGAs of the ZeBu system to proceed with emulation. The ZeBu runtime can be controlled by any of the following ways, depending on the way the design is built:

- A cycle-based C/C++ program
- A transaction-based C/C++ program
- A simulator, such as, VCS or Platform Architect
- A synthesizable testbench
- A Tcl-based program that controls the ZeBu system emulation runtime: **zRun**.
zRun can be used as standalone or with any of the preceding options. **zRun** has a graphical interface but can also be used in batch mode.

You must designate a compiled design to the runtime software by specifying the path to the zebu.work directory.

This chapter discusses the following topics:

- *Controlling Runtime Parameters*
- *zRun*
- *Cycle-Based C/C++ Co-Simulation*
- *Reading Signal Values During Runtime*
- *Changing the Signal Values During Runtime*
- *Randomizing Signals and Memories During Runtime*

5.1 Controlling Runtime Parameters

The parameters that control runtime are set in the **designFeatures** file. The runtime software generates a template file, `designFeatures.<hostname>.help`, to help you write this **designFeatures** file. This file resides in the directory where the runtime software is launched or in the compilation output directory (typically `zebu.work`). The path to the **designFeatures** file can also be provided as an argument to the open method in the testbench. You can control the following runtime parameters using the **designFeatures** file:

- *Emulation Speed*
- *Clock Definition*
- *Memory Initialization*

5.1.1 Emulation Speed

By default, the emulation speed is set by the compilation. To analyze what affects emulation speed, see `zTime.log` in [zTime](#). The `zTime.log` file provides the information about the time taken by various emulation processes. It is possible to fine tune these settings by using the **designFeatures** file.

5.1.2 Clock Definition

To specify the parameters for design clocks declared as outputs of `zceiClockPort`, use the **designFeatures** file. These design clocks are grouped into clock groups. In most cases, you only use one clock group, but you can also have multiple groups. Each group may be separately defined because the tool determines the necessary relationships among clock groups.

When a clock group has several clocks, you must declare a frequency ratio either as a relative value using the `VirtualFrequency` parameter or by declaring a clock waveform using the `Waveform` parameter.

The following example displays how to declare a frequency ratio of 2 using the `VirtualFrequency` parameter. In this example, `clock2` is two times faster than `clock1`:

Controlling Runtime Parameters

```
$U0.M0.clock1.Mode = "controlled"
$U0.M0.clock1.Waveform = "_-"
$U0.M0.clock1.VirtualFrequency = 1;
$U0.M0.clock1.GroupName = "clock_group"
$U0.M0.clock2.Mode = "controlled"
$U0.M0.clock2.Waveform = "_-"
$U0.M0.clock2.VirtualFrequency = 2;
$U0.M0.clock2.GroupName = "clock_group"
```

The following example displays how to declare a frequency ratio of 2 using the Waveform parameter. In this example, `clock2` is two times faster than `clock1`:

```
$U0.M0.clock1.Mode = "controlled"
$U0.M0.clock1.Waveform = "_-"
$U0.M0.clock1.VirtualFrequency = 1;
$U0.M0.clock1.GroupName = "clock_group"
$U0.M0.clock2.Mode = "controlled"
$U0.M0.clock2.Waveform = "_--"
$U0.M0.clock2.VirtualFrequency = 1;
$U0.M0.clock2.GroupName = "clock_group"
```

To check the precision of the clock generator, set the following environment variable:

```
setenv ZEBU_VIRTUAL_FREQ_ROUND_ENABLE 1
```

By default, this function is disabled.

To set the search depth, set the following environment variable:

```
setenv ZEBU_VIRTUAL_FREQ_ROUND_SEARCH_DEPTH 3
```

By default, the value is set to 3.

C API computes the required precision bit from all clock frequency.

Example

The following is an example that shows the precision of the clock generator:

```
ZeBu : 13:56:43.294809 13 : 22095 :
testbench_4_basic_cpp_negedge_sampling_uni_clock : Required
precision: 9 bits*
ZeBu : 13:56:44.611310 14 : 22095 :
testbench_4_basic_cpp_negedge_sampling_uni_clock : Adjust driver
clock frequency as no FWC is enabled anymore.
ZeBu : 13:56:44.617412 14 : 22095 :
testbench_4_basic_cpp_negedge_sampling_uni_clock : Closing the
board, pid(22095), vmem(2811 m)
-- ZeBu : 01:46:14.963671      6 : 128778 : simv : Analyzing clock
precision...
-- ZeBu : 01:46:14.963688      6 : 128778 : simv :   read
clock="timestamp", group="timestamp", virtual frequency=1
-- ZeBu : 01:46:14.963696      6 : 128778 : simv : Required
precision: 1 bits
-- ZeBu : 01:46:14.963711      6 : 128778 : simv : The driverClk
frequency is 9.615 MHz
```

5.1.3 Memory Initialization

To initialize a memory, use the `memoryInitDB` command in the `designFeatures` file:

```
$memoryInitDB = "memory.init";
```

where, `memory.init` is a file containing one or multiple lines as follows:

```
<AAAA.BBBB.CCCC>.mem_core_logic memory.txt
```

where, `<AAAA.BBBB.CCCC>` is the hierarchical name of the memory.

5.2 zRun

zRun is a Tcl-based program that controls the ZeBu system emulation runtime. It can be used in either batch mode or as a graphical interface. It can be used as standalone with a synthesizable testbench or in addition to a C/C++ transactional or cycle-based testbench.

Note

*The \$DISPLAY environment variable must be set correctly before you start **zRun**, especially if you use a remote terminal.*

zRun provides Tcl functions to control emulation. The following command lists **zRun** functions:

```
zRun -functionList
```

This section consists of the following sub-sections:

- [*Common zRun Command-Line Options*](#)
- [*Controlling Clocks*](#)
- [*Managing Memory*](#)

5.2.1 Common zRun Command-Line Options

This section describes the common **zRun** command-line options. This section consists of the following sub-sections:

- [*Using zRun With a Tcl Script*](#)
- [*Using zRun With a C/C++ Testbench*](#)
- [*Starting zRun Without GUI*](#)
- [*Specific Recommendation for the zRun Batch Mode*](#)

5.2.1.1 Using zRun With a Tcl Script

The `-do <tclscript>` option specifies a Tcl script that is executed when **zRun** launches.

The Tcl script can contain any Tcl command and specific **zRun** commands. You can initialize memories and automate the entire run using Tcl commands.

This is also useful when an initialization phase is executed before the **zRun** GUI launches.

For example,

```
$> zRun -design <designFeatures> -zebu.work <zebu.work> -do <Tcl  
script>
```

5.2.1.2 Using zRun With a C/C++ Testbench

If you use **zRun** with a C/C++ testbench, you must specify a testbench executable after the **zRun** command and the `-testBench` option, as displayed in the following code snippet:

```
$> zRun -zebu.work <zebu.work> -testBench <testbench executable>
```

Note

zRun controls the emulation, including the start of the clocks.

5.2.1.3 Starting zRun Without GUI

Use the `-nogui` option to execute **zRun** in batch mode. You must provide a Tcl script to control emulation. This script must include all clock initialization.

For example:

To launch **zRun** in batch mode with the `designFeatures` file:

```
$> zRun -design <designFeatures> -zebu.work <zebu.work> -do <Tcl  
script> -nogui
```

zRun

To launch **zRun** in batch mode with a C/C++ testbench:

```
zRun -nogui -do < Tcl script> -testbench < testbench executable>
```

5.2.1.4 Specific Recommendation for the zRun Batch Mode

The following steps are recommended when **zRun** is launched in batch mode:

1. The testbench and the **zRun** script must be synchronized to ensure that the closing of the testbench does not invalidate the last commands in the script, such as `Dump_off` commands.
2. Use the following command line option to close **zRun**:

```
zRun -synchroClose
```

After the testbench is closed with `zebu->close()`, an explicit closing of **zRun** is expected (using `-nogui` in the script or using the **Close** button in the GUI).

3. Do not use the following code snippet in the script:

```
while { [ZEBU_getStatus]=="open" &&
       [ZEBU_Clock_getStatus clk]=="running" } { after 100 }
```

`[ZEBU_getStatus]=="open"` is not useful (script closes the connection).

Instead, use the following code snippet:

```
while { [ZEBU_Clock_getStatus clk]=="running" &&
       [ZEBU_synchroCloseStatus] == "false" } { after 100 }
```

For example:

```
ZEBU_Dump_file data.fsdb clk
ZEBU_Dump_on
ZEBU_Clock_enableForever clk
while { [ZEBU_Clock_getStatus clk]=="running" &&
       [ZEBU_synchroCloseStatus] == "false" } { after 1000 }
```

```
ZEBU_Clock_disable clk  
ZEBU_Dump_off  
ZEBU_close  
ZEBU_exit
```

5.2.2 Controlling Clocks

This section describes the commands to control the ZeBu clocks.

Note

The commands described in this section cannot be called if ZeBu is not successfully connected.

5.2.2.1 Obtaining a List of Clock Groups

In most cases, only one clock group (default group) is used. But, it is possible to define several groups in the **designFeatures** file with a **GroupName** option.

The **ZEBU_Clock_getGroupNameList** command returns a list of defined clock groups. Each name in this list can be used to obtain the list of clocks in that group.

The **ZEBU_Clock_getNameList** command returns the list of clocks for a specific clock group.

5.2.2.2 Enabling Clocks for N Cycles

Only one clock in a clock group is necessary to enable the group as a whole. When the group is enabled, all the clocks in the group run until the specified clock runs for the given number of cycles. All the groups should be enabled to run concurrently.

When a clock group contains multiple clocks, the clocks run synchronously according to the frequencies and clock waveform declared in the **designFeatures** file.

The maximum number of clock cycles ranges into the hundreds of billions of cycles.

More than one clock group can be run at the same time.

Note

When dumping is enabled, the ZEBU_Clock_enable command does not return anything until the run is finished. This limitation does not allow you to run several groups in parallel before dumping is completed.

Syntax:

```
ZEBU_Clock_enable <clk_name> <nb_cycles>
```

where,

- <clk_name> is the clock name.
- <nb_cycles> is the expected number of cycles for the specified clock.

For example:

The following script runs 10 cycles on each clock group, driven by the first clock of the group:

```
set clkGroupList [ZEBU_Clock_getGroupNameList]
# loop on each group
foreach group $clkGroupList {
    # get the list of clocks in the group
    set clkList [ZEBU_Clock_getNameList $group]
    # get the first clock of the group
    set firstClk [lindex $clkList 0]
    # run 10 cycles on the first clock
    # the other clocks of this group follow synchronously
    ZEBU_Clock_enable $firstClk 10
}
```

5.2.2.3 Enabling Clocks in the Free-Running Mode

The ZEBU_Clock_enableForever command enables a clock with no limit on the number of cycles to run, that is, the clock runs endlessly. This is useful when the ZeBu system is used for software debug.

Syntax:

```
ZEBU_Clock_enableForever <clk_name>
```

where, *<clk_name>* is the clock name returned by ZEBU_Clock_getNameList.

Note

With this command, you can start all clock groups simultaneously using the –debugDriverClk option in the zRun command line. It adds a new special clock group called driverClk, which contains only one clock called driverClk (same name as the clock group) and drives the ZeBu system clock:

- ❑ *Using the ZEBU_Clock_enableForever command on one clock of each clock group, no clock starts physically until the driverClk starts.*
- ❑ *All design clocks start simultaneously with driverClk.*

For example:

The following command lines start all design clocks simultaneously with driverClk.

Note

It works only if –debugDriverClk is used in the zRun command line

```
set clkGroupList [ZEBU_Clock_getGroupNameList]
# loop on each group
foreach group $clkGroupList {
```

zRun

```
# get the list of clocks in the group
set clkList [ZEBU_Clock_getNameList $group]
# get the first clock of the group
set firstClk [lindex $clkList 0]
# Start free running mode on the first clock
# note that no clock starts physically
# other clocks of this group follow synchronously when started
ZEBU_Clock_enableForever $firstClk
}
# Start driverClk that physically start all other clocks free
running
ZEBU_Clock_enableForever driverClk
```

5.2.2.4 Disabling Clocks

A clock group, which is enabled using the `ZEBU_Clock_enable` command or the `ZEBU_Clock_enableForever` command, can be disabled using the `ZEBU_Clock_disable` command.

The `ZEBU_Clock_disable` command stops all the clocks of a clock group defined by the selected clock.

Note

The selected clock can be different from the one used to enable a group.

Syntax:

```
ZEBU_Clock_disable <clk_name>
```

where, `<clk_name>` is the clock name returned by `ZEBU_Clock_getNameList`.

Note*Warning*

It is not possible to precisely control the time when a group is stopped, because disabling a clock is always asynchronous. It depends on the host computer speed and load, the PCIe bus load, and so on.

5.2.2.5 Getting the Clock Status

The ZEBU_Clock_getStatus command indicates whether the clock is enabled or disabled, which is useful when you need to know if a run has terminated.

Syntax:

```
ZEBU_Clock_getStatus <clk_name>
```

where, <clk_name> is the clock name returned by ZEBU_Clock_getNameList.
It returns running or stopped.

5.2.2.6 Getting the Number Clock Cycles Executed

The ZEBU_Clock_getCounter command returns the number of cycles a clock has executed since the last ZEBU_open command. This command is available during a run.

Syntax:

```
ZEBU_Clock_getCounter <clk_name>
```

where, <clk_name> is the clock name returned by ZEBU_Clock_getNameList.

For example:

The following script displays the number of cycles processed during a run:

```
set clkGroupList [ZEBU_Clock_getGroupNameList]
# get the first clock group
set firstClkGrp [lindex $clkGroupList 0]
```

zRun

```

# get the list of clocks in the group
set clkList [ZEBU_Clock_getNameList $firstClkGrp]
# get the first clock of the group
set firstClk [lindex $clkList 0]
# Start 1000000 cycles on firstClk
ZEBU_Clock_enable $firstClk 1000000
# Wait until end of run
# Note : use ZEBU_getStatus because a testbench (if used)
# could close the session during the run
while {      [ZEBU_getStatus] == "open" \
           && [ZEBU_Clock_getStatus $firstClk] == "running" } {
    # Display number of cycles executed
    set nbCycles [ZEBU_Clock_getCounter $firstClk]
    puts "$firstClk still running : $nbCycles has been executed"
    after 500
}

```

5.2.3 Managing Memory

The external ZRM memories, namely, BRAMs and LUTRAMs can be controlled from **zRun** Tcl scripts. This section consists of the following sub-sections:

- *Saving Memory Contents*
- *Loading Memory Contents*

5.2.3.1 Saving Memory Contents

You can save the content of a memory either to a file or to a buffer using the following commands:

- **ZEBU_Memory_storeToFile**: This command stores the memory content into a file. This command is dynamic, that is, it is available while emulation is running.

- `ZEBU_Memory_storeToBuffer`: This command stores the memory content and flushes it in the buffer passed as parameter.

Syntax:

```
ZEBU_Memory_storeToFile <mem_name> <file_name> [<first_addr>  
<last_addr>] [mode]  
ZEBU_Memory_storeToBuffer <mem_name> [<first_addr> <last_addr>]  
[format]
```

where,

- `<mem_name>` is the memory name returned by `ZEBU_Memory_getNameList`.
- `<file_name>` is the name of the memory file.
- `[<first_addr> <last_addr>]` are the decimal values of the first and last memory addresses to be written in the memory file or buffer.
You must specify both `<first_addr>` and `<last_addr>` or none of them. If you specify none, the entire memory content is stored to the file.
- **[mode]** sets the format of the output file (optional parameter):
 - `t` = text (default mode).
 - `b` = binary.
- **[format]** is the format for operation (optional parameter):
 - `%b` for binary.
 - `%o` for octal.
 - `%h` for hexadecimal.

For example:

The following command lines store each complete memory in a different file:

```
# get the memories list with '.' as hierarchy separator  
set memoryList [ZEBU_Memory_getNameList]  
# store it  
set i 0
```

```

foreach name $memoryList {
    puts "Store memory $name in file mem_$i"
    ZEBU_Memory_storeToFile $name mem_$i
    incr i
}

```

5.2.3.2 Loading Memory Contents

You can load content into a memory either from a memory file using `ZEBU_Memory_loadFromFile` or from a buffer using `ZEBU_Memory_loadFromBuffer`.

Loading Content From a File

The `ZEBU_Memory_loadFromFile` command loads the memory content from a file. You can specify to load all or a part of the memory content, as well as an offset for the location of the file to load. This operation is dynamic, that is, it is available while emulation is running.

Note

During a runtime, the precise moment when the content is loaded to the memory is not known. This is because the emulation continues during command interpretation. To avoid non-reproducible results, it is necessary to stop the clocks when loading memory.

Syntax:

```

ZEBU_Memory_loadFromFile <mem_name> <file_name> [<startAddress>
                                                    [<stopAddress> [<fileOffset> ]]]

```

where,

- `<mem_name>`: Specifies the name of the memory to load as returned by `ZEBU_Memory_getNameList`.
- `<file_name>`: Specifies the name of the memory file.

- <startAddress>: Specifies the decimal value of the first memory address in a file to load to the memory (optional parameter, only supported by binary files).
- <stopAddress>: Specifies the decimal value of the last memory address in a file to load to the memory (optional parameter, only supported by binary files).
- <fileOffset>: Offset for the location of the file to load.

For example:

The following command lines initialize every memory with different files:

```
# get the memories list with '.' as hierarchy separator
set memoryList [ZEBU_Memory_getNameList]
# initialize it
set i 0
foreach name $memoryList {
    puts "Initialize memory $name with file mem_$i"
    ZEBU_Memory_loadFromFile $name mem_$i
    incr i
}
```

Loading Content From a Buffer

The ZEBU_Memory_loadFromBuffer command loads the memory content from a buffer passed as parameter.

Note

During a runtime, the moment when the content is loaded to the memory is not known. This is because the emulation continues during command interpretation. To avoid non-reproducible results, it is necessary to stop the clocks when loading memory.

Syntax:

```
ZEBU_Memory_loadFromBuffer <mem_name> <buf_name> [<first_addr>
                                                       [format]]
```

where,

- <mem_name>: Specifies the name of the memory to fill as returned by ZEBU_Memory_getNameList.
- <buf_name>: Specifies the name of the buffer.
- [<first_addr>] : Specifies the Decimal values of the first memory addresses to be written in the memory file or buffer.
- If you do not specify <first_addr>, the entire memory content is loaded. The end address is determined by <first_addr> and size of the buffer.
- [format] is the format for operation (optional parameter, default is %d):
 - %d for decimal.
 - %b for binary.
 - %o for octal.
 - %h for hexadecimal.

5.3 Cycle-Based C/C++ Co-Simulation

In the cycle-based C/C++ co-simulation, the testbench controls clocks and can read/write individual signals using Xilinx's scan-chain feature. The cycle-based C/C++ co-simulation requires the instantiation of a C_COSIM driver in a design, typically the hardware top.

The following example displays how to write a C++ testbench for a cycle-based verification of a design:

```
#include <libZebu.hh>
#include "../zebu.work/dut_ccosim.hh"
using namespace ZEBU;
#include <exception>
#include <iostream>
using namespace std;
int main()
```

```
{  
    Board *zebu = NULL;  
    Signal *din = NULL;  
    Signal *cnt = NULL;  
    Signal *resetn = NULL;  
    Signal *ena = NULL;  
    Signal *load = NULL;  
  
    try  
    {  
        // ZeBu initialization  
        zebu = Board::open("../zcui.work/zebu.work");  
        dut_ccosim = Init_dut_ccosim(zebu);  
        if( dut_ccosim->connect() )  
        {  
            zebu->close( -1, "Fatal error : Cannot connect driver \n" );  
            exit( -1 );  
        }  
        zebu->init();  
  
        // get handles to design signals  
        cnt = dut_ccosim_drv.cnt;  
        din = dut_ccosim_drv.din;  
        resetn = dut_ccosim_drv.resetn;  
        load = dut_ccosim_drv.load;  
        ena = dut_ccosim_drv.ena;  
  
        // reset the design  
        *resetn = b0;  
        dut_ccosim->run(2);  
    }
```

Cycle-Based C/C++ Co-Simulation

```
// force some signals
*resetn      = b1;
*ena         = b1;
*load        = b1;
*din          = "0x0000000000000000fffff0";
dut_ccosim->run(1);
*load        = b0;
dut_ccosim->run(100);

// read some signal values
cout << "Read: " << hex << cnt->get(0) << endl;

}

catch( exception &except )
{
    cerr << except.what() << endl;
}

// close session
if( zebu )
{
    zebu->close( 0, "Simulation finished" );
}

return(0);
}
```

5.4 Reading Signal Values During Runtime

It is possible to read the value of any sequential signal at runtime using the Xilinx scan-chain feature.

This is also possible to read the value of combinational signals if a dynamic-probe has been added to them at compilation time.

For declaration of dynamic-probes, see [Reading Signals](#).

5.4.1 C++ Example

```
unsigned int data_value;  
Board *b = Board::open( "zcui.work/zebu.work" );  
string signalName = "hw_top.dut_0.count_1.cnt";  
Signal *data = b->getSignal( signalName.c_str() );  
data_value = *data;  
  
// alternative method for signals wider than 32 bits  
unsigned int nbWord = ( data->size() - 1 )/32 + 1;  
cout << signal_name << " = 0x";  
for( int i = nbWord-1; i >= 0; i-- )  
    cout << setw( 8 ) << setfill( '0' ) << hex << data->get( i );  
cout << endl;  
  
// alternative method to print the value directly  
cout << signalName << " = 0b" << data->fetchValue( "%b" ) << endl;
```

5.4.2 Tcl Example

```
// Reading a signal/register
ZEBU_Signal_read "hw_top.dut_0.Reg1(31:12)" %h
ZEBU_Signal_read "hw_top.dut_0.Reg2" %h
ZEBU_Monitor_flush
```

5.5 Changing the Signal Values During Runtime

ZeBu offers several possibilities to change the values of signals from the testbench during runtime.

- Deposit: The value stays until the next active clock edge (taking into account the enable signal).
- Inject: The value stays until the value defined by the design changes.
- Force: The value stays until it is explicitly released.

5.5.1 Depositing a value on a signal value

Deposit allows to overwrite the value of any sequential signal at runtime. The value stays until the next active clock edge (taking into account the enable signal).

5.5.1.1 C++ Example

```
unsigned int data_value = 0xFF;
Board *b = Board::open( "zcui.work/zebu.work" );
string signalName = "hw_top.dut_0.count_1.cnt";
Signal *data = b->getSignal( signalName.c_str() );
*data = data_value;
```

```
// alternative method for signals wider than 32 bits
uint64_t nbWord = ( data->size() - 1 )/32 + 1;
for( uint64_t i = 0; i < nbWord; i++ )
    data->set( i, data_value );
```

5.5.1.2 Tcl Example

```
ZEBU_Signal_write "hw_top.dut_0.count_1.cnt(31:12)" 0
```

5.5.2 Injecting a Value on a Signal

Injecting a signal means to set a user-defined value at runtime. The value is retained until the value defined by the design changes.

For declaration of injectable signals at compilation time, see [Forcing and Injecting Values](#).

5.5.2.1 C++ Example

```
using namespace ZEBU;
unsigned int wr_value = 0x410000;
string signalName = "hw_top.dut_0.count_1.cnt";
// Injecting value on a signal
if( Signal::IsInjectable( board, signalName.c_str() ) )
Signal::Inject( board, signalName.c_str(), &wr_value );
board->writeRegisters();
```

Changing the Signal Values During Runtime

5.5.2.2 Tcl Example

```
// Injecting a value on a register/signal  
ZEBU_Signal_inject "hw_top.dut_0.Reg2" 804000  
ZEBU_Monitor_flush
```

5.5.3 Forcing a Value on a Signal

Forcing a signal means to set a user-defined value at runtime from the testbench until it is explicitly released.

5.5.3.1 C++ Example

```
// Forcing value on a signal  
unsigned int value1 = 1;  
Signal::Force( board, "hw_top.dut_0.mem_incr", &value1 );  
board->writeRegisters();
```

5.5.3.2 Tcl Example

```
// For forcing a register/signal  
ZEBU_Signal_force "hw_top.dut_0.Reg1(31:12)" 2000
```

5.5.4 Releasing a Forced Signal

5.5.4.1 C++ Example

```
// Releasing an already forced signal  
Signal::Release( board, "hw_top.dut_0.mem_incr" );  
board->writeRegisters();
```

5.5.4.2 Tcl Example

```
ZEBU_Signal_release "hw_top.dut_0.mem_incr"
```

5.6 Randomizing Signals and Memories During Runtime

Signal randomization specifies the list of signals to randomize using the "selectSignalsToRandomize" method and randomizes the specified signals using the "randomizeSignals" method.

The following sections are related to the randomization of signals and memories:

- *Corruption in ZeBu Power Aware Emulation Flow*
 - initializeRandomizer: Initializes the generator that later applies values to registers, ports and memories in one or all power domains.
 - performRandomizer: Sets all elements of the design, whatever their power domain, to pseudo-random values, or 0 or 1 according to the mode selected in initializeRandomizer.
 - performRandomizerDomain: Sets all elements of a specific domain to pseudo-random values, or 0 or 1 according to the mode selected in initializeRandomizer.
- *\$memset() System Task*

5.6.1 C++ Syntax

```
void selectSignalsToRandomize( const char *signalList = 0 ) throw(
    std::exception );
void randomizeSignals( const unsigned int seed ) throw(
    std::exception );
```

Where:

- The parameter "signalList" is the name of a file specifying the list of signals to randomize.
 - The file must contain the list of hierarchical names of signals separated by an "end of line" character.
 - If the file name is NULL, then all sequential signals are randomized.
- The parameter "seed" of the "randomizeSignals" method specifies the seed of the Pseudo-Random Generator sequence.

5.6.1.1 C++ Example

```
// Specify the list of signals
board->selectSignalsToRandomize( "signals_to_randomize.list" );
// Randomize signals using seed 200
board->randomizeSignals( 200 );
```

With the file called "signals_to_randomize.list" containing:

- hw_top.dut_0.module1.sig_name
- hw_top.dut_0.module2.module3.sig_name

5.6.2 Tcl Syntax

```
ZEBU_selectSignalsToRandomize signalList 0  
ZEBU_randomizeSignals seed
```

where:

- "signalList" is the name of a file specifying the list of signals to randomize.
 - The file must contain the list of hierarchical names of signals separated by an "end of line" character.
 - If the file name is NULL, then all sequential signals are randomized.
- "seed" specifies the seed of the Pseudo-Random Generator sequence.

5.6.2.1 Tcl Example

```
ZEBU_selectSignalsToRandomize signals_to_randomize.list 0  
ZEBU_randomizeSignals 100
```

5.6.3 Memory Randomization

Memory randomization specifies the list of memories to randomize using the "selectMemoriesToRandomize" method and randomizes the specified memories using the "randomizeMemories" method.

5.6.3.1 C++ Syntax

```
void selectMemoriesToRandomize( const char *memoryList = 0 ) throw(  
std::exception );  
void randomizeMemories( const unsigned int seed ) throw(  
std::exception );
```

Where,

- "memoryList" is the name of a file specifying the list of memories to randomize.
 - The file must contain the list of hierarchical names of memories separated by an "end of line" character.
 - If the file name is NULL, then all memories are randomized.
- "seed" specifies the seed of the Pseudo-Random Generator sequence.

5.6.3.2 C++ Example

```
// Specify the list of memories
board->selectMemoriesToRandomize( "memories_to_randomize.list" );
// Randomize memories using seed 100
board->randomizeMemories( 100 );
```

With the file called "memories_to_randomize.list" containing:

- hw_top.dut_0.module1.mem1
- hw_top.dut_0.module2.module3.mem2

5.6.3.3 Tcl Syntax

```
ZEBU_selectMemoriesToRandomize memoryList invert.
// Select memories to randomize.
// List of memories should be specified in a file.
// select from the memories specified in the file.
ZEBU_randomizeMemories seed
// Randomize memories to pseudo random value using seed
```

Where

- "memoryList" is the name of a file specifying the list of memories to randomize.
 - The file must contain the list of hierarchical names of memories separated by an "end of line" character.
 - Invert should always be 0 (zero).

5.6.3.4 Tcl Example

```
ZEBU_selectMemoriesToRandomize mem_list_to_randomize 0
ZEBU_randomizeMemories 100
```

6 RunManager

RunManager is a transactor that provides a flexible and deterministic way to control the emulation runtime to mitigate the limitation of the C-COSIM and ZEBU::Clock object.

RunManager enables you to do the following:

- Control any zceiClock or any clock group and time
- Interrupt an action in progress (from software or hardware)
- Allow control from different threads and processes
- Enable or disable a given manager
- Advance clock/time can be achieved through blocking and non blocking calls:
 - Blocking calls methods do not return until the requested action is finished
 - Non-blocking calls methods return as soon as the requested action is communicated to the HW

Each feature that needs to control the clocks or time can checkout its manager during runtime. You can checkout directly an instance of run manager through a dedicated public API.

This section describes the following subtopics.

- *Enabling RunManager*
- *RunManager APIs*

6.1 Enabling RunManager

By default, RunManager is available in runtime with ZeBu Server 4. In ZeBu Server 3, RunManager must be explicitly enabled. To enable and specify the number of RunManager instances during compilation, add the following command in the UTF file:

```
run_manager -number_of_instances <N>
```

where, <N>: the number of RunManager instances to be used.

If this command is not specified in the UTF project file, no RunManager is instantiated in ZeBu Server 3 and five RunManagers are instantiated in ZeBu Server 4 by default.

6.2 RunManager APIs

RunManager API functions handle control clocks/time/clock groups. The following table lists the APIs supported with RunManager.

TABLE 9 APIs Supported with RunManager

API Name	Description
RunManager *Board::getMainRunManager()	Creates and returns a pointer to the main RunManager instance. If this API is called before Board::init, it stops the clocks at initial time. It provides additional control to start clocks when it is necessary.
RunManager *Board::getNewRunManager(unsigned int instanceNumber = 0)	Creates and returns a pointer to the RunManager instance. Returns NULL when all the instances of RunManager are used or the requested instance is not available.

TABLE 9 APIs Supported with RunManager

API Name	Description
RunManager::getClockStatus	<p>Returns the clock's status.</p> <p>The following values are enumerated:</p> <ul style="list-style-type: none"> • ZEBU_RM_CS_RUNNING • ZEBU_RM_CS_STOPPED • ZEBU_RM_CS_STOPPED_TRIGGER • ZEBU_RM_CS_STOPPED_OTHER • ZEBU_RM_CS_UNKNOWN
RunManager::advanceClock	<p>Allows to advance the clock using the number of edges specified as a parameter.</p> <p>This method can be blocking or non-blocking. When this is complete, the instance of run manager holds the clocks until next call is made to advance or free the clocks.</p>
RunManager::advanceTime	<p>Allows to advance the simulation time using the amount of time specified as a parameter.</p> <p>This method can be blocking or non-blocking. When this is complete, the instance of run manager holds the clocks until next call is made to advance or free the clocks.</p>
RunManager::advanceClockNonBlocking	<p>Allows to advance the controlled clock passed in an argument using the number of edges specified as a second argument.</p>
RunManager::interrupt	<p>Allows to interrupt the execution of the RunManager::advanceClock and RunManager::advanceTime methods.</p>
RunManager::freeGroup	<p>Allows to run a clock group indefinitely.</p>

TABLE 9 APIs Supported with RunManager

API Name	Description
RunManager::freeTime	Allows to run a time group associated clock indefinitely.
RunManager::freeRun	Allows to run all clocks indefinitely.
RunManager::stopGroup	Allows to stop a clock group.
RunManager::stopTime	Allows to stop a time group associated clock.
RunManager::getGlobalTime	Allows to fetch the emulation global time.
RunManager::getSimulationTimeUnit	Allows to fetch the emulation global time unit available when using the clock delay feature.
RunManager::SimulationTimeUnitsToTime	Allows to convert the simulation time units to time available when using the clock delay feature.
RunManager::SimulationTimeToTimeUnits	Allows to convert the simulation time to time units available when using the clock delay feature.

7 Using SystemVerilog Assertions

Assertion-based verification is widely used for functional validation due to its concise behavior description and fast failure identification. SystemVerilog Assertions (SVA) are defined in the IEEE-1800™ standard for SystemVerilog. ZeBu supports SystemVerilog assertions and provides various controls for compile and run time.

To use SVA in ZeBu, perform the following steps:

1. Enable SVA during design compile using UFT commands
2. Start SVA using the Start method
3. Generate reports for SVA post-emulation

This chapter discusses the following topics:

- *SVA Compilation*
- *Runtime Usage*
- *Post Processing SVA*
- *Supported SVA subset*

7.1 SVA Compilation

The compilation options for SVA are available using the UFT command `assertion_synthesis`. This command is mandatory to enable SVA.

Commonly used options to control the assertion synthesis through UFT are as follows:

- `assertion_synthesis -enable ALL`: To compile SVAs in a design.
- `assertion_synthesis [+/-]module <mod_name>`: To enable(+) / disable(-) SVA in the designated module `<mod_name>`.
- `assertion_synthesis [+/-]tree <hier_name>`: To enable(+) / disable(-) SVA in the designated hierarchy `<hier_name>`.
- `assertion_synthesis -auto_disable`: To disable SVA upon first failure. This helps to improve performance, but increases the compiled netlist.
- `assertion_synthesis -never_fatal`: To disable the signaling of failing SVAs. This mechanism can be used with the logic analyzer to help analyze SVA failures.
- `assertion_synthesis -report_only_failure`: Only to report SVA failures.

7.2 Runtime Usage

The synthesized assertions can be controlled at runtime through `zRun` or C/C++ interface. By default, assertions are disabled and no assertion failure message is reported.

SVA assertions can easily be used with the logic analyzer to stop the clocks and control the testbench. By default, all `$fatal` assertions are reported. Any synthesized assertion can be connected to or disconnected from this mechanism at runtime.

Assertions can be in one of the following three states:

- **Disabled**: In this state, no message is reported even if the conditions are failed.
- **Activated**: In this state, messages are reported.
- **Activated and Signaling**: In this state, messages are generated and failure is signaled.

Note

A delay exists between the assertion failure (or success) and the availability of the corresponding message being displayed.

The synthesized assertions can be controlled at runtime through zRun or C/C++ interface. By default, assertions are disabled and no assertion failure message is reported.

This section consists of the following sub-sections:

- [*Controlling Assertions From the C++ Testbench*](#)
- [*Controlling Assertions Using zRun*](#)
- [*Controlling Assertions Using zRun Tcl Commands*](#)

7.2.1 Controlling Assertions From the C++ Testbench

ZeBu provides a C/C++ API to control SVAs at runtime in both live processing and post-processing modes. The SVA class is available in the \$ZEBU_ROOT/include/SVA.h header file.

7.2.1.1 Starting Assertion Processing

To start the SVA processing, the SVA::Start method must be called. This method can be called at any time between Board::open and Board::close methods. Assertion can be enabled in the following two modes:

- [*Post-Processing Mode*](#)
- [*Live Processing Mode*](#)

Post-Processing Mode

In this mode, all assertion failure messages during design run are dumped in a report file for post-emulation analysis. A post-processing tool, zsverilogReport (see “[Post Processing SVA](#)”), reads this report file and generates assertion failure report. The processing interface is as follows:

```
//Post processing interface
static void SVA::Start(
    Board *board,
    const char *clockName,
    const char *filename,
    const unsigned int enableTypes = SVA::ENABLE_REPORT
) throw(std::exception);
```

where,

- `board` is the `ZEBU::Board` object.
- `clockName` is the name of the clock used for message reporting.
- `filename` is the name of the report file used for post processing.
- `enableTypes` can be one of the following:
 - `SVA::ENABLE_REPORT` to enable the report only (default).
 - `SVA::ENABLE_TRIGGER` to enable assertion failure detect only.
 - `SVA::ENABLE_REPORT | SVA::ENABLE_TRIGGER` to enable both the report and the failure detection.

Live Processing Mode

In this mode assertion failures messages are displayed on the screen (standard output) as well as in the runtime log file (with other runtime messages). The processing interface is as follows:

```
//Live processing interface
static void SVA::Start(
    Board *board,
    const char *clockName,
    const unsigned int enableTypes = SVA::ENABLE_REPORT
) throw(std::exception);
```

7.2.1.2 Stopping SVA Processing

To stop SVA processing, call the `SVA::Stop` method. The `SVA::Start` method must have been called prior to the `SVA::Stop` method, and the `SVA::Stop` method must be called before the `Board::close` method. The syntax of the `SVA::Stop` method is as follows:

```
static void SVA::Stop(Board * board);
```

7.2.1.3 Resetting SVA Fail Counters

To reset the value of failure counters for all assertions to 0, call the `SVA::Reset()` method as follows:

```
static void SVA::Reset();
```

You can also reset the failure counters using the `config sva_reset_counters` command. For details, see the *ZeBu Server Unified Command-Line User Guide*.

7.2.1.4 Dealing with Assertion Failures

Clock Stopping on Assertion Failure

When an assertion fails, the clocks can be stopped instantaneously.

To register an `OnStop` callback and enable Clock Stopping upon assertion failure:

```
SVA::EnableClockStoppingOnFailure(Board *board, ZEBU_SVA_OnStop
    callback, void *context)
```

where:

- `callback`: User callback function.
- `context`: User data pointer to be passed to the callback function when called.

To disable Clock Stopping upon assertion failure:

```
SVA::DisableClockStoppingOnFailure(Board *board);
```

Interface of the OnStop Callback

The interface of the OnStop callback is as follows:

```
typedef int (*ZEBU_SVA_OnStop) (const char
**failed_assertion_path,
 const unsigned int *failed_assertion_nb, void *context );
```

where:

- failed_assertion_path: Array of char* (size: failed_assertion_nb). Contains the path of each failing assertion during current stop.
- context: User data registered pointer.

7.2.1.5 Changing the Reported Severity Level

By default, only the assertions without action block or with an action block containing a \$error or \$fatal display a message on the screen when there is a failure.

The SVA::SelectReport method changes the minimum level of severity from which messages are displayed:

```
static void SelectReport(Board *board, const unsigned int severity
= ZEBU_SVA_Failed_Display) throw(std::exception);
```

The available values for severity are defined by the ZEBU_SVA_Severity type in

Runtime Usage

the ZEBU_ROOT/include/Types.h header file:

ZEBU_SVA_Failed_Fatal	(highest)
ZEBU_SVA_Failed_Error	
ZEBU_SVA_Failed_Warning	
ZEBU_SVA_Failed_Info	
ZEBU_SVA_Failed_Display	
ZEBU_SVA_Success	(lowest)

Example

When the severity argument is set to ZEBU_SVA_Failed_Warning, assertions that have an action block with \$warning and higher severities (\$error and \$fatal) are displayed.

7.2.1.6 Disabling/Enabling Assertions and Signaling SVA Failures

The activation of an assertion can be modified with SVA::Set method. By default:

- All the synthesized assertions are active and produce messages.
- Only fatal severity assertions are signaled.

The prototype is as follows:

```
static void Set(
    Board *board,
    const unsigned int types = SVA::ENABLE_REPORT,
    const char *regularExpression = NULL,
    const bool invert = 0,
    const bool ignoreCase = false,
    const char hierarchicalSeparator = '.'
) throw(std::exception);
```

where:

- **types**: Initialization value of an SVA or a group of SVAs. A legal value is 0 for no action;
- **regularExpression**: Path of an SVA or regular expression for an SVA group.
- **invert**: If true, inverts the selection performed by the regular expression.
- **ignoreCase**: Ignores case sensitivity for the regular expression.
- **hierarchicalSeparator**: Sets hierarchical separator for regular expression.

7.2.2 Controlling Assertions Using zRun

The Global Command panel of **zRun** GUI includes an **SVA** button, which is active when any SVAs are compiled in the design. This button opens the System Verilog Assertion panel, as displayed in the following figure.

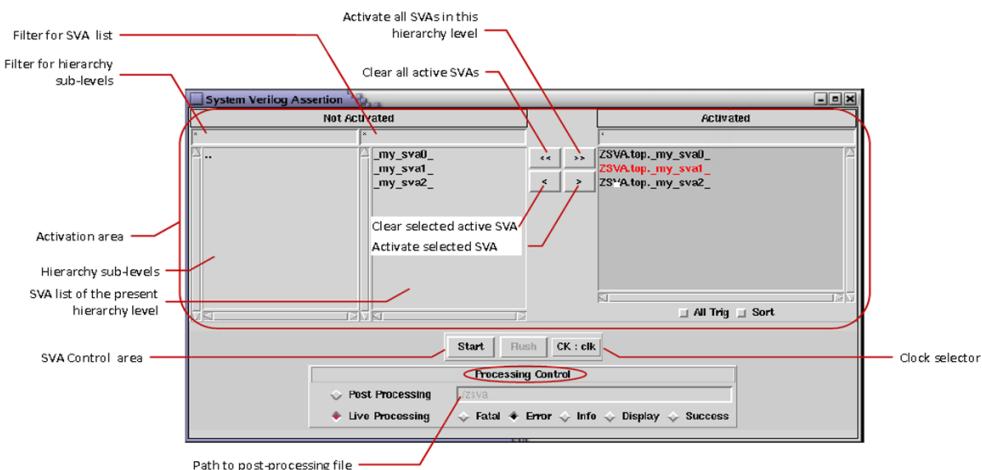


FIGURE 21. System Verilog Assertion Panel

The contents of this panel are set before starting emulation. They cannot be modified once the emulation is running.

7.2.3 Controlling Assertions Using zRun Tcl Commands

You can use the following ZEBU_Sva_* Tcl commands to control the assertions.

TABLE 10 zRun Tcl Commands

Commands	Description
ZEBU_Sva_isDefined	0: Failed 1: Succeeded
ZEBU_Sva_init	0: Failed 1: Succeeded
ZEBU_Sva_getNameList <hierarchyLevel> <filter>	List of all the SVAs' sub- <hierarchyLevel>
ZEBU_Sva_getLocalNode <hierarchyLevel> <filter>	List of all the SVAs' sub- <hierarchyLevel>
ZEBU_Sva_getLocalInstance <hierarchyLevel> <filter>	List of the SVAs' name at hierarchyLevel
ZEBU_Sva_setMaxErrorNumber <maxErrorNumber>	Do not display more than maxErrorNumber errors
ZEBU_Sva_getError [errorNumber]	Displays the reported error that are numbered as "errorNumber"
ZEBU_Sva_setSeverityReport <severity>	Severity must be "fatal" or "error" or "info" or "display" or "success"
ZEBU_Sva_getSamplingClock	Returns the selected group name and the associated clocks name list The first clock name is the currently selected one.
ZEBU_Sva_setSamplingClock <clockName>	clockName must be one of the clock names returned by ZEBU_Sva_getSamplingClock
ZEBU_Sva_start <mode> <samplingClock> [outputDir]	Specified outputDir for post-processing (no need in Live-mode)

TABLE 10 zRun Tcl Commands

Commands	Description
ZEBU_Sva_activate <sva> 0 1	Activates the specified SVA, 1 for enable, 0 for disable
ZEBU_Sva_disable <sva>	Disables the specified SVA
ZEBU_Sva_flush	Display the SVA messages that are yet to be displayed
ZEBU_Sva_stop	Stop all the enabled SVA

7.2.4 Runtime Options for Assertion Control Tasks

To disable/enable ACTs at runtime, the following APIs are added:

- `SVA::EnableAssertionControlTasks (Board *)`: Enables all the assertion control tasks in the design
- `SVA::DisableAssertionControlTasks (Board *)`: Disables all the assertion control tasks in the design

Note

By default, all the assertion control tasks are disabled. These tasks are enabled when enabling SVA at runtime using `SVA::Start()`.

If the assertion control tasks are enabled, disable them using the `SVA::DisableAssertionControlTasks()` function.

If the assertion control tasks are disabled using `SVA::DisableAssertionControlTasks()`, enable them again using `SVA::EnableAssertionControlTasks()`.

7.3 Post Processing SVA

If you select the **Post Processing** option for SVA in the **System Verilog Assertion Panel**, the **zsvaReport** tool generates the SVA messages post-emulation. The syntax for zsvaReport is as follows:

```
$ zsverilog -i <.zsverilog> [-z <zebu.work>] [-s <severity>]
[-n] [-m <number>] [-rt] [-h]
```

For example,

```
zsverilog -i dump.zsverilog -z <zebu.work>
```

where, `dump.zsverilog` is the directory passed to the `SVA::Start` method during runtime.

7.4 Supported SVA subset

In the following table, each construct has one of the following status:

- S in green background: Supported.
- S* in yellow background: Supported with the limitations listed after .
- U in red background: Unsupported.

TABLE 11 Supported SystemVerilog SVA Subset

Supported SystemVerilog SVA Subset		Status
SEQUENCES	<code>cycle_delay_range</code>	S
	<code>## integral_number</code>	S
	<code>##(const_expr)</code>	S
	<code>##(const_expr : const_expr)</code>	S
	<code>##(const_expr : \$)</code>	S*
Consecutive repetition	<code>[* const_expr]</code>	S

TABLE 11 Supported SystemVerilog SVA Subset

Supported SystemVerilog SVA Subset		Status
	[* const_expr : const_expr]	S
	[* const_expr : \$]	S*
Non-consecutive repetition	[= const_expr]	S*
	[= const_expr : const_expr]	S*
	[= const_expr : \$]	S*
Goto repetition	[-> const_expr]	S*
	[-> const_expr : const_expr]	S*
	[-> const_expr : \$]	S*
Sequence operators	throughout	S
	intersect	S
	within	S
	and	S
	or	S
	first_match	U
		S*
Sampled value functions		S*
● Sampled value function calls outside assertions not supported	\$sampled	S
	\$rose	S
	\$fell	S
	\$stable	S

Supported SVA subset

TABLE 11 Supported SystemVerilog SVA Subset

Supported SystemVerilog SVA Subset		Status
	\$past	S
	\$changed	S
System functions	\$onehot	S
	\$onehot0	S
	\$isunknown	S
	\$countones	S
Inferred value functions	\$inferred_clock	S
	\$inferred_disable	S
Global clocking	\$global_clock	S
Sequence local variables		U
Sequence instantiation		S
Sequences with formal arguments		S
Sequences involving function calls		S
Subroutine call on match of a sequence		U
Implicit first_match		U
PROPERTIES	Property instances	S
	Cross-module instantiations	U
	Recursive instantiations	U
	not	S
	or	S
	and	S
	if...else	S*

TABLE 11 Supported SystemVerilog SVA Subset

Supported SystemVerilog SVA Subset		Status
	disable iff	S*
	Implication	S*
	• Combination of implication and first_match not supported	S*
	Property instances	S
	Local variables	U
	Abort properties	reject_on
CLOCK DETECTION	Simple clock resolution	S
	Complex clock resolution	S
	Multi-clocked sequences	S
	Multi-clocked properties	S
	Clock flow analysis	S
ASSERTION CONTROL TASKS Feature supported through C++ API	\$assertkill	S
	\$asserton	S
	\$assertoff	U
CONCURRENT ASSERTIONS	assert	S
	assume	S
	cover	S*
	Concurrent assertions outside procedural code	S

Supported SVA subset

TABLE 11 Supported SystemVerilog SVA Subset

Supported SystemVerilog SVA Subset	Status
Concurrent assertions inside procedural block <ul style="list-style-type: none"> ● Assertions with clocking event different from clocking event of procedural block not supported 	S*
Concurrent assertions inside procedural loop <ul style="list-style-type: none"> ● Assertions with clocking event different from clocking event of procedural loop not supported ● Multi-clocked assertions not supported 	S*
DEFERRED ASSERTIONS	S
IMMEDIATE ASSERTIONS	S
EXPECT	U

8 DPI Synthesis Support in a Design

ZeBu supports SystemVerilog DPI (Direct Programming Interface) imported function calls inside the DUT. Unlike for transactors calls (see [Compiling ZeBu Transactors](#)), the ZeBu hardware-software infrastructure supporting these calls only allows communication from hardware to software (ZeBu to Host) to maximize the runtime efficiency. Therefore, DPI functions must have only inputs and no return value.

DPI function calls are synthesized, but the implementation of the DPI functions is provided as a dynamic runtime library. In a simulation environment, the function calls are initiated by the DUT.

DPI function calls can be controlled at runtime through **zRun** (Tcl commands) or using a C/C++ API.

ZeBu supports the following types of DPI function calls in the SystemVerilog source files:

- Imports only: Imported functions cannot call exported function.
- Functions only: Only operations that do not consume time are possible.
- Inputs only: Since data flows in one direction, the function cannot have any outputs.
- Only void import functions (with no return value) are allowed, that is, no outputs.

Note

If a DPI function call is not compliant with the preceding list, it is considered non-synthesizable and results in an error.

This section discusses the following topics.

- [Compilation](#)
- [Writing DPI Import Functions](#)
- [ZeBu API to Control DPI Functions at Runtime](#)
- [Processing DPI Function Calls Offline](#)
- [Optimization Guidelines](#)
- [Limitations](#)
- [Example](#)

8.1 Compilation

DPI synthesis is controlled with the `dpi_synth` UTF command.

To enable all DPI calls anywhere in the HDL to be synthesized use:

```
dpi_synth -enable ALL
```

It is also possible to have a finer control on the DPI calls to synthesize with the following options.

<code>dpi <instance name></code>	Selects the given DPI function
<code>module <module name></code>	Selects all DPI calls in all instances of the given module
<code>tree <hierarchy name></code>	Selects all DPI calls below the given hierarchy
<code>path <hierarchy name></code>	Selects all DPI calls inside the given hierarchy instance

These options act as selectors to include (sign before the option is +) or exclude (sign before the option is -) the calls from the synthesis.

For example:

```
dpi_synth -enable ALL -module dummy_call
```

Synthesizes all DPI calls present in the design except for the calls performed inside the `dummy_call` module

You can also review the synthesized DPI calls in VCS log file `zcui.work/zCui/log/vcs_splitter_VCS_Task_Builder.log`

Following is an example of the table summarizing the DPI calls identified by VCS:

Compilation

Source File	Type	Function Name	Path	# Inp	# Out	Filtered Reason	DPI
Source L	Status	Call					
/remote/user1/sources/fifo_usage_spy.sv	for module	fifo_usage_spy_notify fifo_usage_spy	sources/fifo_usage_spy.sv	6	0	/remote/user1/sources/fifo_usage_spy.sv import synthesized	DPI synthesis enabled

Synthesis also generates the following files:

- A log file for each module of the design that includes DPI calls.

This file is named as: `zcui.work/design/synth_Default RTL_Group/dpi_log/<module>.log`

This log file contains the following information for each DPI call:

# Function Name	Path	Bits Transferred	Source File
Source Line	Call	Nb	

- A specific header file that includes the prototypes of the synthesized DPI function calls. This file is stored in the back-end compilation directory.

This header is `zcui.work/<backend_compil_dir>/grp0_ccall.h`

8.2 Writing DPI Import Functions

This section consists of the following sub-sections:

- [C/C++ Language Compatibility](#)
- [Header Files](#)

8.2.1 C/C++ Language Compatibility

DPI import functions must be implemented as C functions using only the C types defined in the SystemVerilog standard. The SystemVerilog Language Reference Manual (LRM) defines a mapping between SystemVerilog and C types. The DPI import functions must be compiled with a C or C++ compiler.

Note

When using a C++ compiler, the DPI import functions must be declared as `extern C`.

8.2.2 Header Files

The standard header file for SystemVerilog is provided with the ZeBu software and it must be included when you compile the DPI import functions, as follows:

```
#include "svdpi.h"
```

The header file generated by synthesis (see [Compilation](#)) must also be included to check for consistency of the prototypes between the DPI function calls and their C implementation. This header file is included using the following command:

```
#include "zcui.work/<backend_compil_dir>/grp0_ccall.h"
```

8.3 ZeBu API to Control DPI Functions at Runtime

The ZeBu API provides the following control capabilities:

- Start or stop the DPI function calls (by default, DPI function calls are not active).
- Select a clocking mode, that is, specify a clock group or a clock expression.
- Enable function call synchronization.
- Select function calls only when their inputs change.

The API that controls DPI function calls supports the following options:

- Global control of all the DPI function calls.
- Explicit selection of a design scope to control DPI function calls.
- Pattern matching in the design scope.
- Individual DPI function calls with explicit path and name or based on an index within the scope. This index is known as the call number of the DPI function call.

The C++ API that controls DPI functions is defined by the `ZEBU::CCall` class. The `libZebu.hh` header file must be included in the testbench.

8.3.1 Clocking Mode

The clock signals and edges that initiate DPI function calls must be declared before you activate DPI function calls.

This section consists of the following sub-sections:

- *Specifying a Clock Group*
- *Specifying a Clock Expression*

8.3.1.1 Specifying a Clock Group

A clock group can be selected using the `SelectSamplingClockGroup` method. This method selects a clock group that calls the DPI functions on the simulation or emulation side. This method samples on positive or negative edges of all clocks in the selected group.

The syntax of this method is:

```
static void SelectSamplingClockGroup(Board *board, const char  
*clockGroupName = NULL) throw(std::exception);
```

where `clockGroupName` is the name of a controlled clock group declared in the `designFeatures` file. If `clockGroupName=NULL` then the first arbitrary group is selected (this is the default behavior).

Note

The `SelectSamplingClockGroup()` method cannot be called after `Board::init()` is called.

8.3.1.2 Specifying a Clock Expression

The sampling clocks can be specified with a clocking expression using the `SelectSamplingClocks` method as follows:

```
static void SelectSamplingClocks(Board *board, const char  
*clockExpression = NULL) throw(std::exception);
```

where `clockExpression` is a description of the clock signal and its active transition (edge) having the following format:

```
"[posedge|negedge] <clock name> [or [posedge|negedge] <clock  
name>] ...".
```

For example:

- "posedge clock1": Sampling on rising edges of `clock1`
- "posedge clock1 or negedge clock2": Sampling on rising edges of `clock1` and falling edge of `clock2`
- `clock3`: Sampling on rising and falling edges of `clock3`

NOTE: If `clockExpression=NULL`, all clocks and edges are selected.

Note

The SelectSamplingClocks () method cannot be called after Board::init () is called.

8.3.2 Enabling Synchronization of DPI Calls

By default, DPI function calls are not processed in a defined order. It is possible to synchronize DPI function calls so they are processed in the same order as they are called via the EnableSynchronization method.

The syntax of this method is:

```
static void EnableSynchronization(Board *board)
throw(std::exception);
```

Such synchronization is disabled by default because it decreases the runtime performance.

When synchronization is enabled, **zDPI** functions are executed in the same order as their call time. For a single emulation run, functions called at the same time are ordered in a predictable way.

8.3.3 Selecting Function Calls Only on Input Changes

ZeBu can arrange to call DPI functions to only when their inputs change by the SetOnEvent method. The syntax of this method is:

```
static void SetOnEvent(Board *board) throw(std::exception);
```

This specifies that DPI functions to be called only when their inputs change between two executions.

Note

You must call the SetOnEvent method before calling any DPI function.

8.3.4 Loading Dynamic Libraries

The dynamic libraries containing the C functions can be loaded using the `LoadDynamicLibrary` method. This method may be called multiple times to load multiple libraries. The syntax of this method is:

```
static void LoadDynamicLibrary(Board *board, const char *fullname)
throw(std::exception);
```

where `fullname` is the name of the dynamic library to load: [`<path>`/
`<library name>.so`].

If `<path>` is not specified, the `LD_LIBRARY_PATH` environment variable must include the path to `<library name>.so`.

Note *You must call the `LoadDynamicLibrary` method before calling any DPI function.*

8.3.5 Starting DPI Call Processing

Each DPI call can be controlled using the `Start` method. This method enables a set of DPI function calls (by default, all DPI calls are disabled). The DPI function calls can be specified by their scope name (or regular expression), import name, and call number.

To start the processing of import calls by name:

```
static void Start(
    Board *board,
    const char *scope = NULL,
    const char *importName = NULL,
    const int callNumber = -1
) throw(std::exception);
```

where,

- `scope`: Specifies the hierarchical scope containing the function calls to be enabled. If `scope` is `NULL`, all scopes are enabled.

- **importName:** Specifies the function name to be enabled. If **importName** is **NULL**, all functions are enabled.
- **callNumber:** Specifies the instance number (within the **scope**) of the function call to be enables. If **callNumber=-1**, all functions are enabled.

Note

- ❑ *It is acceptable for any of the preceding arguments to be **NULL** (-1 for **callNumber**).*
- ❑ *If any of the preceding arguments is not **NULL** (-1 for **callNumber**) then its constraint is applied.*
- ❑ *If two or more arguments are non-**NULL** (-1 for **callNumber**) then the constraints are combined.*

To start the processing of import calls specified in a regular expression for the scope:

```
static void Start(
    Board *board,
    const char *scopeExpression,
    const bool invert = false,
    const bool ignoreCase = false,
    const char hierarchicalSeparator = '.',
    const char *importName = NULL,
    ...const int callNumber = -1
) throw(std::exception);
```

where,

- **scopeExpression:** Regular expression that specifies the scopes of the function calls to be enabled. If is **scopeExpression** is **NULL**:
- All scopes are enabled, and
- **invert**, **ignoreCase**, and **hierarchicalSeparator** are ignored
 - ❑ **invert:** Inverts the regular expression
 - ❑ **ignoreCase:** Names are case insensitive

- **hierarchicalSeparator:** Specifies the hierarchical separator to use in the regular expression. Default value is (.)
NOTE: *The Start method can be called multiple times to enable multiple sets of function calls.*

8.3.6 Stopping DPI Call Processing

Each DPI call can be stopped by using the Stop method.

This method disables a set of function calls. The DPI function calls can be specified by their scope name (or a regular expression), import name, and call number.

To stop the processing of import calls by name:

```
ZEBU::C_CALL::Stop(  
    Board *board,  
    const char *scope,  
    const char *importName = NULL,  
    const int callNumber = -1  
) ;
```

where,

- **scope:** Specifies the scope containing the function calls to disable. If scope is NULL, all imports of all scopes are disabled.
- **importName:** Specifies the name of the called function to disable. If importName is NULL, all imports are disabled.

Note

- ❑ *callNumber: Specifies the instance number (within the scope) of the function to disable. If callNumber=-1, all functions are disabled.*
- ❑ *it is acceptable for any of the preceding arguments to be NULL (-1 for callNumber).*
- ❑ *If any of the preceding arguments is not NULL (-1 for callNumber), its constraint is applied.*
- ❑ *If two or more arguments are non-NULL (-1 for callNumber), their constraints are combined.*

To stop the processing of import calls specified in a regular expression for the scope:

```
static void Stop(
    Board *board,
    const char *scopeExpression,
    const bool invert = false,
    const bool ignoreCase = false,
    const char hierarchicalSeparator = '.',
    const char *importName = NULL,
    const int callNumber = -1
) throw(std::exception);
```

where,

- **scopeExpression:** Regular expression that specifies the scopes of the function calls to disable. If is scopeExpression is NULL:
 - All scopes are disabled, and
 - invert, ignoreCase, and hierarchicalSeparator are ignored.
 - invert: Inverts the regular expression.
 - ignoreCase: Names are case sensitive.

- `hierarchicalSeparator`: Specifies the hierarchical separator to use in the regular expression. Default value is `(.)`.

8.3.7 zRun Tcl Commands to Control the DPI Functions

When a C/C++ testbench is not used in the environment, the DPI function calls can be controlled from **zRun** using the following Tcl commands:

- `ZEBU_CCall_selectSamplingClockGroup clockGroupName`
- `ZEBU_CCall_selectSamplingClocks clockExpression`
- `ZEBU_CCall_enableSynchronization`
- `ZEBU_CCall_disableSynchronization`
- `ZEBU_CCall_setOnEvent`
- `ZEBU_CCall_unsetOnEvent`
- `ZEBU_CCall_loadDynamicLibrary fullname`
- `ZEBU_CCall_start [scope [importName [callNumber]]]`
- `ZEBU_CCall_stop [scope [importName [callNumber]]]`
- `ZEBU_CCall_start2 scopeExpression [invert=0 [ignoreCase=0 [hierarchicalSeparator=. [importName [callNumber]]]]]`
- `ZEBU_CCall_stop2 scopeExpression [invert=0 [ignoreCase=0 [hierarchicalSeparator=. [importName [callNumber]]]]]`

For functional details and information about parameters, see the corresponding method of the C++ API in [ZeBu API to Control DPI Functions at Runtime](#).

For an example of a **zRun** script to control the **zDPI** feature, see [Controlling the Calls from zRun](#).

8.4 Processing DPI Function Calls Offline

To enhance runtime performance, you can process DPI function calls offline. The DPI calls are handled after the emulation in a separate process by the **zdpiReport** tool instead of executing during emulation.

Processing DPI calls offline prevents:

- Internal DPI buffers from becoming full.
- Long periods of unavailability of the host PC that processes data.

Executing DPI calls offline is useful for functions that gather statistics and perform actions that do not impact the testbench.

To use this feature, perform the following steps:

1. Identify the DPI function calls you want to execute offline (see [Identifying the DPI Functions](#)).
2. Enable the DPI offline feature for emulation (see [Enabling the DPI Offline Feature During Emulation](#)).
3. Create a shared library that implements the DPI functions (see [Creating a Shared Library](#)).
4. Create an input file listing the DPI function names to be executed offline by **zdpiReport** (see [Creating an Input File for the zdpiReport Tool](#)).
5. After emulation, execute the DPI calls offline using **zdpiReport** (see [Using the zdpiReport Tool](#)).

8.4.1 Identifying the DPI Functions

Before using the offline processing feature, you must first consider which DPI functions you want to execute offline.

8.4.2 Enabling the DPI Offline Feature During Emulation

Once you have listed the DPI functions to process offline, you must enable the DPI offline feature and then run the emulation.

The following two use-models are available, depending on whether you can modify the testbench or not:

- If you can modify the testbench, add specific methods calls to the testbench to manage the offline processing. For more details, see [Using the DPI Offline Feature With Methods Within the Testbench](#).
- If you cannot modify the testbench, set the ZEBU_OFFLINE_DPI environment variable to a file that specifies the DPI calls to execute in offline mode. For more details, see [Using ZEBU_OFFLINE_DPI and its Offline DPI Specification File](#).

8.4.2.1 Using ZEBU_OFFLINE_DPI and its Offline DPI Specification File

The ZEBU_OFFLINE_DPI environment variable allows you to enable the DPI offline feature without modifying the testbench. This section consists of the following sub-sections:

- [Setting the ZEBU_OFFLINE_DPI Environment Variable](#)
- [Writing the Offline DPI Specification File](#)

Setting the ZEBU_OFFLINE_DPI Environment Variable

You set the ZEBU_OFFLINE_DPI environment variable to a file listing all the DPI function calls to execute offline as follows.

```
$ export ZEBU_OFFLINE_DPI=<path to offline_dpi_specification_file>
```

Once this variable is set, the offline DPI specification file is loaded before the first call to one of the following methods in the testbench:

- EnableSynchronization
- DisableSynchronization
- Start
- SetOfflineDumpName

For more information about these methods, see [ZeBu API to Control DPI Functions at Runtime](#).

Writing the Offline DPI Specification File

The offline DPI specification file contains the DPI calls to execute offline and determines how the offline execution process behaves.

Processing DPI Function Calls Offline

After writing the offline DPI specification file, you can use the Start method to enable the DPI offline feature on the DPI functions.

Note

- *it is recommended to use an absolute path for the offline DPI specification file.*
- *The offline DPI specification file can also be used by the ReadOfflineDpiSpecification method (see [Reading the Offline DPI Specification File](#)).*

The offline DPI specification file contains the following information:

- The target (mandatory): Specifies the name of the file where the offline DPI calls are dumped. This file has a .ztdb extension.
- One or multiple DPI function patterns (mandatory): Specifies the DPI functions to execute offline. A DPI function pattern is composed of:
 - The name of the DPI function call or a regular expression defining multiple DPI function calls at once.
 - Optionally, the full hierarchical name of the scope containing the function calls. If no scope is defined, all scopes containing the given DPI functions are marked for offline execution.
- The synchronization mode: Optionally specifies whether the synchronization of function calls is activated or not. Possible values are enabled or disabled (default value).

This setting in the offline DPI specification file overrides the EnableSynchronization and DisableSynchronization methods used at runtime in the testbench.

For more information about synchronization, see [Enabling Synchronization of DPI Calls](#).

The following rules apply to the syntax of the offline DPI specification declaration file:

- One DPI function pattern in one line.
- Blank lines and leading and trailing whitespaces are ignored.
- Comments must start with the pound character (#).
- Regular expressions for DPI function patterns must follow the syntax rules for Portable Operating System Interface (POSIX) Extended Regular Expression. To avoid partial matches, the circumflex (^) and dollar (\$) characters

automatically wrap regular expressions if they are not already present.

Example: f.*o matches foo or foofoo, but does not match foobar

- If the pattern for DPI function name is (*), it is replaced with (.*)

Example 1

An offline DPI specification file that marks all DPI functions for offline mode:

```
target=./my-dump.ztdb
*
```

where,

- Target: Defines the path to the ZTDB file.
- The asterisk (*) in the last line means that all DPI calls are marked for offline execution.

Example 2

An offline DPI specification file that marks four DPI functions for offline mode:

```
target=./my-dump.ztdb
synchronization=enabled
function1 scope=my_scope
function[234]
```

where,

- target defines the path to the ZTDB file.
- synchronization of DPI calls is enabled. The call order in the software side is the same as in hardware. For more information about synchronization, see [Enabling Synchronization of DPI Calls](#).
- my_scope The scope name of the function1 DPI function call.
- The last line is a regular expression defining three remaining DPI function calls to mark in all scopes for offline execution.

8.4.2.2 Using the DPI Offline Feature With Methods Within the Testbench

The methods described in this section allow you to enable and use the DPI offline feature by modifying the testbench directly.

Defining the ZTDB File

The SetOfflineDumpName method sets the name of the ZTDB file where the DPI calls are dumped. This method must be called before starting any DPI call in the offline mode. The syntax of this method is:

```
static void SetOfflineDumpName (
    Board *board,
    const char *dumpName,
    const bool dumpAllActivatedTracers = true
) throw(std::exception);
```

where,

- Dumpname: Specifies the name of the file where the DPI calls are dumped. dumpAllActivatedTracers=true means that all tracers active on a given FPGA are dumped into the ZTDB file. Please note that this parameter cannot be modified in current version.

For example,

```
SetOfflineDumpName(zebu, "myDump.ztdb");
```

Activating DPI Function Calls in Offline Mode

The StartOffline method marks the DPI function calls for offline execution. DPI function calls can be specified by their scope (or a regular expression), import name, and call number.

To start the offline processing of DPI calls by name:

```
static void StartOffline(  
    Board *board,  
    const char *scope = NULL,  
    const char *importName = NULL,  
    const int callNumber = -1  
) throw(std::exception);
```

where,

- scope: Specifies the hierarchical scope containing the function calls marked for execute offline. If scope is NULL, all scopes are marked.
- importName: Specifies the function name to mark for offline execution. If importName is NULL, all functions are marked.
- callNumber: Specifies the instance number (within the scope) of the function to mark for offline execution. If callNumber=-1, all functions are marked.

Note

- ❑ *It is acceptable for any of the preceding arguments to be NULL (-1 for callNumber).*
- ❑ *If any of the preceding arguments is not NULL (-1 for callNumber), its constraint is applied.*
- ❑ *If two or more arguments are non-NULL (-1 for callNumber), then the constraints are combined.*

To start offline processing of function calls specified using a regular expression for the scope use:

```
static void StartOffline(  
    Board *board,  
    const char *scopeExpression,
```

```

    const bool invert = false,
    const bool ignoreCase = false,
    const char hierarchicalSeparator = '.',
    const char *importName = NULL,
...const int callNumber = -1
) throw(std::exception);

```

where,

- scopeExpression: Regular expression that specifies the scopes of the function calls to mark for offline execution. If scopeExpression is NULL:
 - All the scopes are marked.
 - invert, ignoreCase, and hierarchicalSeparator are ignored.
 - invert: Inverts the regular expression.
 - ignoreCase: Names are case insensitive.
 - hierarchicalSeparator: Specifies the hierarchical separator to use in the regular expression. Default value is (.).

Note

The StartOffline method can be executed multiple times to start multiple function calls.

Reading the Offline DPI Specification File

The optional ReadOfflineDpiSpecification method reads and uses an offline file that specifies all DPI functions to execute offline. This method is equivalent to the ZEBU_OFFLINE_DPI variable described in [Identifying the DPI Functions](#).

The syntax of this method is:

```

static void ReadOfflineDpiSpecification(
    Board *board,
    const char *fileName
) throw(std::exception);

```

where `filename` is the path to the offline DPI specification file.

For example,

```
ReadOfflineDpiSpecification(zebu, "./dpi_spec_file");
```

8.4.3 Creating a Shared Library

After running the emulation with the DPI offline feature, you must create a shared library implementing the DPI functions to be executed offline with the `zdpiReport` tool.

For example

```
$ g++ -fPIC -I$ZEBU_ROOT/include -shared my-functions.cc -o my-functions.so
```

8.4.4 Creating an Input File for the `zdpiReport` Tool

You must create a file that lists the names of DPI functions to execute offline by `zdpiReport`. The syntax of this file is similar to the one used by the offline DPI specification file.

8.4.5 Using the `zdpiReport` Tool

`zdpiReport` is a dedicated tool that executes the offline DPI function calls dumped during emulation. This tool can be launched as follows:

```
$ zdpiReport -f <flist filename> -i <.ztdb filename> -l <.so filename> [options]
```

[Table 12](#) and [Table 13](#) list mandatory parameters and options, respectively, for the `zdpiReport` command.

TABLE 12 zdpiReport Mandatory Parameters

Parameters	Description
-f <flist filename>	Specifies the filename containing the names of DPI functions to be executed by zdpiReport .
-i <.ztdb filename>	Specifies the ZTDB filename where the offline calls were previously dumped.
-l <.so filename>	Specifies the shared library containing the DPI function implementation.

TABLE 13 zdpiReport Options

Parameters	Description
-synchronize	Synchronizes DPI calls.
-z <zebu.work>	Specifies the compilation directory where the runtime database is available (default is . /zebu.work).
To speed up decoding	
[-mtdecode]	Enables parallelized.ztdb decoding.
[-mtexec [N]]	Enables parallelized zDPI execution, in at most N threads.
[-mtread M[,T]]	Enables parallelized.ztdb reader: maximum M MB of memory and T threads.
[-synchronize-scope]	Synchronizes DPI calls within scopes.
<ul style="list-style-type: none"> • If the [-mtexec [N]] option is used, functions belonging to different synchronization groups must be thread safe. • If the -synchronize-scope option is used, a separate synchronization group is created for zDPI calls in each scope. • If -f is used multiple times with -synchronize, functions enabled by each -f file are placed in a separate synchronization group. • If the -synchronize option is not used, all functions execute in a single thread. 	

TABLE 13 zdpiReport Options

Parameters	Description
To diagnose decoding	
[-diag <features>]	<p>Enables specific diagnostic features. Where,</p> <ul style="list-style-type: none"> • <features> is a list of comma-separated name [=value] pairs: • dump: Prints information about each executed zDPI call. • file=FILENAME: Redirects diagnostics to the given file. If this option is not used, the output is printed to the console and saved in the ZeBu log file. • filter=REGEX: Only enables diagnostics on functions selected by the given regular expression: <ul style="list-style-type: none"> • The expression is matched against scopeName.functionName. • By default, a substring match is performed. For a full match, use the ^ and \$ anchors. • If the switch is used multiple times, the union of all matches is applied. • This must be the last item in <features> (REGEX patterns may contain commas). • from=TIME: Enables diagnostics from the given time onwards. However, only the last occurrence of this switch is applied. • nocall: Prints diagnostic information only; do not execute the function. <ul style="list-style-type: none"> • This option may be in effect for the entire duration of the run, even if from/to is used. • profile: Prints summary information at the end of the selected period: <ul style="list-style-type: none"> • Number of calls for each function. • Time spent inside functions (all calls accumulated). • to=TIME: Disables diagnostics after the given time. Only the last occurrence of this switch is counted. <p>The switch can be used multiple times: zdpiReport -diag dump -diag from=12.</p> <p>If neither dump, nocall nor profile is used, dump is implied.</p>

Processing DPI Function Calls Offline

For example,

```
$ zdpiReport -f my-functions.lst -i rundir/myDump.ztdb  
-z zcui.work/zebu.work -l ./my-functions.so
```

Note

As **zdpiReport** does not need to connect to ZeBu, it is recommended to launch it from a system other than the host PC.

8.4.5.1 Speed-up Decoding

Decoding speed can be increased using multi-threading or module-based synchronization.

When DPI calls synchronizations are required, you can speed-up decoding by limiting the synchronization of DPI calls at scope level only [-synchronize-scope]. The speed can further be increased using multi-threaded decoding [-mtexec [N]].

8.4.5.2 Diagnostics

Diagnostics can be used to analyze what is happening between hardware and software, and when and which DPI calls are executing.

Example

```
zdpiReport -diag dump,file=dump.log -i <ztdb> -l <libs> -f  
<filelist> -synchronize  
$cat dump.log  
    12 00000080 top.aaa.bbb.dpi1  
    20 00000080 top.aaa.bbb.dpi2  
    24 00000A10 top.aaa.ddd.dpi3  
    50 00000BBB top.eee.fff.dpi4  
    76 00000CFE top.eee.ggg.dpi5
```

8.4.5.3 Profiling

Profiling can be used to analyze which DPI calls are consuming most of the time (includes any I/O delays).

Example

```
zdpiReport -diag file=profile.log,profile -i <ztdb> -l <libs> -f
<filelist> -synchronize

$cat profile.log
zDPI calls: 42.512s, 16733741.

 27.465    3080816 top.aaa.bbb.dpi1
   7.678    2901869 top.aaa.bbb.dpi2
   3.412    2292203 top.aaa.ddd.dpi3
   1.622    623494  top.eee.fff.dpi4
   0.682    807176  top.eee.ggg.dpi5
   ...
...
```

8.5 Optimization Guidelines

This section contains recommendations for optimal use of the **zDPI** feature and consists of the following sub-sections:

- *Enhancing Data Transfer*
- *Defining Sampling Clock Frequency*
- *Investigating Emulation Slowdowns*

8.5.1 Enhancing Data Transfer

When calling DPI functions from the testbench, it is recommended to use data types that map to native C data types like `char`, `shortint`, and so on.

SystemVerilog DPI introduces data types like `svBitVecVal` and `svOpenArrayHandle` for mapping SystemVerilog data types to C. However, these can impact performance, as they require additional conversion operations on the host side. The table below lists the mapping between SystemVerilog types with C types.

TABLE 14 Mapping Between SystemVerilog Types with C Types

SystemVerilog Type	C Type
<code>byte</code>	<code>char</code>
<code>int</code>	<code>int</code>
<code>real</code>	<code>double</code>
<code>bit</code>	<code>unsigned char</code>
<code>shortint</code>	<code>short int</code>
<code>longint</code>	<code>long long</code>
<code>shortreal</code>	<code>float</code>
<code>string</code>	<code>char *</code>
<code>open Arrays</code>	<code>const svOpenArrayHandle</code>
<code>Bit/logic vector</code>	<code>const svBitVecVal/ svLogicVecVal</code>
<code>Bit/logic vector array</code>	<code>const svBitVecVal*/ svLogicVecVal*</code>

To maximize data transfer speed to the host PC, consider the following factors:

- Avoid unused bits by using the smallest data type possible.
- Avoid logic vector (four-state) arguments and unpacked structures.
- Use arguments smaller than or equal to 32 bits.
- Concatenate large numbers of short vectors in Verilog and unpack them on the C++ side. This allows for more data transfer with less communication overhead.
- Bit-vectors with sizes that are multiples of 32-bits yield greater performance. Similarly, use of 64-bit integers natively mapped to `longint` on the C-side enhance performance.

NOTE: If a memory or array is 16-bit or 8-bit wide, then you should use the appropriate

data type for building C-side arrays. For example, use byte to map the 8-bit wide vectors, shortint for 16-bit wide vectors, and so on.

- Using wider elements than needed is not recommended for the following reasons:
 - It increases memory usage
 - Unused bytes added to pad array elements take valuable CPU cache space and reduce overall performance.

8.5.2 Defining Sampling Clock Frequency

The maximum frequency for the DPI sampling clock is based on the following formula:

$$\frac{50 \text{ MHz}}{((\text{size_in_bits} \div 32) + 5)}$$

This means that if all the DPI calls are performed on the same sampling clock (on posedge or negedge), using a slower primary clock is recommended.

For example,

When packing 8-words, the maximum frequency is 5 MHz. As data is only sent every eight cycles, it is possible to use a slower sampling clock which is one-eighth of the design clock. Therefore, the maximum design clock frequency is computed as follows:

$$5 \text{ MHz} \times 8 \text{ cycles} = 40 \text{ MHz}$$

8.5.3 Investigating Emulation Slowdowns

If the **zDPI** feature is believed to cause an emulation slowdown, it can be investigated using the following ways:

- *Checking with zDPI Disabled*
- *Checking With zDPI Enabled*

8.5.3.1 Checking with zDPI Disabled

You can try to run the emulation with the `zDPI` feature disabled (with calls to the `ZEBU::CCall::Start` method commented out) provided it does not break the testbench: This allows you to check whether `zDPI` is causing a slowdown or not.

If frequent DPI calls are causing a slowdown, combining multiple DPI calls into one and limiting the argument lists is recommended. Each DPI call incurs a minimum overhead of 32-bits plus additional 32-bit words for argument values.

For example,

If `f()` and `g()` are functions, each taking a one-bit wide argument (SystemVerilog bit data type), then "`f(a); f(b); g(c); g(d);`" causes a transfer of 4×2 words.

Combining DPI Calls

Combining DPI calls is a way to minimize the transfers. Argument lists are limited to 480 bits - larger argument lists are split into multiple pieces.

Take into account the formula for DPI sampling clock frequency as described in [Defining Sampling Clock Frequency](#).

For example,

In the preceding example (see [Checking with zDPI Disabled](#)), if you create a new function combining the four calls `ffgg` (bit `a`, bit `b`, bit `c`, bit `d`), you can reduce the total transfer size from 8-words to 2-word.

Limiting the Argument Lists

It is recommended to declare each argument with the narrowest size possible. For example, do not use the `int` data type if the argument can only be 0 or 1.

8.5.3.2 Checking With zDPI Enabled

When the `ZEBU_IGNORE_DPI` environment variable is set, all DPI data coming from the emulator is ignored. Therefore, you can compare runs with or without the `ZEBU_IGNORE_DPI` environment variable and determine the overhead in the processing of DPI calls.

If there is a difference, you can try to run the emulation with some DPI functions

commented out. This allows you to assess if there is still possibility for code optimization.

If you still have a performance gap, you must analyze the arguments' layout (preferably in the generated `grp0_ccall.c` source) and look for any inefficient functions.

Note

This is a diagnostics feature that might break the test functionality.

8.6 Limitations

This section lists the limitation of the **zDPI** feature in the current version of the software.

8.6.1 Multiple Clock Groups

If multiple clock groups are declared in the `designFeatures` file, some DPI calls might not be detected if they do not operate on a clock of the group selected with the `SelectSamplingClockGroup` method.

Note

*Due to this limitation, it is recommended to use only one clock group when running a design with the **zDPI** feature.*

8.6.2 Performance

Multiple start and stop DPI requests (with `Start` and `Stop` methods in the testbench) impact the performance of the emulation.

8.6.3 zRun GUI

The **zRun** graphical interface is not upgraded with the **zDPI** feature.

Example

8.6.4 Synthesis

- DPI function calls in `always_latch` blocks, `always_comb` blocks, or for loops are not synthesized.
- DPI function calls in functions/tasks are not synthesized.
- System tasks are not supported in this version.

8.7 Example

This example implements a 32-bit counter in which DPI function calls are used to print the value of a counter. The DPI function has only inputs (the value of the counter), thus, it can be implemented in ZeBu with the **`zDPI`** feature.

The verification environment is C++ co-simulation. This example demonstrates two different methods to control the DPI function calls:

- From the C++ co-simulation testbench
- From **`zRun`** with a dedicated script

8.7.1 Design

This section describes the design aspect of the example and consists of the following sub-sections:

- *System Verilog Code*
- *Synthesis*

8.7.1.1 System Verilog Code

The DPI function is called from the SystemVerilog code. To do so, the function must be imported in the module where it is called as displayed in the following code snippet:

```
import "DPI" function void print_count (input bit [31:0] id,
input bit ci,  input bit [size-1:0] count);
```

The DPI function is called from the design on each rising edge of the design clock `clk`

as displayed in the following code snippet:

```
// DPI function call  
always @(posedge clk) begin  
    if (print_count_enable == 1'b1) begin  
        print_count(1, ci, counts00h);  
    end  
end
```

The Following is the complete SystemVerilog test:

```
module counter (clk, reset, load, data, count, ci,  
print_count_enable, co);  
  
parameter size = 32;  
  
input      clk;  
input      reset;  
input      load;  
input [size - 1:0] data;  
output     [size - 1:0] count;  
input      ci;  
input      print_count_enable;  
output     co;  
  
reg [size - 1:0]  counts00h;  
// Declaration of the DPI function  
import "DPI" function void print_count (input bit [31:0] id,  
input bit ci,  input bit [size-1:0] count);  
always @(posedge clk or posedge reset) begin
```

Example

```
if (reset) begin
    counts00h <= {size{1'b0}};
end
else if (load) begin
    counts00h <= data;
end
else if (ci) begin
    counts00h <= counts00h + {{(size - 1){1'b0}}, 1'b1};
end
end

assign count = counts00h;
assign co = (ci && (counts00h == {size{1'b1}}));

// Call of the DPI function
always @(posedge clk) begin
    if (print_count_enable == 1'b1) begin
        print_count(1, ci, counts00h);
    end
end

endmodule
```

8.7.1.2 Synthesis

The design explained in [Design](#) can be synthesized with **zFAST**. Check the settings described in [Compilation](#).

After synthesis, the summary of the DPI calls can be found in the dedicated log file for the counter module:

```
zcui.work/design/synth_Default RTL_Group/dpi_log/counter.log
```

This file contains the following information for the print-count function:

#	Function Name	Path	Bits Transferred	Source File
	Source Line	Call Nb		
	print_count	{counter}	65	../../../../src/
	counter.v	38	0	

8.7.2 Implementation of DPI Functions

The import functions are compiled into a dynamic library compiled with g++. The DPI functions are declared as extern "C".

The grp0_ccall.h file is included to check that the prototypes of the user DPI functions are compliant with the import declarations in SystemVerilog.

The print_count DPI function is implemented in the dpi.cc file:

```
#include "svdpi.h"
#include "grp0_ccall.h" // generated during compilation
#include <stdio.h>

extern "C" void print_count(const svBitVecVal *id, const svBit ci,
const svBitVecVal *count)
{
    svScope s = svGetScope();
    printf("Counter '%s', id=%u, count=%u\n", svGetNameFromScope(s),
id[0], count[0]);
}
```

In this example, a dpi.so dynamic library is created from dpi.cc:

```
$ g++ -fPIC -shared -o dpi.so dpi.cc -I$ZEBU_ROOT/include
-Izcui.work/zebu.work
```

Example

8.7.3 Controlling DPI Calls from the Testbench

This section describes how DPI calls can be controlled from the C++ testbench and consists of the following sub-sections:

- *Implementation of the Testbench*
- *Proceeding With Runtime*

Note

*DPI calls may also be controlled using **zRun**. However, they cannot be controlled simultaneously by both the C++ testbench and **zRun**.*

8.7.3.1 Implementation of the Testbench

In this example, the C++ testbench controls the co-simulation driver and the DPI calls.

The following are the specific calls that control the DPI calls:

- To load the `dpi.so` dynamic library:

```
CCall::LoadDynamicLibrary(z, "dpi.so");
```

- To sample the DPI calls on both edges of clock `clk`:

```
CCall::SelectSamplingClocks(z, "clk");
```

- To start all the DPI calls:

```
CCall::Start(z);
```

The source code of the testbench (`tb.cc`) is as follows:

```
#include <libZebu.hh>
using namespace ZEBU;
using namespace std;
```

```
int main() {
    try {
        // Board init
        Board *z = Board::open("zcui.work/zebu.work");
        Driver *d      = z->getDriver("counter_cosim");
        Signal *reset = d->getSignal("reset");
        Signal *load   = d->getSignal("d");
        Signal *data   = d->getSignal("data");
        Signal *ci     = d->getSignal("ci");
        Signal *pce    = d->getSignal("print_count_enable");

        d->connect();

        // enabling the DPI calls
        CCall::LoadDynamicLibrary(z, "dpi.so");
        CCall::SelectSamplingClocks(z, "clk");
        CCall::Start(z);

        z->init();
        // counter reset
        *reset = 1;
        d->run(2);
        // counter load
        *reset = 0;
        *load = 1;
        *data = 0xBABEFACE;
        d->run(1);
        // counter enable for 200000 cycles
        *ci = 1;
    }
}
```

Example

```

d->run(200000);
// Stopping the DPI calls
CCall::Stop(z);

d->disconnect();

z->close();

} catch (exception &x) {
    cerr << "Got exception " << x.what() << endl;
}
}/

```

The testbench is compiled with the g++ compiler into an executable file (tb):

```
$ g++ -o tb tb.cc -I$ZEBU_ROOT/include -L$ZEBU_ROOT/lib -lZebu
```

8.7.3.2 Proceeding With Runtime

Ensure the LD_LIBRARY_PATH environment variable is correctly set, and launch the testbench as displayed in the following code snippet:

```
$ export LD_LIBRARY_PATH=<path to dpi.so>:$LD_LIBRARY_PATH
$ ./tb
```

8.7.4 Controlling the Calls from zRun

When DPI calls implemented by the dynamic library (dpi.so) are controlled from **zRun**, the testbench (tb) only controls the co-simulation driver so a specific script is used for **zRun** (script_dpi.tcl).

8.7.4.1 Implementing the zRun Script

The following is an example of the script (`script_dpi.tcl`) that controls the DPI function calls:

```
# open the board
ZEBU_open

# load of the DPI dynamic library
ZEBU_Call_loadDynamicLibrary dpi.so

# selection of both edges of clk to sample the DPI calls
ZEBU_CCall_selectSamplingClocks clk

# Start of all the DPI calls
ZEBU_CCall_start

# run the clocks clk
ZEBU_Clock_enable clk 200000
while { [ZEBU_getStatus] == "open" && [ZEBU_Clock_getStatus clk] ==
"running" } { after 500 }

# stop dpi processing
ZEBU_CCall_stop

# close the board
ZEBU_close
```

Example

8.7.4.2 Implementing a Testbench for Co-Simulation

The source code of the testbench (tb.cc) is as follows:

```
#include <libZebu.hh>
using namespace ZEBU;

using namespace std;
int main()
{
    try {
        // Board init
        Board *z = Board::open("zcui.work/zebu.work");

        Driver *d      = z->getDriver("counter_cosim");
        Signal *reset = d->getSignal("reset");
        Signal *load   = d->getSignal("d");
        Signal *data   = d->getSignal("data");
        Signal *ci     = d->getSignal("ci");
        Signal *pce    = d->getSignal("print_count_enable");

        d->connect();

        z->init();
        // counter reset
        *reset = 1;
        d->run(2);
        // counter load
        *reset = 0;
```

```
*load = 1;
*data = 0xBABEFACE;
d->run(1);

// counter enable for 200000 cycles
*ci = 1;
d->run(200000);

d->disconnect();

z->close();

} catch (exception &x) {
    cerr << "Got exception " << x.what() << endl;
}
}
```

The testbench is compiled with g++ compiler into an executable file (tb):

```
$ g++ -o tb tb.cc -I$ZEBU_ROOT/include -L$ZEBU_ROOT/lib -lZebu
```

8.7.4.3 Proceeding With Runtime

The LD_LIBRARY_PATH environment variable is set before launching zRun to run both the testbench and the script that controls DPI calls:

```
$ export LD_LIBRARY_PATH=<path to dpi.so>:$LD_LIBRARY_PATH
$ zRun -testbench ./tb -do script_dpi.tcl
```

9 Power Aware Verification With ZeBu

Power Aware Verification with ZeBu is based on the industry-standard Unified Power Format (UPF) 2.0, which is described in the IEEE-1801 standard. UPF uses a Tcl dialect to specify a design's power intent.

Using UPF commands, you can specify the supply network, power switches, isolation, retention, and other power management aspects. The same set of power specification commands are used throughout the design, analysis, verification, and implementation flow.

This chapter discusses the following topics:

- *UPF Commands Supported by ZeBu*
- *ZeBu Power Aware Emulation Flow*
- *Emulation Runtime*
- *Limitations*
- *Investigating Power Bugs*
- *Troubleshooting*

9.1 UPF Commands Supported by ZeBu

UPF commands and options are parsed by VCS.

ZeBu supports the subset of UPF commands listed in the following table.

TABLE 15 Supported UPF Commands for ZeBu

UPF Command	Supported Options
associate_supply_set	-handle supply_set_handle
connect_supply_net	[-ports list]
create_power_domain	<ul style="list-style-type: none"> [-elements element_list] [-include_scope]
	<ul style="list-style-type: none"> [-supply {supply_set_handle [supply_set_ref]}]*
create_power_switch	<ul style="list-style-type: none"> -output_supply_port {port_name [supply_net_name]} {-input_supply_port {port_name [supply_net_name]} }* {-control_port {port_name [net_name]} }* {-on_state {state_name input_supply_port {boolean_function}} }* {-off_state {state_name {boolean_function}} }* [-domain domain_name]
create_supply_net	<ul style="list-style-type: none"> [-domain domain_name] [-reuse] [-resolve <>]
create_supply_port	<ul style="list-style-type: none"> [-domain domain_name] [-direction <>]
create_supply_set	<ul style="list-style-type: none"> [-function {func_name [net_name]}]* [-update]
load_upf upf_file_name	[-scope instance_name]

UPF Commands Supported by ZeBu

TABLE 15 Supported UPF Commands for ZeBu

UPF Command	Supported Options
name_format	<ul style="list-style-type: none"> [-isolation_prefix string] [-isolation_suffix string]
set_design_attributes	<ul style="list-style-type: none"> -models model_list [-attribute name value]*
set_design_top	
set_domain_supply_net	<ul style="list-style-type: none"> -primary_power_net supply_net_name -primary_ground_net supply_net_name
set_isolation	<ul style="list-style-type: none"> -domain ref_domain_name [-elements element_list] -source source_supply_ref -sink sink_supply_ref -applies_to [-isolation_power_net net_name] [-isolation_ground_net net_name] [-no_isolation] [-isolation_supply_set supply_set_list] [-isolation_signal signal_list <>] [-isolation_sense {<high low>}*] [-clamp_value {< >}*] [-location <>] [-diff_supply_only {TRUE FALSE}]
set_isolation_control	<ul style="list-style-type: none"> -domain domain_name -isolation_signal signal_name [-isolation_sense {high low}] [-location <>]

TABLE 15 Supported UPF Commands for ZeBu

UPF Command	Supported Options
set_port_attributes	<ul style="list-style-type: none"> [-ports {port_list}] [{-elements {element_list} <>}] [{-applies_to <>}] [{-receiver_supply supply_set_ref}] [{-driver_supply supply_set_ref}]
set_retention	<ul style="list-style-type: none"> -domain domain_name [{-elements element_list}] [{-retention_power_net net_name}] [{-retention_ground_net net_name}] [{-retention_supply_set ret_supply_set}] [{-save_signal {{logic_net <>}}} -restore_signal {{logic_net <>}}] [{-save_condition {{boolean_function}}}] [{-restore_condition {{boolean_function}}}]
set_retention_control	<ul style="list-style-type: none"> -domain domain_name -save_signal {{net_name <>}} -restore_signal {{net_name <>}}
set_scope	

Note

UPF commands not listed in this table are parsed and ignored without displaying error messages.

9.2 ZeBu Power Aware Emulation Flow

This section describes the ZeBu UPF compilation flow.

9.2.1 UPF Support in Unified Compile

The UC flow supports the same UPF syntax, UPF command support, and error messaging as VCS. The same UPF file is used for both simulation and emulation. When compiling for ZeBu, VCS parses and elaborates both the design and UPF files.

The ZeBu front-end compiler generates the EDIF files, containing design and power intent information. The EDIF files and Core Definition Files are processed by **zTopBuild** in the back-end compilation. The Xilinx Place and Route software generates the final bitstream files that are downloaded into an FPGA.

The following figure displays the UPF compilation flow.

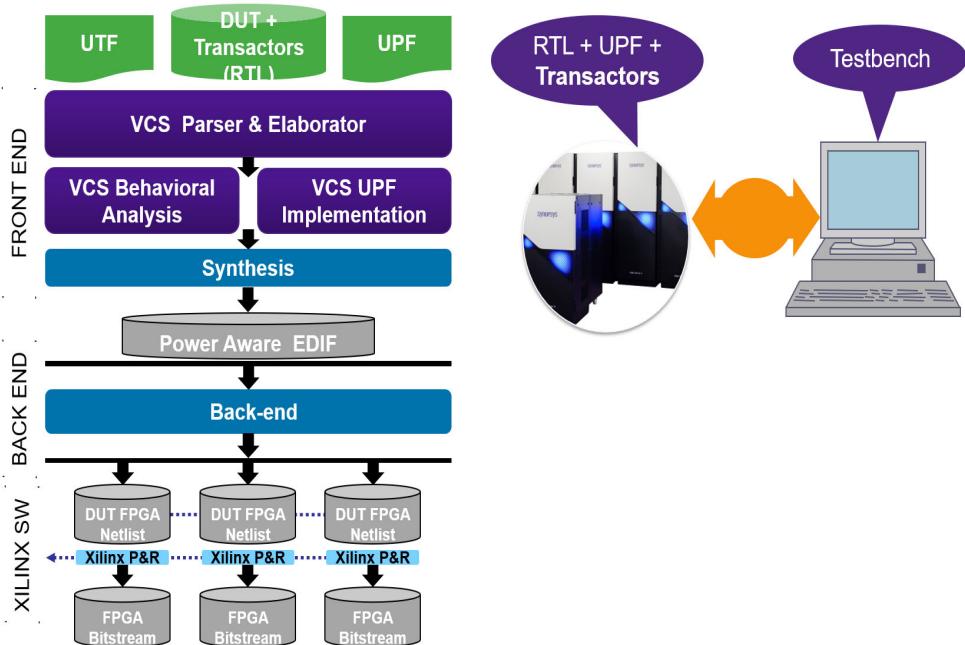


FIGURE 22. UPF Flow in UC

9.2.1.1 Benefits

The ZeBu UPF compilation flow provides the following benefits:

- Same semantics and analysis like VCS
- A similar set of supported UPF constructs like VCS
- Early error (RTL/UPF) flagging by VCS
- Precedence rules and implementation matching Synopsys cross-tools like, VCS, DC, and ICC
- Integrated debug capability (ZeBu/VCS/Verdi)

9.2.1.2 ZeBu Front-End Compilation

In the front-end VCS parses and creates the data-model comprising both functional intent and power intent. After VCS execution the design is synthesized into EDIF.

To synthesize for Power Aware Verification, you can choose **zFAST** or **zFAST Script Mode** in the Synthesizer selector of the RTL Group panel of **zCui**.

Note

Other synthesizers are not supported for Power Aware Verification. In addition, it is only possible to synthesize multi-RTL groups when the UPF Script is declared in the top group. UPF Scripts cannot be declared in more than one group.

The following figure displays the front-end compilation flow.

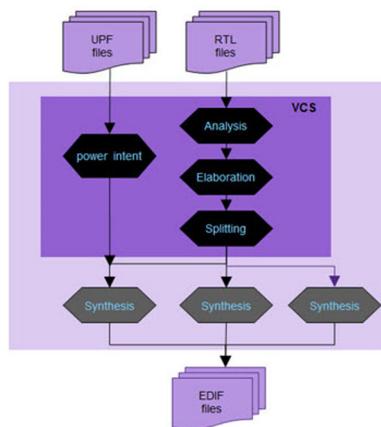
ZeBu Power Aware Emulation Flow

FIGURE 23. Front-End Compilation Flow

Design Files

Power-Aware Verification with ZeBu uses the same design source files as used for emulation without power-aware.

Power Management Script (UPF)

The UPF script describes the topology of the power network. This script is an input to the compiler.

ZeBu supports the following UPF (Unified Power Format) versions:

- UPF 1.0 and UPF 2.0 (IEEE 1801-2009 standard)
- Limited set of commands from UPF 2.1 (IEEE 1801-2013 standard)

ZeBu also supports both hierarchical and non-hierarchical UPF. In the UPF file, design hierarchical paths can be declared and applied using the usual Tcl rules.

9.2.1.3 ZeBu Back-End Compilation Flow

After front-end compilation, the next steps in ZeBu emulation are back-end compilation and FPGA Place and Route. The EDIF generated by front-end compilation and Core Definition Files are processed by **zTopBuild**. The Xilinx Place and Route software generates the final bitstream files that are downloaded into the FPGAs.

9.2.2 ZeBu Power Aware Compilation

The UPF file is specified in the VCS command line (or VCS script) with the **-upf** option:

```
% vcs -upf <filename.upf> <vcs_options> <design_files>
```

9.2.3 Corruption in ZeBu Power Aware Emulation Flow

ZeBu models corruption of shutdown domain by randomizing its outputs and its scrambling internal state elements.

Randomization

As long as a power domain is OFF, its output ports are continuously randomized with pseudo-random values, as illustrated in the following figure:

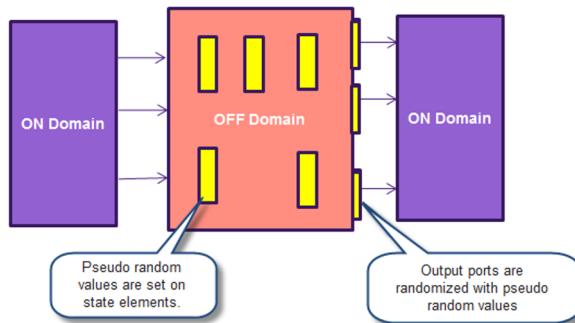


FIGURE 24. Corruption in ZeBu Power Aware Emulation Flow

9.2.3.1 Scrambling

When a power domain switches on or off, pseudo random values are injected into its internal state elements: registers, latches, and memories. This happens only during power switching - not continuously. Scrambled values are not injected into combinational logic.

9.2.4 UPF File Example

```
set_design_top top

create_power_domain TOP -elements {} -include_scope
create_power_domain VCC0 -elements {child0 adder0}
create_power_domain VCC1 -elements {child1 adder1}
```

```
create_power_domain VCC2 -elements {adder2}

# VCC
create_supply_port VCC_TOP_port -domain TOP
create_supply_net  VCC_TOP_net   -domain TOP
connect_supply_net VCC_TOP_net   -ports VCC_TOP_port

# VSS
create_supply_port VSS -domain TOP
create_supply_net  VSS_net     -domain TOP
connect_supply_net VSS_net     -ports VSS

## Switch output
create_supply_net  VCC0_SW   -domain VCC0
create_supply_net  VCC1_SW   -domain VCC1
create_supply_net  VCC2_SW   -domain VCC2

#####
## Set Domain supplies
#####
set_domain_supply_net TOP \
    -primary_power_net VCC_TOP_net \
    -primary_ground_net VSS_net

## Declare that all power sets have a common ground.
create_supply_set VCC0.primary -function {ground
TOP.primary.ground} -function { power VCC0_SW} -update
create_supply_set VCC1.primary -function {ground
TOP.primary.ground} -function { power VCC1_SW} -update
```

ZeBu Power Aware Emulation Flow

```
create_supply_set VCC2.primary -function {ground  
TOP.primary.ground} -function { power VCC2_SW} -update  
create_power_switch VCC0_SWITCH \  
-domain VCC0 \  
-input_supply_port {VCC_TOP_port VCC_TOP_net} \  
-output_supply_port {VCCU_SW VCC0_SW} \  
-control_port {ctrl_sig switch_ctrl_0_reg} \  
-on_state {VCCU_ON VCC_TOP_port {ctrl_sig} } \  
-off_state {VCCU_OFF {!ctrl_sig} }  
  
create_power_switch VCC1_SWITCH \  
-domain VCC1 \  
-input_supply_port {VCC_TOP_port VCC_TOP_net} \  
-output_supply_port {VCCG_SW VCC1_SW} \  
-control_port {ctrl_sig switch_ctrl_1_reg} \  
-on_state {VCCG_ON VCC_TOP_port {ctrl_sig} } \  
-off_state {VCCG_OFF {!ctrl_sig} }  
  
create_power_switch VCC2_SWITCH \  
-domain VCC2 \  
-input_supply_port {VCC_TOP_port VCC_TOP_net} \  
-output_supply_port {VCCG_SW VCC2_SW} \  
-control_port {ctrl_sig switch_ctrl_2_reg} \  
-on_state {VCCG_ON VCC_TOP_port {ctrl_sig} } \  
-off_state {VCCG_OFF {!ctrl_sig} }
```

```
#-----  
#           set isolation strategies  
#-----  
name_format -isolation_prefix "ISO_PREFIX_"  
  
set_isolation VCC0_isolation -domain VCC0 \  
  -isolation_power_net VCC_TOP_net \  
  -isolation_ground_net VSS_net \  
  -applies_to outputs \  
  -clamp_value 0  
  
set_isolation_control VCC0_isolation -domain VCC0 \  
  -isolation_signal switch_ctrl_0_reg \  
  -isolation_sense low \  
  -location self  
  
set_isolation VCC1_isolation -domain VCC1 \  
  -isolation_power_net VCC_TOP_net \  
  -isolation_ground_net VSS_net \  
  -clamp_value 1  
  
set_isolation_control VCC1_isolation -domain VCC1 \  
  -isolation_signal switch_ctrl_1_reg \  
  -isolation_sense low \  
  -location self  
  
set_isolation VCC2_isolation -domain VCC2 \  
  -isolation_power_net VCC_TOP_net \  
  -isolation_ground_net VSS_net
```

ZeBu Power Aware Emulation Flow

```
-isolation_ground_net VSS_net \
-clamp_value Z
set_isolation_control VCC2_isolation -domain VCC2 \
-isolation_signal switch_ctrl_2_reg \
-isolation_sense low \
-location self
#-----
#           set retention strategies
#-----
set_retention VCC0_retention \
-domain VCC0 \
-retention_power_net VCC_TOP_net \
-retention_ground_net VSS_net \
-save_signal      { save_sig_0_reg high } \
-restore_signal   { restore_sig_0_reg high } \
set_retention VCC1_retention \
-domain VCC1 \
-retention_power_net VCC_TOP_net \
-retention_ground_net VSS_net \
-save_signal      { save_sig_1_reg high } \
-restore_signal   { restore_sig_1_reg high } \
set_retention VCC2_retention \
-domain VCC2 \
-retention_power_net VCC_TOP_net \
-retention_ground_net VSS_net \
```

```

-save_signal      { save_sig_2_reg high } \
-restore_signal  { restore_sig_2_reg high } \

#-----
#                               tools option set
#-----

set_design_attributes -attribute {SNPS_reinit TRUE}

```

9.2.5 Low Power Optimization and Reporting Commands

ZeBu back-end provides multiple low power optimization and reporting commands. You can enable or disable these commands depending on your low power verification requirements. This behavior is specified in the `config_upf` command using **Build System Parameters > Advanced Command File (Advanced > Custom User Files panel, System workspace)** option in `zCui`:

```

config_upf [-boundary_output <BOUNDARY_OUTPUT>]
# [-const_propagate_thru_iso] [-constant_vector_port] [-firmware_lib <FIRMWARE_LIB>] [-hier_sep <HIER_SEP>]
# [-ignore_buffers <IGNORE_BUFFERS>] [-pass_through_const <PASS_THROUGH_CONST>] [-reg_init <REG_INIT>]
# [-share_randomizers <SHARE_RANDOMIZERS>] [-tristate_boundary_output <TRISTATE_BOUNDARY_OUTPUT>]
# [-upf_report_inferred_isolation] [-upf_report_inferred_retention] [-upf_report_spa_srsn]

```

Note

The following options must be set to YES or NO to enable or disable them:

- ❑ `ignore_buffers`
- ❑ `-pass_through_const`
- ❑ `-share_randomizers`

The following table lists the `config_upf` command's options:

TABLE 16 config_upf Command Options

Option	Description
<code>-ignore_buffers</code>	Ignores simple continuous assignments on power domain boundary for corruption. By default, this option is enabled.
<code>-pass_through_const</code>	Does not corrupt constants on the power domain boundary. By default, this option is enabled.
<code>-shared_randomizers</code>	Allows sharing of one randomizer for four ports. The randomizer uses a 4-bit serial register. This optimal sharing of randomizer can reduce the randomizer overhead by 75%. Not having a randomizer for each boundary port may reduce the coverage (and hence quality) of corruption. By default, this option is not enabled.
<code>-boundary_output</code>	Defines the value to be driven on boundary output ports of powered-down domains. Possible values are: <ul style="list-style-type: none"> • <code>pseudo_random</code>: Pseudo-random values are continuously driven on the port (default). • <code>0</code>: is driven on the ports. • <code>1</code>: is driven on the ports. • <code>REG</code>: Configurable value to drive on the ports. The value is configured using the <code>-reg_init</code> option.
<code>-constant_vector_port</code>	Specifies to never power off a 0 constant mapped to instance ports.
<code>-firmware_lib</code>	Defines the name of the firmware library provided to the system-level compiler without using the hierarchical path.
<code>-hier_sep</code>	Defines the hierarchical separator to use when parsing requests.

TABLE 16 config_upf Command Options

Option	Description
-reg_init	If -boundary_output is set to REG, you can use this option to define the value to be driven on powered-down domains. Possible values are: <ul style="list-style-type: none"> • 0: (Default value). This value can be changed during emulation. • 1: This value can be changed during emulation.
-upf_report_inferred_isolation	Creates a report listing all isolation cells.
-upf_report_inferred_retention	Creates a report listing all retention cells.
-upf_report_spa_srsn	Creates a report listing ports to which the set_related_supply_net command is applied.
-verbose_retention	Displays all retention registers.

Note

Optimization commands have a direct impact on the hardware cost.

9.3 Emulation Runtime

A design compiled with UPF can be emulated at runtime with the same test environment when no Power Aware Verification is required.

9.3.1 C++ Interface

The C++ API methods are provided in the `PowerMgt` class, which is described in the `$ZEBU_ROOT/include/PowerMgt.hh` header file. This API is included in the `ZEBU` namespace.

For easier implementation, `PowerMgt.hh` header file is automatically available when including the `libZebu.hh` header file.

The `PowerMgt` APIs must be called during emulation runtime in the following order:

1. Anytime during emulation, call
`PowerMgt::isPowerManagementEnabled()` to find whether power aware verification is enabled during runtime.
2. To initialize power aware verification during runtime, call `PowerMgt::init`.
3. To enable power aware verification during runtime, call `PowerMgt::enable`.
4. The following APIs must be called before calling `PowerMgt::enable`.
 - a. `PowerMgt::initializeRandomizer`
 - b. `PowerMgt::setForceMode`
 - c. `PowerMgt::setPollingSleepTime`
5. The following APIs must be called after calling `PowerMgt::enable`:
 - a. `PowerMgt::getPowerDomainState`
 - b. `PowerMgt::releasePowerDomain`
 - c. `PowerMgt::getLastTriggeredDomain`
 - d. `PowerMgt::getSupplyState`
 - e. `PowerMgt::enableIsolation`
 - f. `PowerMgt::enableIsolationStrategy`
 - g. `PowerMgt::isIsolationStrategyEnabled`
 - h. `PowerMgt::enableRetention`

- i. PowerMgt::enableRetentionStrategy
 - j. PowerMgt::isRetentionStrategyEnabled
 - k. PowerMgt::setDomainOnPreCallback
 - l. PowerMgt::setDomainOnPostCallback
 - m. PowerMgt::setDomainOffPreCallback
 - n. PowerMgt::setDomainOffPostCallback
 - o. PowerMgt::enableSRSN
 - p. PowerMgt::getListOfDomains
 - q. PowerMgt::getListOfIsolationStrategies
 - r. PowerMgt::getListOfRetentionStrategies
 - s. PowerMgt::supplyOn
 - t. PowerMgt::StartWriteBack/FlushWriteBack
6. The following APIs must be called after PowerMgt::init and after or before PowerMgt::enable:
- a. PowerMgt::supplyOff
 - b. PowerMgt::setCorruptionState
 - c. PowerMgt::setScramblingState
 - d. PowerMgt::getPowerDomainName
 - e. PowerMgt::setMultiPowerDomainState
 - f. PowerMgt::setPowerDomainState
 - g. PowerMgt::setPowerDomainOn
 - h. PowerMgt::setPowerDomainOff
 - i. PowerMgt::disableIsolation
 - j. PowerMgt::disableIsolationStrategy
 - k. PowerMgt::disableRetention
 - l. PowerMgt::disableRetentionStrategy
 - m. PowerMgt::disableSRSN

This API provides the following methods:

- To start emulation runtime with Power Aware Verification. See [Starting Emulation Runtime with Power Aware Verification](#).
- To initialize the environment. See [Initializing the Environment](#).
- To get information about the design. See [getListOfDomains](#).
- To get information about retention. See [Managing Retention Strategies](#).
- To control the power domains. See [Controlling Power Domains](#).
- To declare user callbacks. See [Declaring User Callbacks](#).
- To manage forces and injections. See [Managing Forces and Injections](#).

For more information on featured example of a testbench for Power Aware Verification, see [C++ Example](#).

Note

All methods described in this section throw an exception if the pointer to the ZeBu board is incorrect. For any other failure, these methods return a Boolean value:

- *true for a correct processing.*
- *false in case of error.*

The following table lists the C++ API methods to control power aware verification. These methods are described in this section.

TABLE 17 C++ API to Control Power Aware Verification: `PowerMgt` class

C++ Methods	Description
<code>Init</code>	Starts Power Aware Verification.
<code>Enable</code>	Enables Power Aware Verification.
<code>initializeRandomizer</code>	Initializes the generator that later applies values to registers, ports and memories in one or all power domains.
<code>performRandomizer</code>	Set all elements of the design, whatever their power domain, to pseudo-random values, or 0 or 1 according to the mode selected in <code>initializeRandomizer</code>

TABLE 17 C++ API to Control Power Aware Verification: `PowerMgt` class

C++ Methods	Description
<code>performRandomizerDomain</code>	Set all elements of a specific domain to pseudo-random values, or 0 or 1 according to the mode selected in <code>initializeRandomizer</code> .
<code>isPowerManagementEnabled</code>	Checks if the design is compiled to support Power-Aware Verification.
<code>getListOfDomains</code>	Returns a list of all power domains declared in the Power Management (UPF) Script.
<code>getPowerDomainState</code>	Returns the state of a power domain.
<code>getLastTriggeredDomain</code>	Returns the list of power domains that changed state (ON→OFF or OFF→ON).
<code>enableRetentionStrategy</code>	Enables a retention strategy.
<code>disableRetentionStrategy</code>	Disables a retention strategy.
<code>isRetentionStrategyEnabled</code>	Returns information about the retention strategy.
<code>getListOfRetentionStrategies</code>	Returns the list of available retention strategies
<code>setPowerDomainOn</code>	Switches a power domain ON.
<code>setPowerDomainOff</code>	Switches a power domain OFF.
<code>setPowerDomainState</code>	Switches a power domain to the given state.
<code>releasePowerDomain</code>	The domain is no longer controlled from the testbench, but by the design itself.
<code>setPowerDomainOffCallback</code>	Designates a replacement function for the default behavior of the software when a power domain is switched OFF.
<code>setPowerDomainOnCallback</code>	Designates a replacement function for the default behavior of the software when a power domain is switched ON.
<code>setForceMode</code>	Defines the behavior of forces and injections regarding power domain states.

Note

For legibility purposes, <method_name> is often used in this chapter in place of PowerMgt::<method_name>.

9.3.1.1 Starting Emulation Runtime with Power Aware Verification

Power Aware Verification must be started with the following methods in the following order:

1. `init(Board*);`
2. `enable(Board*);`

The `PowerMgt::init` method must be called after the `Board::open` and before the `Board::init` methods.

Note

If your design is compiled with a Power Management script, but you do not want to enable power aware verification during runtime, the init and enable methods can be omitted.

For example:

```
Board* zebu = Board::open(workdir);
PowerMgt::init(zebu);
zebu->Board::init();
PowerMgt::enable(zebu);
```

9.3.1.2 Initializing the Environment

The environment is initialized to define the randomizer mode. This section describes the commands to initialize the environment.

initializeRandomizer

This method initializes the random generator that applies values to registers, ports, and memories in shutdown power domains. Three different modes are available:

pseudo-random values (to simulate X values), all-0 values, or all-1 values.

```
bool initializeRandomizer (Board *board, const string &mode, const  
unsigned int seedValue) throw(std::exception);
```

where,

- board: Pointer to the ZeBu::Board object.
- mode: Type of randomization; the following values are supported:
 - MODE_ZERO: Forces all elements to value 0.
 - MODE_ONE: Forces all elements to value 1.
 - MODE_RND: Forces all elements to a pseudo-random value.
 - seedValue: Integer value to initialize the pseudo-random generator.

Note

All-0 and all-1 values are only applicable to state elements (registers, latches, and memories). They do not apply to the power-domain's interface ports.

performRandomizer

All elements in the design, irrespective of their power domain, are forced to 0, 1, or a pseudo-random value according to the mode specified in initializeRandomizer. If the power domain is set to ON or is controlled by the design, the performRandomizer has no effect.

```
bool performRandomizer (Board *board) throw(std::exception);
```

where, board is the pointer to the ZeBu::Board object.

Note

All-0 and all-1 values are only applicable to state elements (registers, latches, and memories). They do not apply to the power-domain's interface ports.

performRandomizerDomain

All elements of a domain are set to pseudo-random values or 0 or 1 according to the

mode selected in `initializeRandomizer`. If the power domain is set to ON or is controlled by the design, the `performRandomizer` method has no effect.

```
bool performRandomizer (Board *board, const std::string
&domainname) throw(std::exception);
```

where,

- `board`: Pointer to the `ZeBu::Board` object.
- `domainname`: Name of the domain (reference to object).

Note

All-0 and all-1 values are only applicable to state elements within the domain such as registers, latches and memories.

At runtime, you can get information about the power management status (whether it is enabled or not), current state of power domains, and so on. This section describes the methods to get this information at runtime.

`isPowerManagementEnabled`

This method checks whether the design is compiled to support Power Aware Verification.

```
bool isPowerManagementEnabled (Board *board, unsigned int
&enabled) throw(std::exception);
```

where,

- `board`: Pointer to the `ZeBu::Board` object.
- `enabled`: Capability to run with power-aware verification (reference to object).

`getListOfDomains`

This method returns a list of all power domains created by the Power Management Script.

```
bool getListOfDomains (Board *board, std::set<std::string>
&domains)
```

where,

- board: Pointer to the ZeBu::Board object.
- domains: List of domains names (reference to board).

getPowerDomainState

This method provides the state of a power domain.

```
bool getPowerDomainState (Board *board, const std::string  
&domainname, unsigned int &state) throw(std::exception);
```

where,

- board: Pointer to the ZeBu::Board object.
- domainname: Name of the power domain declared in the Power Management Script
- state: Pointer to the current state of domainname; 1 stands for ON and 0 stands for OFF.

getLastTriggeredDomain

This method retrieves the list of power domains for which the state changed (ON→OFF or OFF→ON).

```
bool getLastTriggeredDomain (Board *board, std::set<std::string>  
&triggerChangedDomain);
```

where,

- board: Pointer to the ZeBu::Board object.
- triggerChangedDomain: Names of domains whose power state changed (reference to object).

9.3.1.3 Managing Retention Strategies

At runtime, you can turn retention strategies on or off for analysis or performance tuning. This section describes the methods to manage retention strategies.

enableRetentionStrategy

This method enables the retention strategy.

```
bool enableRetentionStrategy (Board *board, const std::string  
&strategyName) throw(std::exception);
```

where,

- board: Pointer to the ZeBu::Board object.
- strategyName: Name of the retention strategy.

disableRetentionStrategy

This method disables the retention strategy.

```
bool disableRetentionStrategy (Board *board, const std::string  
&strategyName) throw(std::exception);
```

where,

- board: Pointer to the ZeBu::Board object.
- strategyName: Name of the retention strategy.

isRetentionStrategyEnabled

This method retrieves information about the retention strategy.

```
bool isRetentionStrategyEnabled (Board *board, const std::string  
&strategyName, bool &enabled) throw(std::exception);
```

where,

- board: Pointer to the ZeBu::Board object.
- strategyName: Name of the retention strategy.
- enabled: Capability to see if a retention strategy is enabled.

getListOfRetentionStrategies

This method retrieves the list of available retention strategies.

```
bool getListOfRetentionStrategies (Board *board,  
std::set<std::string> &strategies) throw(std::exception);
```

where,

- board: Pointer to the ZeBu::Board object.
- strategies: List of retention strategies sorted in alphabetical order.

9.3.1.4 Controlling Power Domains

The methods described in this section provide runtime control for turning power domains ON or OFF. This is another way of testing the low power behavior of the design.

setPowerDomainOn

This method switches a power domain ON (the power domain is no longer controlled by the design).

```
bool setPowerDomainOn (Board *board, const std::string  
&domainname) throw(std::exception);
```

where,

- board: Pointer to the ZeBu::Board object
- domainname: Name of the domain (reference to object)

setPowerDomainOff

This method switches a power domain OFF (the power domain is no longer controlled by the design).

```
bool setPowerDomainOff (Board *board, const std::string  
&domainname) throw(std::exception);
```

where,

- board: Pointer to the ZeBu::Board object
- domainname: Name of the domain (reference to object)

setPowerDomainState

This method switches a power domain to the given state (the power domain is no longer controlled by the design).

```
bool setPowerDomainState (Board *board, const std::string
&domainname, const unsigned int state) throw(std::exception);
```

where,

- board: Pointer to the ZeBu::Board object
- domainname: Name of the domain (reference to object)
- state: Integer value that specifies the state (0 for OFF, any non-zero value for ON).

Note

- setPowerDomainState (board, domainname, 1) is equivalent to setPowerDomainOn(board, domainname).
- setPowerDomainState (board, domainname, 0) is equivalent to setPowerDomainOff(board, domainname).

releasePowerDomain

This method designates the domain to be controlled by design, and no longer by the testbench.

```
bool releasePowerDomain (Board *board, const std::string
&domainname) throw(std::exception);
```

where,

- board: Pointer to the ZeBu::Board object
- domainname: Name of the domain (reference to object)

9.3.1.5 Declaring User Callbacks

These methods enable you to override the default behavior of a power domain when it changes state from ON to OFF or vice-versa.

setPowerDomainOffCallback

By default, ZeBu sets all registers/memories and all ports of a power domain to pseudo-random values when the domain is switched OFF.

This method is used to specify a function that overrides the default behavior when a power domain is switched OFF.

Note *Pseudo-random values are applied to ports with a user-callback as well. The user-callback only applies to registers and memories.*

```
bool setPowerDomainOffCallback (Board *board, const std::string &domainname, void (*callback) (void *), void *userData)  
throw(std::exception);
```

where,

- board: Pointer to the ZeBu::Board object
- domainname: Name of the domain (reference to object)
- callback: Pointer to the callback function declared by the user
- userData: Pointer to the data structure transmitted to the user callback

The default behavior applies when the `setPowerDomainOffCallback()` method is not called by the testbench or when it is called with as null function, as `setPowerDomainOffCallback(board, NULL, NULL)`.

setPowerDomainOnCallback

By default, ZeBu sets all registers and memories of a power domain to pseudo-random values to prevent power-on to restart with a previous (and valid) state.

This method is used to specify a function that overrides the default behavior when a

power domain is switched ON.

```
bool setPowerDomainOnCallback (Board *board, const std::string
&domainname, void (*callback) (void *), void *userData)
throw(std::exception);
```

where,

- board: Pointer to the ZeBu::Board object
- domainname: Name of the domain (reference to object)
- callback: Pointer to the callback function declared by the user
- userData: Pointer to the data structure transmitted to the user callback

The default behavior applies when the `setPowerDomainOnCallback()` is not called by the testbench or when it is called with a null function as `setPowerDomainOnCallback(board, NULL, NULL)`.

9.3.1.6 Managing Forces and Injections

Forcing a signal means to set a user-defined value at runtime until explicitly released. Injecting a signal means to set a user-defined value at runtime, which is overwritten by the design at the next write operation. The `setForceMode` method defines the behavior of forces and injections of power domain states:

- If set to 0 (default mode), the power domain state is considered when applying forces and injections.
- If set to 1, forces and injections are applied without considering the power domain states.

```
bool setForceMode (Board *board, unsigned int mode)
throw(std::exception);
```

where, `board` is the pointer to the ZeBu::Board object.

9.3.1.7 C++ Example

The following example displays a C++ testbench for Power Aware Verification after initialization of the ZeBu board:

```
using namespace ZEBU;

// Initialize generator of pseudo-random values
PowerMgt::initializeRandomizer(zebu, "MODE_RND", 42);

// Run the testbench with power methods activated
top_ccosim->run(cycle);

// force the signal top.regs.d2 to the value "2"
unsigned int value2 = 0;
Signal::Force(zebu, "top.regs.d2", &value2);
[...]
Signal::Release(zebu, "top.regs.d3");
top_ccosim->run(cycle);
// display the power domains whose state has changed
std::set<std::string> domains;

PowerMgt::getLastTriggeredDomain(_zebu, domains);

for (std::set<std::string>::const_iterator dit = domains.begin();
     dit != domains.end(); ++dit)
{
    const std::string& domain = *dit;
    std::cerr << "Domain " << domain << " has switched" << std::endl;
}
```

9.3.2 C Language Interface

For testbenches written in C, equivalent functions are available to match the functional description of the C++ methods described in [C++ Interface](#). The corresponding header file is `ZeBu_PowerMgt.h`.

Except when stated otherwise in the description in the header file, all functions of the C API return:

- 0 indicates success.
- A non-zero value indicates an error.

TABLE 18 C++ API / C API Equivalence

C++ API	Plain C API
initializeRandomizer	<code>ZEBU_PowerMgt_initializeRandomizer</code>
setPowerDomainOnCallback	<code>ZEBU_PowerMgt_setPowerDomainOnCallback</code>
setPowerDomainOffCallback	<code>ZEBU_PowerMgt_setPowerDomainOffCallback</code>
setPowerDomainOn	<code>ZEBU_PowerMgt_setPowerDomainOn</code>
setPowerDomainOff	<code>ZEBU_PowerMgt_setPowerDomainOff</code>
setPowerDomainState	<code>ZEBU_PowerMgt_setPowerDomainState</code>
releasePowerDomain	<code>ZEBU_PowerMgt_releasePowerDomain</code>
performRandomizer	<code>ZEBU_PowerMgt_performRandomizer</code>
performRandomizerDomain	<code>ZEBU_PowerMgt_performRandomizerDomain</code>
isPowerManagementEnabled	<code>ZEBU_PowerMgt_isPowerManagementEnabled</code>
getPowerDomainState	<code>ZEBU_PowerMgt_getPowerDomainState</code>
getLastTriggeredDomain	<code>ZEBU_PowerMgt_getLastTriggeredDomain</code>
getListofDomains	<code>ZEBU_PowerMgt_getListOfDomains</code>
init	<code>ZEBU_PowerMgt_init</code>
setTime0CorruptionFilter	<code>ZEBU_PowerMgt_setTime0CorruptionFilter</code>
setMode	<code>ZEBU_PowerMgt_setMode</code>
Enable	<code>ZEBU_PowerMgt_enable</code>

TABLE 18 C++ API / C API Equivalence

C++ API	Plain C API
enableRetentionStrategy	ZEBU_PowerMgt_enableRetentionStrategy
disableRetentionStrategy	ZEBU_PowerMgt_disableRetentionStrategy
isRetentionStrategyEnabled	ZEBU_PowerMgt_isRetentionStrategyEnabled
getListOfRetentionStrategies	ZEBU_PowerMgt_getListOfRetentionStrategies
setForceMode	ZEBU_PowerMgt_setForceMode

For example:

The following example displays a C testbench for Power Aware Verification after initialization of the ZeBu board:

```
// Initialize power engine with parameters for pseudo-random number
// generator
ZEBU_PowerMgt_initializeRandomizer(zebu, "MODE_RND", 42);

// Run the testbench with power methods activated
ZEBU_Driver_run(top_ccosim, cycle);

// display the power domains whose state has changed
char **triggerChangedDomain = NULL;
unsigned int cnt = ZEBU_PowerMgt_getLastTriggeredDomain(ZEBU_Board
*board, &triggerChangedDomain)

for (unsigned int i = 0; i < cnt; ++i)
{
    printf("domain %s has triggered.", triggerChangedDomain[i]);
}
```

Emulation Runtime

The following example displays a C++ testbench for `supply_on/supply_off` calls after initialization of the ZeBu board:

```
#include "PowerMgt.hh"

int main(int argc, char** argv) {

    PowerMgt::init(zebu);
    // zebu init
    zebu->init();
    PowerMgt::enable(zebu);
    // Initialize generator of pseudo-random values
    PowerMgt::initializeRandomizer (zebu, "MODE_RND", 42);
    ...
    ZEBU::Signal *din      = top_ccosim-
>getSignal("zebuAutoTB.d");
    ZEBU::Signal *rst      = top_ccosim-
>getSignal("zebuAutoTB.rst");
    ZEBU::Signal *ctrl_sig = top_ccosim-
>getSignal("zebuAutoTB.ctrl_sig");
    //design signals
    *ctrl_sig = b0;
    *rst = b1;
    *din = b0;
    // Run for 8 cycles
    top_ccosim->run(8);
    *din = b1;
    top_ccosim->run(4);
    PowerMgt::supplyOff ("zebuAutoTB.top.VDD");
    *din = b0;
```

```
    top_ccosim->run(10);
    // supply_on call on UPF port
    PowerMgt::supplyOn("zebuAutoTB.top.VDD", 1.8);
    top_ccosim->run(4);
    *din = b1;

    top_ccosim->run(10);
    zebu->close();
    ...
}
```

Controlling Cycle 0 Corruption

The C ZEBU_PowerMgt_setTime0CorruptionFilter API is added to control cycle 0 corruption.

Note

ZEBU_PowerMgt_setTime0CorruptionFilter must be called before ZEBU_PowerMgt_enable. Otherwise, all triggers are enabled by default.

The usage of this API is as follows:

```
void ZEBU_PowerMgt_setTime0CorruptionFilter(int
timeZeroCorruptionFilter)
param[in] timeZeroCorruptionFilter
```

Where, param[in] is one of the following:

- 0 - ignore all time 0 triggers
- 1 - allow only triggers from supplyOn/Off calls
- 2 - allow only triggers from PowerMgt::enable

- 3 - allow all triggers

The C API enables the ability to detect the design-driven power domain switches as follows:

```
static bool enable(Board *board, int timeZeroCorruptionFilter = 3)
throw(std::exception);
```

By default, `timeZeroCorruptionFilter` is set to 3 to all triggers. That is, all power domains are set to OFF using the `supplyoff` command or default power domain status starts corruption from cycle 0.

Setting Low Power Mode At Runtime

You can explicitly set low power modes at runtime using the C++ API or C API, `setMode`.

Note

The `setMode` API must be called before `Board::open`. `powerDB` is loaded depending on the low power mode.

The usage of this API helps to reduce initialization time during the UPF model load.

If you are using any other old API to turn ON all power domains such as `setPowerDomainon`/`setPowerDomainState` or `setMultiPowerDomainState`, recommend you to remove them and use `setMode` API, which is more efficient to turn ON all power domains.

If you turn ON selective power domains, select the `LP_POWER_AWARE` mode that is the default mode when `PowerMgt::enable` is called and then use other API to selectively to turn ON any selective power domains.

C++ API

```
typedef enum {
    LP_POWER_AWARE = 0,
    LP_ALWAYS_ON,
    LP_NO_HARM,
    LP_UNSET_MODE
} LowPowerMode;
```

```
static void setMode(LowPowerMode mode);
```

C API

```
void ZEBU_PowerMgt_setMode(int mode);
```

Where, mode can be set to one of the following:

- 0 - Power Aware mode
- 1 - AON mode
- 2 - No-Harm mode
- 3 - Default mode value, user cant set

9.4 Limitations

The following features are not supported by the current version of ZeBu Power-Aware Verification with:

- Voltage-level (value) shifting - only ON/OFF is supported.
- Force and injection on retained domains when registers are restored.
- Randomization on registers in case of incorrect configuration of the retention control signals.
- Power-related attributes in the design source code.
- UPF commands related to power state tables are parsed and ignored.
- Retention on memory mapped as **zMem** (a tool that infers the synthesizable memory in ZeBu).
- Definitions of tuples (triplets of isolation supply, isolation signal and sense and isolation clamp arguments in an isolation strategy).
- Mapping the design on FPGAs connected to Reduced Latency DRAM (RLDRAM).

9.5 Investigating Power Bugs

ZeBu provides several means to investigate issues with Power Aware Verification typically found when the design controls the power domains specified in UPF.

9.5.1 Checking Isolation between Power Domains

When an unexpected behavior is observed, the issue may be caused by isolation between power domains. In particular, the pseudo-random values applied to the output ports of an OFF power domain may cause unexpected values on other ON domains not properly isolated. For example, the isolation control is not enabled, or the isolation supply is OFF.

These unexpected values must be investigated upstream to locate the corresponding power domain that is switched OFF.

Once a particular power domain is identified as the root cause for the isolation problem, it can be manually switched OFF using the ZeBu API and then verify the inputs of the other ON domains.

The ZeBu C++ API for power aware verification offers specific methods to manually control the activation of power domains, see [Controlling Power Domains](#).

C++ Method	Description
setPowerDomainOn	Switches a power domain ON (the power domain is no longer controlled by the design).
setPowerDomainOff	Switches a power domain OFF (the power domain is no longer controlled by the design).
setPowerDomainState	Switches a power domain to the given state (the power domain is no longer controlled by the design).
releasePowerDomain	The domain is no longer controlled from the testbench but by the design itself.

9.5.2 Examining the Power State of Design Instances

You can check whether any design instance is ON or OFF by connecting a dynamic-probe to the ZEBU_POWER_ON signal using `zSelectProbes/zDbPostProc`. This signal is automatically available for any design instance compiled for power aware verification:

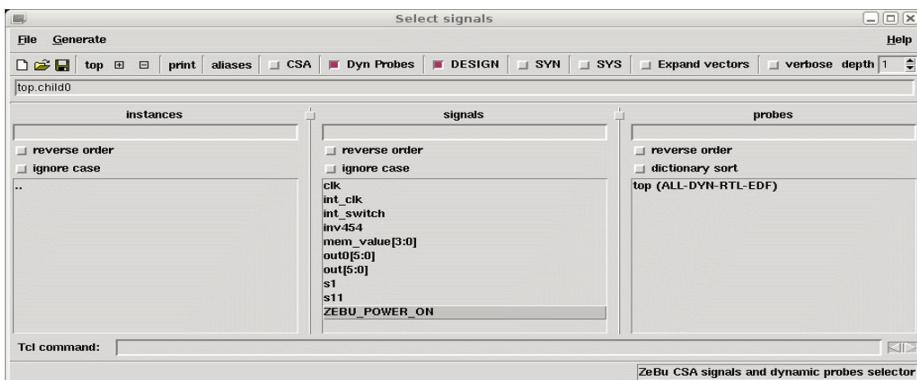


FIGURE 25. ZEBU_POWER_ON Signal in zSelectProbes

9.5.3 Examining the State of Power Domains

The ZeBu C++ API for Power-Aware Verification offers specific methods to check the properties and state of a power domain, see [Controlling Power Domains](#).

C++ Method	Description
<code>isPowerManagementEnabled</code>	Checks whether the design has been compiled for Power Aware Verification.
<code>getListOfDomains</code>	Returns a list of all power domains declared in the UPF Script.
<code>getPowerDomainState</code>	Returns the (ON/OFF) state of a power domain.
<code>getLastTriggeredDomain</code>	Returns the list of power domains whose state changed (ON→OFF or OFF→ON).

Investigating Power Bugs

In addition, messages are reported in the runtime logs indicating when power domains change state (ON/OFF).

9.5.4 Retention Control Signals

Retention strategies can be switched ON/OFF by the testbench through the `enableRetentionStrategy` and `disableRetentionStrategy` methods, see [Controlling Power Domains](#).

A dynamic-probe can be connected to a retention strategy using `zSelectProbes`. The corresponding instances and condition signals are displayed in the following figures:

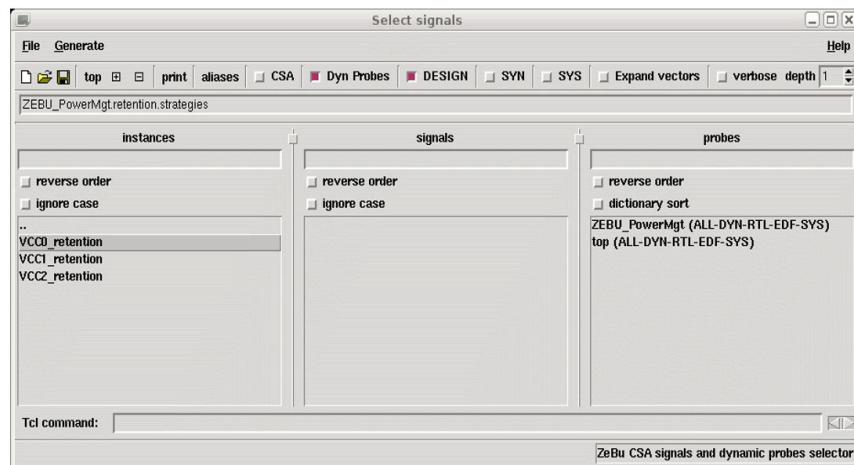


FIGURE 26. Retention Strategy Instances in `zSelectProbes`

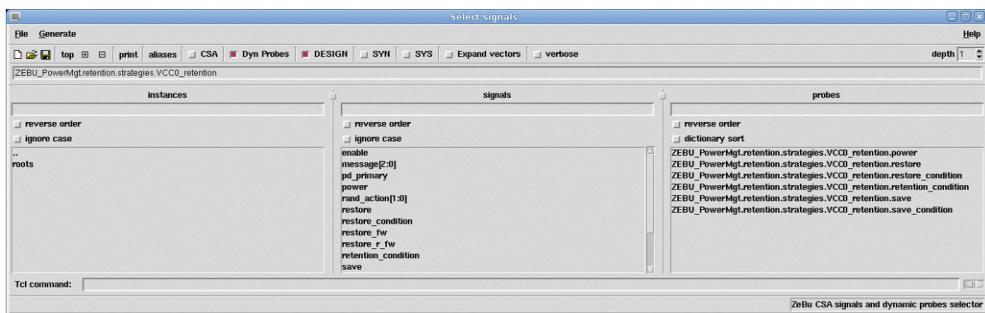


FIGURE 27. Retention Strategy Signals in zSelectProbes

9.6 Troubleshooting

The following sections display the error messages reported in the log file of a Power Aware Verification testbench, and solutions to solve each problem.

- *Incorrect Order when Calling PowerMgt::init*
- *Deprecated Emulation Runtime Control Methods*
- *Failure when Calling any Power Aware Method*
- *Failure When Calling any Retention-Related Method*

9.6.1 Incorrect Order when Calling PowerMgt::init

If you do not call the `PowerMgt::init` method before the `PowerMgt::enable` method, the testbench fails with the following error message:

```
-- ZeBu : tb : ERROR : ZHW1027E : Call 'PowerMgt::init' before any
call to 'PowerMgt::enable'
```

9.6.2 Deprecated Emulation Runtime Control Methods

The following methods to control emulation runtime for Power Aware Verification are now deprecated.

- `waitDriver` (C equivalent: `ZEBU_PowerMgt_waitDriver` or `ZEBU_PowerMgt_waitDriverEX`)
- `runDriver` (C equivalent: `ZEBU_PowerMgt_runDriver`)

Contact Synopsys support at zebu_support@synopsys.com for further details, if your testbench was previously functional, but now fails with one of the following error messages:

```
-- ZeBu : tb : ERROR : ZHW1032E : 'waitDriver' call failed: Power Awareness is not in co-simulation mode
```

Or

```
-- ZeBu : tb : ERROR : ZHW1032E : 'runDriver' call failed: Power Awareness is not in co-simulation mode
```

9.6.3 Failure when Calling any Power Aware Method

If you call any of the methods related to the Power Aware Verification feature (see [C++ Interface](#)) without compiling the design with this feature, the testbench fails with one of the following error messages:

```
-- ZeBu : tb : ERROR : ZHW1031E : 'init' call failed: Power Awareness feature not available
```

Or

```
-- ZeBu : tb : ERROR : ZHW1031E : 'initializeRandomizer' call failed: Power Awareness feature not available
```

It is mandatory that you compile your design with the Power Aware Verification feature before attempting to use any of its features.

9.6.4 Failure When Calling any Retention-Related Method

If you call any of the methods related to retention strategies without prior definition of the retention strategy in your UPF file, the testbench fails with one of the following error messages:

```
-- ZeBu: tb: ERROR : ZHW1146E : 'getStrategies' call failed. Power  
Retention is not available
```

Or

```
-- ZeBu: tb: ERROR : ZHW1143E : 'isStrategyEnabled' call failed.  
Power Retention is not available
```

Your UPF file must define the given retention strategies in order to call any retention-related method on them.

10 Compiling ZeBu Transactors

ZeBu transactors are a family of protocol-specific transaction-based verification solutions that allow you to quickly build a complete system-level test environment for your system-on-chip (SoC) designs. ZeBu transactors support common protocols and standards, such as PCI Express 3.0, AMBA , USB, MIPI CSI-2 and MIPI DSI, I2C, I2S, Gigabit Ethernet and 10 Gigabit Ethernet, Digital Video, and JTAG.

These transactors include Bus Functional Models (BFMs) that are mapped into the ZeBu RTB hardware. These BFMs provide maximum performance and ensure that a transactor is synchronized to an emulated design.

Each transactor includes C/C++ APIs to quickly create testbenches and drivers that generate real-world traffic, or to link to virtual platforms. ZeBu transactors are complemented by the ZEMI-3 transactor compiler, so you can generate your own ZeBu-compatible custom transactors. ZeBu transactors are fully compatible with ZeBu's interactive design debugging, supporting single-stepping of the clocks, and waveform dumping with Dynamic-Probes.

This chapter describes how you can compile a ZeBu transactor and discusses the following topics:

- [*Instantiating a Transactor*](#)
- [*Compiling a Transactor*](#)
- [*Compiling a Transactor Example*](#)
- [*Guidelines for Mapping Transactors in a Multi-RTB Environment*](#)

10.1 Instantiating a Transactor

To instantiate a transactor wrapper, perform the following steps:

1. Connect the controlled clock (.xtor_cclock0) generated by the zceiClockPort as an input to the transactor.

For example, if you are using a CGMAC transactor, the ref_clk_3125 clock must be connected to the input .xtor_cclock0.

2. Instantiate the transactor black box in the desired location. This black box can be found in the \$ZEBU_IP_ROOT/uc_xtor directory.
3. Configure the zceiClockPort in the top level Verilog wrapper.

The transactor can be instantiated at any level of the hierarchy.

The following is an example of instantiating a CGMAC transactor wrapper in the top level Verilog wrapper.

```
module example_cgmac_wrapper;  
  
`ifdef MII_SDR128b  
  
    wire          ref_clk_3125;  
    wire          ref_clk_1562_5;  
    wire          cgmii_rx_clk1;  
    wire [127:0]   cgmii_rxd1;  
    wire [15:0]    cgmii_rxc1;  
    wire          cgmii_tx_clk1;  
    wire [127:0]   cgmii_txd1;  
    wire [15:0]    cgmii_txcl1;  
    wire          cgmii_rx_clk2;  
    wire [127:0]   cgmii_rxd2;  
    wire [15:0]    cgmii_rxc2;  
    wire          cgmii_tx_clk2;  
  
`endif
```

Instantiating a Transactor

```

wire [127:0]      cgmii_txd2;
wire [15:0]       cgmii_txc2;
wire [31:0]       timestamp;

cgmac_sdr128_driver cgmac_driver_1 (
    .ref_clk_3125          (ref_clk_3125),
    .ref_clk_3125_primary   (1'b1),
    .ref_clk_1562_5         (ref_clk_1562_5),
    .ref_clk_1562_5_primary (1'b1),
    .cgmii_rx_clk           (cgmii_rx_clk1),
    .cgmii_rxd              (cgmii_rxd1[127:0]),
    .cgmii_rxc              (cgmii_rxc1[15:0]),
    .cgmii_tx_clk           (cgmii_tx_clk1),
    .cgmii_txd              (cgmii_txd1[127:0]),
    .cgmii_txc              (cgmii_txc1[15:0]),
    .xtor_cclock0           (ref_clk_3125)
);

cgmac_sdr128_driver cgmac_driver_2 (
    .ref_clk_3125          (ref_clk_3125),
    .ref_clk_3125_primary   (1'b1),
    .ref_clk_1562_5         (ref_clk_1562_5),
    .ref_clk_1562_5_primary (1'b1),
    .cgmii_rx_clk           (cgmii_rx_clk2),
    .cgmii_rxd              (cgmii_rxd2[127:0]),
    .cgmii_rxc              (cgmii_rxc2[15:0]),
    .cgmii_tx_clk           (cgmii_tx_clk2),
    .cgmii_txd              (cgmii_txd2[127:0]),
    .cgmii_txc              (cgmii_txc2[15:0]),
    .xtor_cclock0           (ref_clk_3125)
);

```

```
zceiClockPort ref_clk_1 ( .cclock  (ref_clk_1562_5));
zceiClockPort ref_clk_2 ( .cclock  (ref_clk_3125));

dut dut(
    .ref_clk_1562_5          (ref_clk_1562_5),
    .timestamp                (timestamp),
    .cgmii_rx_clk1           (cgmii_rx_clk1),
    .cgmii_rxd1               (cgmii_rxd1),
    .cgmii_rxc1               (cgmii_rxc1),
    .cgmii_tx_clk1           (cgmii_tx_clk1),
    .cgmii_txd1               (cgmii_txd1),
    .cgmii_txc1               (cgmii_txc1),
    .cgmii_rx_clk2           (cgmii_rx_clk2),
    .cgmii_rxd2               (cgmii_rxd2),
    .cgmii_rxc2               (cgmii_rxc2),
    .cgmii_tx_clk2           (cgmii_tx_clk2),
    .cgmii_txd2               (cgmii_txd2),
    .cgmii_txc2               (cgmii_txc2)
);

`endif

endmodule
```

10.2 Compiling a Transactor

This section provides instructions to compile the transactors in ZeBu.

Prerequisites

Ensure that you have created the UTF file and the top level Verilog wrapper.

To compile the transactors, perform the following steps:

1. Point the \$ZEBU_IP_ROOT environment variable to the location where your ZeBu transactors are installed.
For example, using a csh shell: `setenv ZEBU_IP_ROOT/remote/vginterfaces1/zebu_ip`
2. Add the following command to the ZeBu UTF project file: `xtors -use_zebu_ip_root true`
3. Instantiate the transactor instance in the hardware top. The module description can be found in the following Verilog file: (`ZEBU_IP_ROOT/uc_xtot/xtrodrivername.v`).

ZeBu performs the black-box testing of the transactor module at the front-end compilation stage.

After the front-end compilation, ZeBu collects the transactor's source from the \$ZEBU_IP_ROOT path and proceeds to with back-end compilation.

10.2.1 Example of a .utf file

The following is an example of a .utf file for CGMAC.

```
vcs_exec_command { vcs -full64 \
    -hw_top=example_cgmac_wrapper \
    +libext+.v \
    -y $ZEBU_IP_ROOT/uc_xtor \
    +define+$CGMAC_ITF \
    ../src/dut/example_cgmac_wrapper.v }
```

```
        .. /src/dut/dut.v }
architecture_file -filename "$env(FILE_CONF)"
grid_cmd -submit $env(REMOTECMD) -delete {}
grid_cmd -queue ZebuIse -submit $env(REMOTECMD) -delete {} -njobs
$env(MAXISEJOBS)
xtors -use_zebu_ip_root true
```

If you want to use Xilinx primitives, include the path of these primitives in the example transactor, as shown below:

```
-y $ZEBU_XIL/ISE/verilog/src/unisims
```

For more information about the example .utf file, see
example\src\env\<example_transactorname_wrapper>.utf.

10.3 Compiling a Transactor Example

In this section, the CGMAC transactor is used as an example to explain transactor compilation in ZeBu.

To compile the CGMAC transactor, perform the following steps:

1. Set the \$ZEBU_IP_ROOT environment variable to the installation path of the transactor.
2. Navigate to the example/Zebu directory.
3. Launch the compilation using % make compil.

10.4 Guidelines for Mapping Transactors in a Multi-RTB Environment

When a large number of transactors is used, it is necessary to map these transactors to different FPGAs. Such an environment is called multi-RTB environment.

You can declare a script for mapping a transactor in a multi-RTB environment using the following command:

```
|xtors -mapping_script <transactor_mapping_file>.tcl
```

The transactor mapping script is a tcl file, containing the following commands:

```
defmapping -rtlname <xator_name> <module_loc>
```

where,

- <xator_name>:
 - Specifies the name of the instantiated transactor.
 - Supports wildcards in defmapping with RTL names.
- <module_loc>: Specifies the ZeBu module to which the transactor is mapped.
The format for <module_loc> is U0_<module_ID>, for example, U0_M0.

11 Custom DPI Based Transactors With ZEMI-3

ZEMI-3 is a Standard Co-Emulation Modeling Interface (SCE-MI) 2.0 compatible behavioral SystemVerilog compiler for transactor BFM^s that simplifies writing cycle-accurate BFM^s and exchange data with a C++ or SystemVerilog testbench. The ZEMI-3 infrastructure enables writing transactors for functional testing of a design. ZEMI-3 is based on the following important concepts:

- The communication between the hardware and software parts uses inter-language function calls, based on the DPI mechanism, which is part of the IEEE standard for SystemVerilog (IEEE 1800-2012).
- The description of the hardware part of the transactor uses behavioral coding style.

This chapter discusses the following topics:

- *Choosing a Transactor Architecture*
- *Data Exchange Between Hardware and Software*
- *Clocks*
- *Advanced Features*
- *Writing the Hardware Part*
- *Writing the Software Part*
- *Compiling the Hardware Part of a ZEMI-3 Transactor*
- *Running Emulation of ZEMI-3 Transactors*
- *Running Emulation of ZEMI-3 Transactors in a Heterogeneous Environment*
- *Performing Simulation of ZEMI-3 Transactors*

11.1 Choosing a Transactor Architecture

A transactor organizes the interaction between the testbench and the DUT:

- The software part of the transactor is written in C/C++, and interacts with your testbench through high-level commands.
- The hardware part interacts with the DUT at a cycle-level or signal-level; it processes multi-cycle operations as an untimed functional model. These interfaces ease the integration of the testbench and DUT.

In a ZEMI-3 transactor, the hardware and the software interact through inter-language function calls. The transaction data correspond to the arguments of the function call, data transfers, and time synchronizations happen automatically. The calls can be initiated by the hardware or software according to the verification environment.

Selecting the transactor architecture depends on the types of operations processed by the software part and hardware part.

11.1.1 Selecting Hardware or Software Part for Processing

A transactor is composed of hardware and software parts and some operations work better in one part or the other. For example, complex arithmetic operations are more efficient in software and require a lot of resources in hardware. On the other hand, bit-level operations such as CRC calculations are more efficient in hardware.

11.1.2 Avoid Using an Unnecessary Large Bandwidth

With ZEMI-3, it is simple to stream the data out of the DUT for analysis purposes. However, streaming thousands of bits at every cycle is not an efficient way of using ZeBu, and may result in performance degradation.

When data is required by the software part, send only the necessary data.

11.1.3 Use Multi-Cycle Transactions

ZEMI-3 performance is maximized when multiple cycles are executed for each interaction between the software and hardware parts of the transactor. The more cycles in a single transaction, the faster it runs.

Data Exchange Between Hardware and Software

The ratio between the number of clock cycles and the number of inter-language calls should be maximized for the best performance.

11.1.4 Use Modular Transactor Architecture

When implementing a complex protocol such, as PCI Express, it is important to create a modular architecture with separate software-initiated commands, such as, Read PCI and Write PCI. Such a modular architecture makes it easier to write transactor software.

It also makes it easier to upgrade a transactor, that is, to add a new feature, you can create a separate function without impacting the existing code.

Advanced features, such as, protocol mode or serialized calls can help with modular transactor architecture.

11.2 Data Exchange Between Hardware and Software

An exchange can be initiated by the hardware or the software according to your verification environment. The choice for hardware or software -initiated exchanges depends on the architecture of the transactor.

11.2.1 Simulation Time Consumption

The simulation time consumption concept indicates that tasks consume simulation time (that is, clock cycles) when they wait for particular events or clock edges.

When using the SystemVerilog DPI, tasks consume time while functions must return instantaneously, without consuming time.

11.2.2 Hardware -Initiated Transactions

Hardware-initiated transactions are preferred to transfer data from hardware to software. In such cases, the hardware part typically requires a complex operation result, which is easier to implement in the software part.

You can use an import function, which is a C-language function called from the SystemVerilog code. This function has optional inputs that send data to the software and outputs that return data to the hardware. This mechanism allows the hardware part to send data to the software and then use the results produced by the software part.

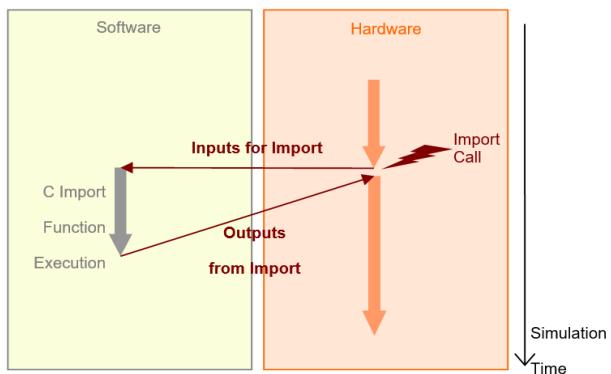


FIGURE 28. Hardware-Initiated Transaction

During the execution of an import function (in the software part), the hardware part of the transactor stops (the design clocks do not run). Such calls degrade the performance if used frequently, such as, at every clock cycle. The C import function is executed in its own software thread.

11.2.3 Software -Initiated Transactions

Software-initiated transactions are preferred to transfer data from software to hardware. In such cases, you can use a SystemVerilog export function. The software part of the transactor provides the inputs for this function. It allows the software part to send data to the hardware part and then use the results produced by the hardware.

Data Exchange Between Hardware and Software

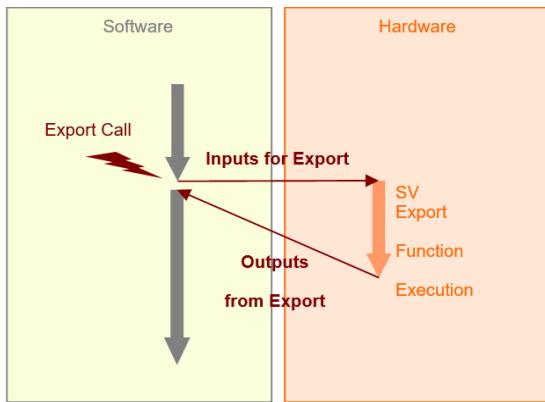
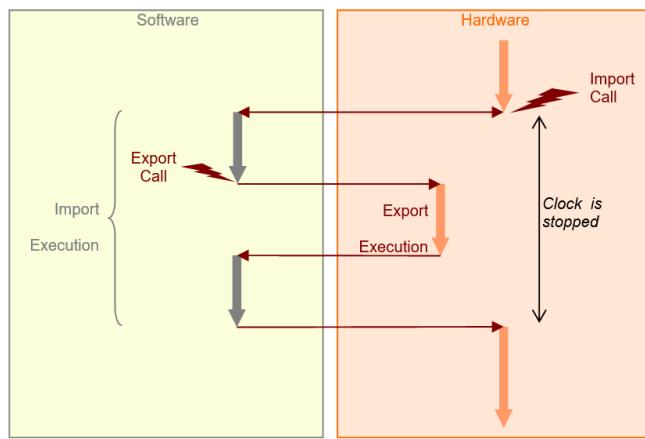


FIGURE 29. Software-Initiated Transactions

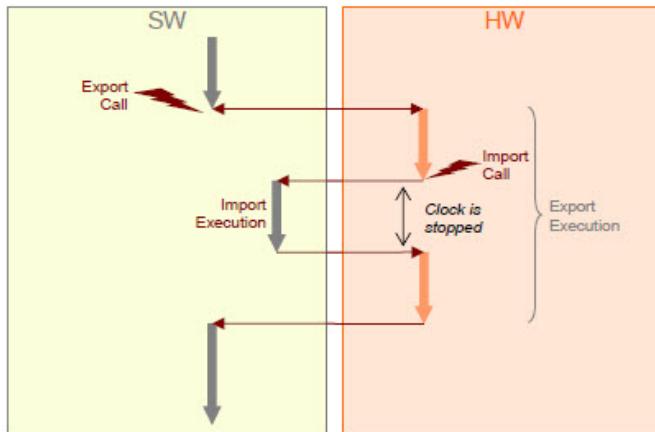
During export function execution, the software part of the transactor waits for the outputs to be returned by the hardware.

11.2.4 Mixing Imports and Exports

In ZEMI-3, an export function can be called from an import function and an import function can be called from an export function or task. The maximum level of nesting is two, that is, one import can call multiple exports; however, these exports cannot call another import.

**FIGURE 30.** Calling an Export From an Import (Context Import)

An import function that calls an export must be declared as a context import, as described in [Import Function](#). The export called from the import must not consume DUT clock cycles, because ZEMI-3 imposes the restriction that import functions must not consume simulation time.

**FIGURE 31.** Calling an Import From an Export

The export can consume simulation time. However, the included import cannot consume simulation time.

11.2.5 Functions and Tasks

For import calls only functions can be used because the software part does not directly control simulation time of the hardware part. From a hardware point of view, functions execute immediately, which means that any call of an import function stalls the transactor clock.

For export calls, both tasks and functions can be used but only tasks can wait for events to happen. Functions can only set or read values and must return immediately.

Export tasks and functions are executed by the software calls. They execute in parallel with all other hardware blocks. Only one instance of an export call can be active at any one time. If there are multiple calls, they execute sequentially (the second such export task waits until the first task finishes.)

Clock events can be used directly or through wait statements, as displayed in the following example:

```
always
begin
  @(posedge clk)
  a <= b;
  wait(clk == 1'b0);
  a <= c;
end
```

If a task that has no wait or event statement it should be written as a function.

Function or task arguments are transferred across the hardware and software parts without any delay at the synchronization points, see [Synchronization Points](#). The ZEMI-3 infrastructure prevents the channels from overloading.

Import tasks are not supported in the ZEMI-3 infrastructure, therefore, an import does not consume any simulation time; that is, in case of nesting, only an export function (not an export task) may be called from an import.

The size of the data exchanged between hardware and software at a synchronization point can be up to 8,000 bits in each direction (input or output). If the data is bigger

than this limit, the messages are automatically turned into multi-cycle messages, hence, impacting runtime performance.

11.2.6 Managing the Call Order of Software Import Functions

When functions are called in the same SystemVerilog process (always or initial block), the order of calls is guaranteed to be the same on both hardware and software parts, even in a streaming environment.

When function is called in separate processes, the order in which they execute on the software part can differ from the order in which they were called by the hardware part. This order can be made deterministic through the serializing calls feature, see [Serializing Calls](#).

11.2.7 Synchronization Points

Synchronization points are needed when bi-directional data exchanges occur between the hardware and software parts of the transactor. In this case, the hardware and software parts need to wait for each other.

It is recommended to avoid functions or tasks that mix inputs and outputs to minimize synchronization points. For example,

- Using an export function or task with outputs (or with a return code) prevents the software from performing any other operation in parallel.
- Using a non-streaming import function in a clocked always block stops the clock (for more details about streaming data, see [Streaming Data](#)). If required, replace the import function by a streaming import function and a streaming export function.

Such cases should be rare, or should correspond to a large number of cycles of activity in the design.

Synchronization points are required to create the full environment. However, the synchronization points should be minimized to limit the impact on performance. Too many synchronization points may result in frequent communication and degrade performance. The best performance can be achieved when those points are performed after a large number of cycles, or completely avoided as in data streaming.

11.2.8 Streaming Data

It is important to remove the synchronization points in a pipe-like communication channel in which the data moves in a single direction: software to hardware or hardware to software. Such situations exist when import or export functions have only inputs. Functions with only outputs cannot be considered for streaming because they require the following bi-directional exchanges:

- When the call is made; although, there is no input.
- When the outputs are sent back.

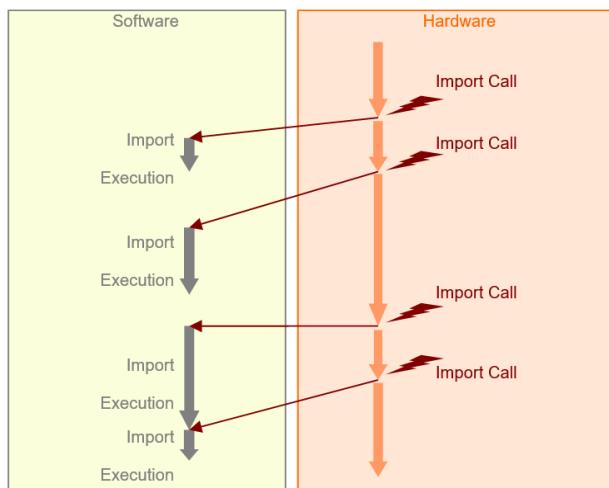


FIGURE 32. Streaming Import

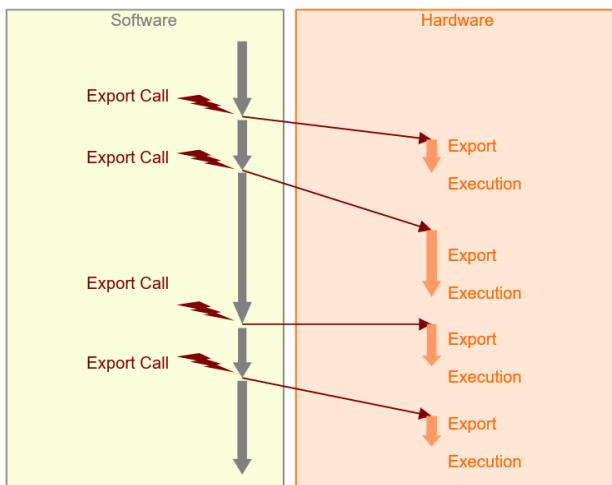


FIGURE 33. Streaming Export

It is possible for ZEMI-3 to turn function calls into a streaming communication to avoid synchronization points and to optimize the transactor performance. When there are only inputs, hardware and software parts can execute concurrently without having to wait for one another.

In streaming mode, the ZEMI-3 infrastructure automatically optimizes the communication speed by buffering the requests and avoid any possible overloading of the communication channel.

The following call types are automatically processed as streaming communication:

- Non-context import functions with inputs only
- Export functions with inputs only
- Export tasks with inputs only

The automatic conversion to streaming mode can be disabled for some call types. Context import functions can also be forced into streaming mode, see [Forcing/Disabling Streaming Mode](#). However, nested import and export functions (see [Mixing Imports and Exports](#)), must never be forced into streaming mode, because it causes unexpected runtime behavior.

It is recommended to create transactions in the streaming mode otherwise ZEMI-3 cannot optimize communication in the presence of synchronization points.

Clocks

If output-only import functions are used significantly by the environment, the prefetch mechanism may improve performance. For more details, see [Prefetch Calls](#).

In ZEMI-3 transactors, `zceiMessageOutPort` is used to send arguments of import DPI calls from hardware to software.

To stop sending constant arguments of import DPI call through `zceiMessageOutPort`, use `zemi3 -optimize_dpi_constant_args true`.

This can help optimize and save hardware - software communication bandwidth and the hardware capacity by reducing the size of `zceiMessageOutPort`.

11.3 Clocks

11.3.1 Controlled Clocks

A ZEMI-3 transactor must have at least one controlled clock. Such clocks are input ports to the transactor and they are declared in the hardware top by instantiating a `zceiClockPort` (see [Instantiating a ZEMI-3 Transactor in the Hardware Top](#)).

Controlled clocks are enabled by default. If no data is available from the software or the software is not ready to receive data during a DPI call then all controlled clocks of the transactor are automatically stopped until the data becomes available or the software is ready. If there are multiple clocks, all the groups to which they belong are stopped.

ZEMI-3 transactor supports sample clock optimizations to avoid the sampled clock effect by setting `zemi3 * -sample_clock_mode as true`.

11.3.2 Sampled Clocks

In some cases, clocks can be generated by the DUT as in an LCD design. This kind of clock is known as a sampled clock, and it must not be declared in the same way as a controlled clock.

11.3.3 Ensuring the Best Performance

For higher performance it is best to limit the number of clocks used in a transactor. If possible, avoid divided clocks and gated clocks.

Using sampled clocks may slow down the transactor because the controlled clocks have to be stopped to ensure consistent inputs from the design.

From a ZEMI-3 point of view, any signal used in a @`(posedge <signal>)` or @`(negedge <signal>)` statement is considered a controlled or sampled clock.

11.4 Advanced Features

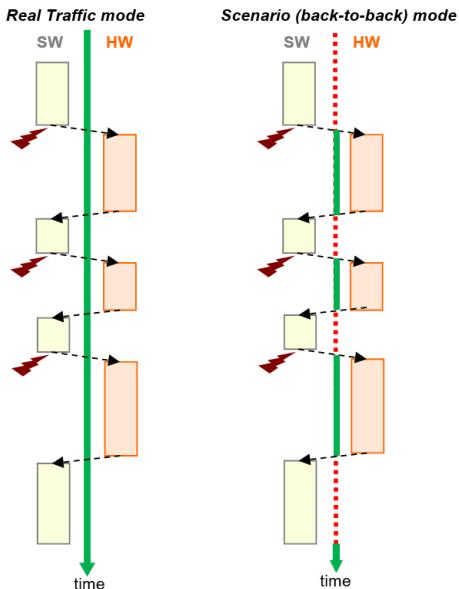
For implementation details of the features described in this section, see [Supported System Calls](#). The following are the advanced features supported by the ZEMI-3 infrastructure:

- [Back-to-Back Mode](#)
- [Protocol Mode](#)
- [Prefetch Calls](#)
- [Serializing Calls](#)
- [Lossy Calls](#)
- [Port Buffering](#)
- [Multi-Cycle Messaging \(Port Multiplexing\)](#)
- [Automatic Compiler Optimization](#)

11.4.1 Back-to-Back Mode

When verifying your design using transactors, you must consider the following verification strategies:

- System Functional Testing: In Real Traffic mode, transactor clocks are always running. The delay between consecutive transactions is not deterministic.
- Corner-Case Testing: In Scenario mode, insertion of idle cycles between transactions is forbidden except when explicitly requested (clocks have to be controlled in such a way that only requested cycles are processed). This is also called the back-to-back mode.

**FIGURE 34.** Back-to-Back Mode

During the verification process, you must switch from the real traffic mode to the scenario mode many times, with a minimum development cost. The back-to-back mode enables you to design your transactor as compliant with your real-traffic specifications. The same transactor can be operated in either mode.

The back-to-back mode can be selected at runtime (from the C/C++ API). The initial back-to-back mode can be set during compilation (in UTF file).

11.4.2 Protocol Mode

When several export functions (or tasks) use shared resources, it is important to prevent concurrent accesses from different processes in order to avoid read/write conflicts. The ZEMI-3 infrastructure addresses this via the protocol mode, which automates the scheduling of functions (and tasks).

This mode is activated by declaring an additional advanced property when the export functions are declared in the SystemVerilog code.

Streaming mode can be used for protocol-gathered export tasks and functions.

11.4.3 Prefetch Calls

When an import function has only outputs, the software can sometimes prepare data so that the hardware part does not have to wait for the data at the synchronization points. This is known as prefetch calls.

- Without prefetching, an import call causes the hardware to request new data from the software. The software services the request by sending the data back to the hardware. The clocks then restart until the next call.
- Prefetching (for an import with output data only) eliminates the need to wait for the actual call to prepare the data for the (next) request.

Prefetching is similar to the ZEMI-3 streaming mode for software to hardware communication, except that the software controls the communication flow.

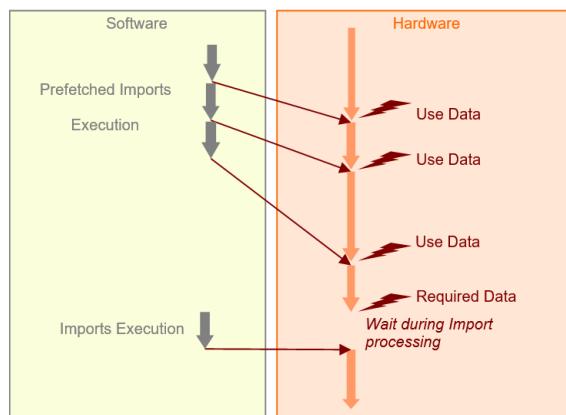


FIGURE 35. Prefetch Mode

The compiler automatically creates a specific prefetch function for each import function with only outputs. The software calls the prefetch function and stores the results until they are needed by the hardware part.

Prefetch mode can be enabled or disabled independently for each:

- When enabled, the original import function is not called; instead, the data is provided by the prefetch function.
- When disabled, the import function is called.

11.4.4 Serializing Calls

When multiple processes call import functions, ZEMI-3 does not guarantee that the software handles the calls in the same order as they were called in the hardware.

When two functions are called at different times, the later call may be executed first by the software.

When it is important to preserve the call order, you should explicitly serialize the calls.

A general guideline is that software functions that operate on the same data must be serialized (for example, read and write calls to the same memory).

Serializing calls allow you to use the same hardware port for several imports. However, if an import requires a high throughput then it should not be serialized.

11.4.5 Lossy Calls

This feature can be specified for an import function to keep it from sending messages to the software when the communication channel is busy. For more details, see

[Enabling System Calls](#).

11.4.6 Port Buffering

This feature can be specified for a streaming import function to create buffers for the message. This can be useful when exchanging small packets that can tolerate higher latency. For more details, see [Port Buffering](#).

11.4.7 Multi-Cycle Messaging (Port Multiplexing)

This feature allows splitting messages into smaller packets, which allows for smaller hardware ports. This feature applies to imports and exports and can be set in the UTF file.

11.4.8 Automatic Compiler Optimization

Compiler optimization settings for the hardware part of the transactor can be specified in the UTF file, in particular for hardware-software communication.

These settings combine the advanced features described in [Advanced Features](#), according to the environment and constraints.

11.5 Writing the Hardware Part

11.5.1 Imports and Exports in SystemVerilog Code

11.5.1.1 Import Function

An import function declared in SystemVerilog can be called by the hardware part of the transactor. Both the declaration and the call reside in the same Verilog module.

The function is implemented in C by the software part of the transactor. For more details, see [Implementing Import Functions in C](#).

For example:

```
// Import declaration in SystemVerilog
import "DPI-C" function void read_addr(input bit [31:0] addr,
                                         output bit [31:0] data);

// Import call in SystemVerilog
reg addr;
reg data;
always @(posedge clk)
begin
    addr <= addr + 1;
    read_addr(addr, data);
    val <= data;
end
```

When you want to use scope information, the import function must be declared explicitly as a context import. This is required when the import function is nested within an export function.

Writing the Hardware Part

For example:

```
import "DPI-C" context function void send_one_addr(input bit [31:0]
addr, input bit [31:0] data);
```

11.5.1.2 Export Function/Task

For exports, both the declaration and the implementation of the function (or task) reside in the same SystemVerilog module.

The export declaration does not include the function prototype; the prototype is specified only in the implementation.

For example:

```
// Export declaration in the SystemVerilog code
export "DPI-C" task write_one_addr;

// Export implementation in the SystemVerilog code
task write_one_addr(input bit [31:0] addr, input bit [31:0] val);
begin
    mem_we <= 1;
    mem_addr <= addr;
    mem_din <= val;
    @(posedge mem_clk);
    mem_we <= 0;
    @(posedge mem_clk);
end
endtask
```

11.5.1.3 Blocking Assignment in Export Tasks

Non-blocking assignments in export tasks can result in unexpected runtime behavior, in particular when assigning a value to an output argument as the final statement of the task.

For example:

```
task receiveCmd(input bit[31:0] cmd);
begin
  @(posedge clk);
  while (!lastCmdEnd)
    @(posedge clk);
  nextCmd = cmd;
  newCmdEn = 1'b1;
  @(posedge clk);
  newCmdEn = 1'b0;
end
endtask
```

A non-blocking assignment to an output argument never passes the correct value to the software because the non-blocking assignment is not executed correctly. Always use blocking assignments in export tasks to avoid such situations.

11.5.2 Using Behavioral Coding Style

The hardware part of a transactor designed with ZEMI-3 can be written using behavioral code, which is normally not synthesizable.

11.5.2.1 Supported Behavioral Subset

The following constructs are supported in the behavioral description of the hardware part of a transactor:

TABLE 19 Supported Sub-Set of Behavioral Code

Supported Constructs	Example
Standard synthesizable RTL subset	-
User-Defined Primitives (UDPs)	Typically, Xilinx primitives

Writing the Hardware Part

TABLE 19 Supported Sub-Set of Behavioral Code

Supported Constructs	Example
initial statements	initial counter = 0;
@edge in always/initial statements	initial begin a = 0; @(posedge clk); a = 1; end
@edge in if and loops	for (i=0;i<128;i=i+1) begin @(posedge clk); mem[i] = 0; end
Multiple clocks/edges in the same process	always @(posedge clk) begin a = 0; @(negedge clk); a = 1; @(negedge reset); a = 2; end
Complex edge combination	always @(posedge clk or negedge clk2)
Edge and level events	always @(clk or reset or posedge sig)
Unbounded loops	Initial while (1) begin @(posedge clk1) c <= c + 1; end
wait statements	wait(clk);
Named events	event my_event; initial -> my_event; always begin @(my_event); a <= b; end
Static SystemVerilog data types	logic, bit, int, longint, byte, void, struct, enum
Block disable within a process	-

TABLE 19 Supported Sub-Set of Behavioral Code

Supported Constructs	Example
Delta delays for races	zemi3_event advanced property
System tasks	<ul style="list-style-type: none"> • \$display • \$fdisplay • \$writeh • \$fclose <p>For a complete list of supported system tasks, see List of Supported System Calls.</p>

11.5.2.2 Clocks, Multiple Clocks, and Multiple Edges

Any combination of clocks and edge expressions are supported in transactors, even inside export tasks. It is possible to mix, in the same process, multiple clocks and multiple edges of the same clock. For example, inserting statements like @(posedge clock) or @(negedge clock) in the blocks. It is also possible to use such statements to describe a sequence of actions that execute as an implicit state machine.

Note

Do not mix controlled clocks and sampled clocks in the same event expression.

11.5.2.3 Usage of Blocking and Non-Blocking Assignments

Mixing blocking and non-blocking assignments to the same register is not allowed and results in a compiler error.

Non-blocking assignments should not be used for function outputs or return values since functions cannot consume time. However, they can be used for side effects.

11.5.2.4 Limitations

The following features are not supported in ZEMI-3:

- SystemVerilog data types other than logic, bit, int, longint, byte, void, struct, enum.
- fork/join

Writing the Hardware Part

- cmos, nmos, pmos, rcmos, rnmos, rpmos
- Strengths on drivers
- Delays for behavior
- System tasks/functions other than:
 - \$display, together with: \$displayh, \$displayb, \$displayo
 - \$fdisplay, together with: \$fdisplayh, \$fdisplayb, \$fdisplayo
 - \$write, together with: \$writeh, \$writeb, \$writeo
 - \$fwrite, together with: \$fwriteh, \$fwriteb, \$fwriteo
 - \$fopen; \$fclose
- Procedural assign/deassign
- Inter-process block disable and task disable

11.5.3 Adding Delta Delays

It is possible to add to the hardware part of the transactor delta delays clocked by the ZeBu driverClock, the internal high-speed clock. Because delta delays introduce sequential elements, they can be used to tradeoff performance for hardware size. Since the frequency of the driverClock is higher than any design clock in the system, delta delays are shorter than the shortest design clock cycle.

Adding delta delays in loops can reduce the hardware size, but they reduce the speed.

A delta delay is specified using the zemi3_event pragma in an empty statement:

```
(* zemi3_event *);
```

Be sure to add the semi-colon; otherwise, you may experience incorrect runtime behavior.

You can also use delta delays to split long combinational paths, or to reduce the number of ports required for a memory, as in the following example:

```
reg [31:0] mem [0:1023];
integer i;
for (i=0;i<1024;i=i+1) begin
(*zemi3_event*);
mem[i] = 0;
end
```

In this case, the 1024-port memory requires a multi-port if synthesized without delta delays. The insertion of delta delays at each access transforms the memory into a single-port memory with 1024 consecutive accesses, but the performance is lower since it requires 1024 driverClock cycles to execute the loop.

11.5.4 Supported System Calls

11.5.4.1 List of Supported System Calls

The following system calls are supported in the source code of a ZEMI-3 transactor:

- \$display, together with: \$displayh, \$displayb, \$displayo
- \$fdisplay, together with: \$fdisplayh, \$fdisplayb, \$fdisplayo
- \$write, together with: \$writeh, \$writeb, \$writeo
- \$fwrite, together with: \$fwriteh, \$fwriteb, \$fwriteo
- \$fopen
- \$fclose

All these system calls are serialized with respect to each other (so they execute in order and share resources), but they remain independent (not serialized) with respect to DPI calls.

11.5.4.2 Enabling System Calls

To enable system calls, you must add the following line to the ZEMI-3 attribute file before compiling:

```
zemi3 -support_dollar_display true
```

11.5.5 Advanced Properties

The following table lists the SystemVerilog pragmas that denote advanced properties that can be declared in the hardware part of a transactor to modify or optimize its behavior:

TABLE 20 Advanced Properties

Advanced Properties	Example
(* zemi3_event *) ;	Inserts one delta delay (see Usage of Blocking and Non-Blocking Assignments).
(* zemi3_stream=0 *)	Disables streaming mode on import/export (see Streaming Data and Forcing/Disabling Streaming Mode).
(* zemi3_highpriority *)	Sets the ports associated with import/export to high priority.
(* zemi3_protocol=<my_protocol> *)	Specifies the name of the protocol when exported tasks share critical resources (see " Protocol Mode ").
(* zemi3_serialize=<my_serializer> *)	Specifies the name of the serializer when imported tasks are serialized (see Serializing Calls and Serialization).
(* zemi3_call_skipped = "fail" *)	Specifies the import to be a lossy call and the name of the bit indicating if the import is performed or not (see Lossy Calls).
(* zemi3_bufferedport [= <size>] *)	Specifies the import to be buffered and gives the size of the buffer (see Port Buffering).

11.5.5.1 Forcing/Disabling Streaming Mode

If you do not want ZEMI-3 to optimize DPI calls into the streaming communication, you can disable streaming mode by specifying `zemi3_stream=0` in the DPI declaration:

```
(* zemi3_stream=0 *)  
import "DPI-C" function int start_test ();
```

By default, context import functions are not optimized for streaming because they may call export functions. If the context import implementation contains no export call, the streaming mode can be enabled by specifying `zemi3_stream=1` in the DPI declaration:

```
(* zemi3_stream=1 *)  
import "DPI-C" context function int start_test ();
```

11.5.5.2 Protocol Mode

To declare export tasks for a given protocol, you must specify the same advanced property (`zemi3_protocol=<protocol_name>`) before the declaration of each export task that belongs to the protocol.

For example:

```
(* zemi3_protocol="dut_inc_dec", zemi3_stream=0 *)  
export "DPI-C" task inc_dut_input;  
  
(* zemi3_protocol="dut_inc_dec" *)  
export "DPI-C" task dec_dut_input;
```

11.5.5.3 Serialization

To serialize import calls, use the `zemi3_serialize=<serializer name>` advanced property before the declaration of each import task that belongs to the same serializer.

 Writing the Hardware Part

For example:

```
(* zemi3_serialize="mem_read_write" *)
import "DPI-C" context function int mem_read_addr (int addr);

(* zemi3_serialize="mem_read_write" *)
import "DPI-C" context function void mem_write_addr (int addr, int
data);
```

11.5.5.4 Lossy Calls

The lossy calls feature allows you to skip an import call when the communication channel between the hardware and the software is busy. Without the lossy calls feature, the import would block the hardware part until the communication channel becomes available.

To determine if the call has executed, an artificial output bit must be added at the end of the argument list. This bit is set to 1 after the call to indicate that the call did not execute.

To specify the import to be a lossy call, the following property has to be used:

```
(* zemi3_call_skipped = "fail" *)
import "DPI-C" function void send_result(input bit [127:0] dout,
output bit fail) ;
```

For example:

```
module xtor(input CLK,
            input [31:0] dout) ;
  integer i ;

  reg [31:0] failures ;
  bit fail ;
```

```
(* zemi3_call_skipped = "fail" *)
import "DPI-C" function void send_result(input bit [127:0]
dout,output bit fail) ;

initial failures = 0 ;

always @(dout) begin
    send_result(dout, fail) ;
    if (fail == 1) failure = failure + 1 ;
end

/* ... */

endmodule
```

11.5.5.5 Port Buffering

For streaming import calls, it is not necessary for the software to see the effect of the import call immediately. Instead, the cost of sending many small messages can be saved by accumulating a number of these messages and sending them as a single block.

Sending fewer messages allows you to increase the global frequency of the controlled clocks. However, this increases the latency between the hardware call and the import function call.

This can be done only for imports that are streaming and not part of any serializer (named or anonymous).

```
(* zemi3_bufferedport[=<size>] *)
```

where `<size>` is the size of the final accumulated block. This value is optional - if it is not given, a default value is used (which can be modified in the UTF file).

Writing the Software Part

For example:

```
//Create a buffer of 31 import calls (buffer is 4096 for 128-bit
messages)

(* zemi3_bufferedport=4096 *)
import "DPI-C" function void send_result(input bit [127:0] dout)
```

11.6 Writing the Software Part

11.6.1 Calling Export Functions/Tasks in C

Export functions (and tasks) are SystemVerilog functions (and tasks) that are called from the software part of the transactor (see *Import Function*).

The prototype of the export function is available in the ZEMI-3 transactor header file (<my_xtor>.h) generated by **zcui** during compilation of the hardware part of the ZEMI-3 transactor (see [Compiling Output Files for the Software Part of the Transactor](#)). In the software part of the transactor, the export function is called as any other C function:

```
for(uint i=0;i<10;i++) {
    write_one_addr(i, data[i]);
}
```

Before calling an export function, the current scope has to be properly set to the instance path of the export function in the transactor. This is done using the standard SystemVerilog DPI utility function `svSetScope`.

For example:

```
/* Setting the scope of the export call */
svScope s = svGetScopeFromName("top.xtor0");
svSetScope(s);
```

11.6.1.1 Prefetch Mode

In prefetch mode (see [Prefetch Calls](#)), the import function is not called with its SystemVerilog name. The following specific functions are generated for each import function having only outputs:

- The `block_prefetch_<import_name>` function provides a blocking mode.
- The `try_prefetch_<import_name>` function provides a non-blocking mode.

Enabling or Disabling the Prefetch Mode

Enable the prefetch mode (the original import function is not called while prefetching is enabled) by declaring the function as follows:

```
extern "C" void enable_prefetch_<import_name> ()
```

Disable the prefetch mode as follows:

```
extern "C" void disable_prefetch_<import_name> ()
```

Blocking Prefetch Functions

Calling the `block_prefetch_*` function blocks the prefetch function when the message buffer is full. It is recommended to call the `block_prefetch_*` function in a separate thread, so that the main processing is never blocked. This function always sends the message (once the message buffer becomes available) and returns true.

```
bool block_prefetch_<import_name> ([<return_value>,] <import args>)
```

where,

- `<import_name>` is the original name of the import function.
- `<import args>` is the list of arguments of the import function.
- `<return_value>` is the value returned by the import function if any.

Non-Blocking Prefetch Functions

The `try_prefetch_*` and `block_prefetch_*` functions are similar, except that `try_prefetch_*` does not send the message when the message buffer is full, and returns false. It returns true if the message is sent.

```
bool try_prefetch_<import_name> ([<return_value>,] <import args>)
```

where,

- `<import_name>` is the original name of the import function.
- `<import args>` is the list of arguments of the import function.
- If the import function returns a value using the return mechanism, `<return_value>` is this value returned by the import function.

11.6.1.2 zemi3_ExportCallIsBlocked() Macro

This macro can be used to determine if an export function executes without blocking the software part. For example, if the hardware part is not ready to receive any message at this time, the macro returns true once the hardware part is ready to execute the export call. For example,

```
//...
while (zemi3_ExportCallIsBlocked(xtor0_send) ) {
    xtor1_get_data(data);
    screen_display(data);
}
// Now the export is ready to be executed
xtor0_send(command);
//...
```

11.6.2 Implementing Import Functions in C

An import function (or task) is a C function that is called from the hardware part of the transactor (see [Import Function](#)). The import function must be implemented as an extern "C" function:

```
extern "C" void read_addr(const uint *addr, const uint *data)
{
    *data = array[*addr];
}
```

Note

- All import functions are resolved in the global scope.
- In a ZEMI-3 transactor, calls from anywhere in the hardware scopes call the same import function (same name import). If the code needs to understand the call context, then the import should be declared as context. In this case, the SystemVerilog function `svGetScope()` can be used to determine the context of the call.

11.6.3 C Types Supported by ZEMI-3

The DPI standard defines the mapping between SystemVerilog types and their C counterparts. The ZEMI-3 transactor does not use all of them, but supports the main ones.

TABLE 21 Mapping Between SystemVerilog and C Language Types

DPI Formal Argument Types	Corresponding Types Mapped to C
byte	char
byte unsigned	unsigned char
shortint	short int
shortint unsigned	unsigned short int
int	int

TABLE 21 Mapping Between SystemVerilog and C Language Types

DPI Formal Argument Types	Corresponding Types Mapped to C
int unsigned	unsigned int
longint	long long
longint unsigned	unsigned long long
Scalar values of bit type	unsigned char
Packed one-dimensional arrays of both: • bit types • logic types	Canonical arrays of both: • svBitVecVal • svLogicVecVal
Packed multi-dimensional arrays are also supported, but they are presented as single dimensional arrays on the C-side.	

The specific SystemVerilog types are declared in the standard header file, which can be found in \$ZEBU_ROOT/include/svdpi.h.

11.6.4 SystemVerilog C Functions Supported by ZEMI-3

The following table lists the SystemVerilog standard DPI utility functions supported in the ZEMI-3 environment (for more details, see the SystemVerilog LRM):

TABLE 22 Supported SystemVerilog DPI Utility Functions

Prototype	Description
svScope svGetScope(void)	Gets the scope of called imported function
svScope svSetScope (const svScope scope)	Sets the scope for exported function
int svPutUserData (const svScope scope, void *userKey, void *userData)	Associates a data and a scope

TABLE 22 Supported SystemVerilog DPI Utility Functions

Prototype	Description
void *svGetUserData (const svScope scope, void *userKey)	Gets data associated to a scope
const char *svGetNameFromScope (const svScope scope)	Gets the scope name
svScope svGetScopeFromName (const char *scopeName)	Gets the scope from its name
int svGetCallerInfo (char **fileName, int *lineNumber)	Gets the line number and file name of the caller
const char *svDpiVersion(void)	Gets the DPI version number
svBit svGetBitselBit (const svBitVecVal *s, int i)	Gets a bit from a svBitVecVal
void svPutBitselBit (svBitVecVal *d, int i, svBit s)	Puts a bit in a svBitVecVal
void svGetPartselBit (svBitVecVal *d, const svBitVecVal *s, int i, int w)	Gets a part of a svBitVecVal
void svPutPartselBit (svBitVecVal *d, const svBitVecVal s, int i, int w)	Puts a part of svBitVecVal in another svBitVecVal

These functions are declared in the standard header file, which can be found in \$ZEBU_ROOT/include/svdpi.h.

11.6.5 Controlling the Back-to-Back Mode From Your Testbench

With the back-to-back mode you can stop the clocks of a specified transactor between calls to export functions. This mode can be enabled at compilation or runtime and can be disabled at runtime.

The following table lists C functions available for back-to-back control.

TABLE 23 Back-to-Back Mode Functions

Function	Description
<code>void ZEMI3_startBack2Back const char *xtorName)</code>	Starts the back-to-back mode for the specified transactor.
<code>void ZEMI3_stopBack2Back const char *xtorName)</code>	Stops the back-to-back mode for the specified transactor.
<code>bool ZEMI3_isBack2BackStarted (const char *xtorName)</code>	Returns true if the back-to-back mode is started for the specified transactor.

11.7 Compiling the Hardware Part of a ZEMI-3 Transactor

The hardware part of a ZEMI-3 transactor is compiled from its SystemVerilog description, and, optionally, EDIF netlists and zMem memory descriptions. The hardware part of a ZEMI-3 transactor is compiled by **zCui**'s Unified Compile as illustrated in the following figure.

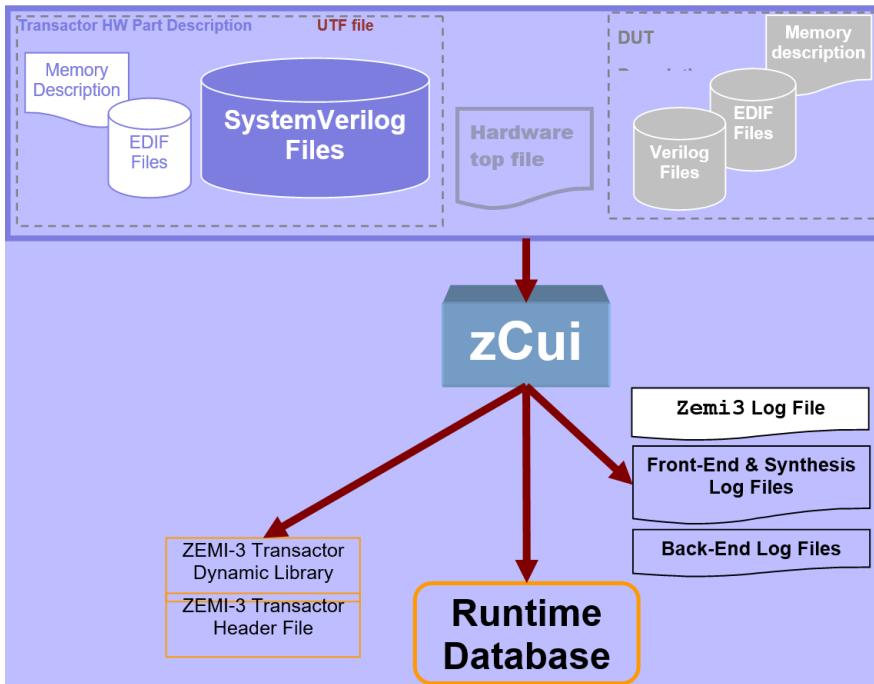


FIGURE 36. Compiling the Hardware Part of a ZEMI-3 Transactor

The transactor is first processed by VCS to identify all the export and import functions and tasks. This operation is done before launching the design front-end for synthesis.

This compilation process requires a hardware top in which the transactor is connected to the DUT and the controlled clocks are declared.

The compilation of the software part of the transactor is different according to the runtime environment.

11.7.1 Instantiating a ZEMI-3 Transactor in the Hardware Top

A ZEMI-3 transactor can be instantiated anywhere in the design hierarchy beneath the hardware top-level (`hw_top`). The transactor is instantiated with its connections to the DUT in a SystemVerilog compliant way.

In a ZEMI-3 transactor, the clock is declared like any other port of the transactor. The clock must be a controlled clock generated by ZeBu, accordingly, the clock port of the transactor is connected to the output of a `zceiClockPort`.

For example:

```
streamer_out s0(
    .clk(clk),
    .data(count)
);

zceiClockPort c(
    .cclock(clk)
);
```

Note

A transactor top module cannot have the name "reset". This is a known limitation and it is recommended to use a name other than "reset".

11.7.2 Adding and Setting a ZEMI-3 Transactor in the UTF File

11.7.2.1 Adding a ZEMI-3 Transactor

To add a ZEMI-3 transactor in the compilation project, use the following UTF command:

```
xtors -add {list of xtor_module_names} -type ZEMI3
```

For a detailed description, type "vcs -help utf+xtors".

The following table lists ZEMI-3 options available in the UTF file. For command line help, type "vcs -help utf+zemi3".

TABLE 24 ZEMI-3 Options available in the UTF File

Function	Description
<code>*-module {list_of_zemi3_transactors_modules}</code>	Provides a list of ZEMI-3 transactor modules.
<code>*-instance <xmr path to xtor> -disable <xmr to signal>]</code>	Specifies a disable signal for a ZEMI-3 transactor instance.
<code>*-allow_streaming <bool></code>	Disables streaming for the transactor when this is set to false.
<code>*-start_in_back_to_back <bool></code>	Starts run in back to back mode (clocks do not progress until an export is called.).
<code>*-support_dollar_display <bool></code>	Adds support for \$display.
<code>*-do_profiling <bool></code>	Enables profiling.
<code>*-hard_max_out_port_width <int></code>	Sets the maximum ZCEI output port width.
<code>*-hard_max_in_port_width <int></code>	Sets the maximum ZCEI input port width.
<code>*-auto_flush_interval <int></code>	Sets the auto-flush period for buffering in <code>driverclock</code> cycles.
<code>*-port_optimization_mode 0 1 2 3 0:AREA_OPT 1:THROUGHPUT_OPT 2:LATENCY_OPT 3:DEFAULT_OPT</code>	Sets the port optimization mode.
<code>*-profile_counters_width <int></code>	Sets the width of profiling counters.

Compiling the Hardware Part of a ZEMI-3 Transactor

TABLE 24 ZEMI-3 Options available in the UTF File

Function	Description
<code>*-all_tf_args_are_automatic <bool></code>	Gives control over task/function arguments. By default, it is set to false. When task/function is duplicated (inlined): If <code>all_tf_args_are_automatic</code> is false, the static variables need to be synchronized across copies. Otherwise, they are considered automatic and no synchronization is needed.
<code>*-external_controlled_clocks {list_of_external_controlled_clocks}</code>	Specifies a list of additional control clocks for the transactor.
<code>* -allow_mixed_design_clocks <bool></code>	Allows a mix of sampled clocks and controlled clocks in the same blocks.
<code>*-exports_excluded_from_back_to_back {list_of_export_DPI_excluded_from_back_to_back_mode}</code>	Lists exports excluded from back to back.
<code>*-max_loop_iterations <int></code>	Controls the maximum size of combinational iterating loops in ZEMI-3 synthesis.
<code>*-max_clocked_loop_iterations <int></code>	Controls the maximum size of sequential iterating loops in ZEMI-3 synthesis.
<code>*-merge_multiple_comb_writers <bool></code>	Handles multiple drivers of the same variable from different blocks. Writes are serialized.
<code>*-default_cclk {xmr to signal that is connected to zceiClockport}</code>	Sets the default controlled clock if the compiler is unable to select it automatically.

TABLE 24 ZEMI-3 Options available in the UTF File

Function	Description
* -sample_clock_mode <bool> Note: -sample_clock_mode is a global option. Other -module -instance options cannot be used with -sample_clock_mode.	Enables sampled clock optimization mode to avoid sampling lock effect.
* -timestamp <bool> Note: -timestamp can be used with or without -module.	Allows using svGetTimeFromScope in DPI imports.

Setting for the Streaming Option (Optional)

Streaming mode improves performance by optimizing single-direction data exchanges (see also *Streaming Data* and *Forcing/Disabling Streaming Mode*). It is enabled by default; however, you can disable it using the following UTF command:

```
zemi3 -module {list_of_zemi3_transactors_modules} -allow_streaming
false
```

Setting for the Back to Back Mode (Optional)

Running your emulation in back to back mode eases the debug analysis of ZEMI-3 transactors. For more details, see *Back-to-Back Mode*.

To enable the back to back mode, add the following command in the UTF file:

```
zemi3 -module {list_of_zemi3_transactors_modules} -
start_in_back_to_back true
```

Defining Advanced Settings (Optional)

Some advanced settings for ZEMI-3 transactors can be set in the UTF file. The following option can be used for automatic optimization:

```
zemi3 -module {list_of_zemi3_transactors_modules}
    -port_optimization_mode 0|1|2|3

0 for AREA_OPT2 for LATENCY_OPT
1 for THROUGHPUT_OPT3 for DEFAULT_OPT
```

Compiling the Hardware Part of a ZEMI-3 Transactor

where,

- AREA: Minimizes FPGA resources by factoring the usage of hardware ports for imports and exports. This option also reduces, whenever feasible, the message size by splitting the message in sequential packets (multi-cycle).
- LATENCY: Minimizes the exchange time between hardware and software.
- THROUGHPUT: Maximizes data throughput by buffering streaming import messages. You can use the `zemi3_highpriority` property (see [Supported System Calls](#)) when a transactor contains an import that needs maximum throughput. Also, user-defined optimizations, specific to the message multiplexing and message buffering, can be specified in the UTF file with `zemi3` option.
- Use Message Multiplexing: Activates message multiplexing for software/hardware and hardware/software communication with the maximum port width in the associated field. The default values are 8096 for software/hardware, and 8160 for hardware/software.
- Use Message Buffering: Activates streaming for imports, with the maximum message port width in the associated field (the default value is 8096). The port width declared here is the default value used for message port buffering, which can be modified with `zemi3_bufferedport` property (see [Advanced Properties](#)).

11.7.3 Compiling a ZEMI-3 Transactor with zCui

If you have all the sources of a ZEMI-3 transactor and DUT, you can compile with zCui using the following command.

```
zCui -u <project.utf> <other options>
```

11.7.4 Results of the ZEMI-3 Transactor Pre-Processing

The log information can be found at the following location:

```
zcui.work/xtor_<my_xtor>/zxtor/<my_xtor>/zxtor.log
```

The logs provide information about:

- Each inter-language functions. For each import/export function, the elements are listed in the following table:

TABLE 25 Log File Elements

Log File Element	Description
Name	Name of the function/task
Type	Function or task
in bits	Total size of input bits
out bits	Total size of output bits
Protocol	Name of the protocol belonging to the task (if defined)
streaming	Activation of the streaming (yes no)
Context	Activation of the context (yes no)
prefetchable	Activation of prefetching (yes no)

- Message ports intended for advanced optimization of transactional environment.

For example:

```

INFO: DPI exports summary:
INFO: +-----+-----+-----+-----+-----+
INFO: |      name | type | in bits | out bits | protocol | streaming
|
INFO: +-----+-----+-----+-----+-----+-----+
INFO: | stream_in | task |      32 |         0 |        -- |       Yes |
INFO: +-----+-----+-----+-----+-----+-----+
INFO:
INFO: DPI imports summary:
INFO: +-----+-----+-----+-----+-----+
INFO: | name |      type | in bits | out bits | context | streaming
| prefetchable |
INFO: +-----+-----+-----+-----+-----+-----+
INFO: +-----+-----+-----+-----+-----+-----+

```

Compiling the Hardware Part of a ZEMI-3 Transactor

```

INFO:
INFO: Message Ports Summary:
INFO: +-----+-----+-----+-----+
INFO: | name | type | size | count |
multiplex |
INFO: +-----+-----+-----+-----+
INFO: | streamer_in.stream_in_in_port | in | 32 | 1 | |
INFO: | streamer_in.b2b_in_port | in | 1 | 1 | |
INFO: | streamer_in.b2b_out_port | out | 1 | 1 | |
INFO: +-----+-----+-----+-----+
INFO:
INFO: Total Message In Ports: 2 (33 bits)
INFO: Total Message Out Ports: 1 (1 bits)

```

11.7.5 Compiling Output Files for the Software Part of the Transactor

ZeBu compilation generates two types of output files, described in the following sections.

11.7.5.1 Dynamic Libraries

For each ZEMI-3 transactor, the dynamic library must be created before starting runtime using the following commands.

```

make -f dpixtor.mk -C $(ZCUIWORK) /$(ZEBUWORK) clean all
      or
cd <zgui.work>/zebu.work
make -f dpixtor.mk clean all

```

Note

- These commands must be run on a PC running the same OS as the runtime host.
 - These commands must be re-executed if the OS of the runtime host changes.
-

11.7.5.2 Header File

For each ZEMI-3 transactor, the ZeBu compiler generates a `<my_xtor>.h` header file where `<my_xtor>` is the name of the ZEMI-3 transactor instantiated in the hardware top.

The header file is stored in the `zcui.work/zebu.work` directory.

This header file declares the following:

- The prototypes of the import and export functions of the transactor.
- The user transactor interface class, derived from the base class ZEMI-3 transactor.

This header file must be included in the software part of the transactor (see [Calling Export Functions/Tasks in C](#)) and in the testbench (see [Testbench Function](#)).

11.8 Running Emulation of ZEMI-3 Transactors

The ZEMI-3 environment is very similar to a simulation environment. A generic executable called `zEmiRun` handles the emulation automatically. You must provide the implementation of the import calls and optionally a testbench to control the emulation. A testbench is useful for export tasks that consume simulation time. The following figure illustrates the ZEMI-3 Runtime Environment.

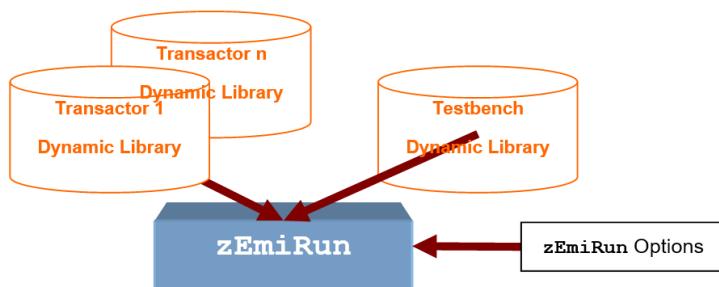


FIGURE 37. ZEMI-3 Runtime Environment

The following advanced runtime environments can also be used with a ZEMI-3 transactor

- An existing SystemC software testbench environment (see [Using a SystemC Testbench](#)).
- A ZCEI transactors along with ZEMI-3 transactors (see [Using ZCEI Transactors and ZEMI-3 Transactors](#)).

11.8.1 Prerequisites

The following are the prerequisites to run the ZEMI-3 environment:

- `zEmiRun` is a multi-threaded application, thus, it is highly recommended to run it on a multi-processor host to get the best performance.
- If the compilation is performed on a host configured differently from the one used for emulation runtime, the dynamic libraries can be independently recompiled with the `dpixtor.mk` makefile in the `zebu.work` directory. This makefile has two different targets:

- all: Compiles all dynamic libraries and copies them to `zebu.work`.
- clean: Cleans all the dynamic libraries.
- The following options can be used to override the makefiles generated by the compilation, if necessary:
 - `LDFLAGS` to add linker options [default: empty].
 - `CXXFLAGS` to add compiler options [default: `-O`].
 - `ZEMI3_LIB` to change the ZEMI-3 library [default: `zebuZEMI3`].
 - `XTOR_LIB` to rename the generated `*.so` file [default:<xtor_mod_name>.so].
- Ensure that none of the dynamic libraries loaded by `zEmiRun` are explicitly linked to `libZebu.so` or to `libZebuThreadssafe.so`. You can check using `ldd <libname>.so`.
- Ensure to remove all `-lZebu` and `-lZebuThreadsafe` when creating the `.so` file.

11.8.2 Preparing the Files Required for Runtime

You must provide the following for ZEMI-3 runtime:

- Implementation of the import calls.
- Implementation of the testbench function.

The import and export functions and the testbench function must be declared as `extern "C"`.

The source of the testbench must include the standard header file for SystemVerilog DPI (`svdpi.h`) and the transactor header files generated by the compilation (`<my_xtor>.h`) (see [Compiling Output Files for the Software Part of the Transactor](#)).

11.8.2.1 Testbench Function

You can provide a C function that executes export calls to control the emulation. This testbench function is called by the `ZEMI3Manager` object. When this function returns, the emulation terminates.

This testbench function must be declared as `extern "C"`. It can call export tasks that consume simulation time.

The testbench function prototype is as follows:

```
int my_testbench_function (int argc, char **argv)
```

 Running Emulation of ZEMI-3 Transactors

where, the `int` is the returned status of `zEmiRun`.

If no testbench function is provided and barring some other termination mechanism (for example, fixed time or number of clock cycles), the emulation runs indefinitely.

11.8.2.2 Compilation of the Dynamic Library

Your dynamic library must be linked with the `libZebuZEMI3.so` library.

The `-rdynamic` and `-fPIC` options should be used when linking with C++ code that contains import functions.

The following example displays how to compile dynamic library:

```
$ g++ -fPIC -c user_testbench.cc -Izcui.work/zebu.work/
  -I$ZEBU_ROOT/include
$ g++ -fPIC -rdynamic -shared -o user_testbench.so user_testbench.o
 \
  -L$ZEBU_ROOT/lib -lZebuZEMI3
```

11.8.3 Launching zEmiRun

Launch `zEmiRun` to run the ZEMI-3 environment. `zEmiRun` can be launched in different ways depending on your requirements, see [Standard Usage](#).

11.8.3.1 Standard Usage

You can launch `zEmiRun` with the name of the testbench function and its containing library with the `-m` option:

```
$ zEmiRun -m <user_dyn_lib>[:<tb_func>]
```

For example:

```
$ zEmiRun -m user_testbench.so:my_testbench_function
```

zEmiRun can also be launched without a testbench function:

```
$ zEmiRun -m user_testbench.so
```

11.8.3.2 Complete List of **zEmiRun** Options

zEmiRun can be launched as follows:

```
$ zEmiRun [options] [-- <user main arguments>]
```

where,

- [options] are options for running the ZEMI-3 transactor. They are described in the following table.
- [-- <user main arguments>] specifies arguments passed to the user main function (using -m option), initialization function (using -i option), and setup function (using -s option) as described in the following table).

TABLE 26 **zEmiRun** Options

zEmiRun option	Description
-z <zebu.work>	Specifies the directory for the design database. Default is ./zebu.work.
-f <designFeatures>	Specifies the file containing the runtime configuration. This file contains information regarding clocks and clock groups. There is no default.
-p <processName>	Specifies a name for the process. This is useful when running multi-process emulation. There is no default.
-v	Sets verbose mode. This mode prints more information messages during the run. Default is non-verbose (quiet).

TABLE 26 `zEmiRun` Options

<code>zEmiRun</code> option	Description
<code>-c <xtor_configuration></code>	<p>Specifies a configuration file for the ZEMI-3 transactors. The default file is the <code>zCui</code> generated file <code>./zebu.work/xtor_dpi.lst</code>. This file lists all the ZEMI-3 transactors to load with each transactor listed in a single line in the following format:</p> <pre><xtorModuleName> <xtorInstanceName> [<xtorDynamicLib>]</pre> <p>The dynamic library (or object) can be left unspecified if it is later specified by the <code>-x</code> option. The latter also overrides any C specification here.</p>
<code>-x <xtor_dyn_lib>:<xtor_module></code>	<p>Specifies the dynamic library (object, <code><xtor_dyn_lib></code> for a transactor module <code><xtor_module></code>) that contains the given transactor implementation.</p> <p>It is useful when the files are moved out of the standard location and the files specified in the transactor configuration point to the wrong (old) location.</p>
<code>-l <usr_dyn_lib></code>	<p>Loads the dynamic library (<code><usr_dyn_lib></code>) in the global scope. This library contains implementations of the import functions used by the transactors plus the optional <code>main/init</code>/callback functions, which are specified with the <code>(:<func>)</code> option.</p> <p>There is no default.</p> <p>Multiple <code>-l</code> options can be specified. If the library refers to any symbol from the transactor objects (example, export functions), ensure it is compiled/linked with the <code>-fPIC</code> compiler option; otherwise it fails to load.</p> <p>Note:</p> <p>The dynamic libraries specified with the <code>-m</code>, <code>-i</code> or <code>-b</code> options are loaded in the global scope, so this option is not necessary when all libraries needed are specified with one of the other options.</p>

TABLE 26 `zEmiRun` Options

<code>zEmiRun</code> option	Description
<code>-m <usr_dyn_lib>[:<tb_func>]</code>	<p>Specifies a function to be called as the user main function.</p> <p>The user main function is called in parallel with the ZEMI-3 service.</p> <p>If a function is not specified, then this option behaves like the <code>-l</code> option.</p> <p>There is no default.</p> <p>Only one <code>-m</code> option can be specified.</p> <p>The function is passed the command line arguments specified after the <code>'--'</code> flag. The return value of this function is used as the exit status from <code>zEmiRun</code>.</p> <p>The function is resolved within this library. The library is loaded globally and is available to resolve other global symbols.</p>
<code>-i [<usr_dyn_lib>:]<init_func></code>	<p>Specifies a transactor initialization function (<code><init_func></code>) to be called just after the opening the board and before starting the import processing. For example, set back-to-back mode, set prefetch mode, and so on. The optional (<code><usr_dyn_lib></code>) designates the dynamic library that includes the function.</p> <p>There is no default.</p> <p>The initialization function is called after a ZeBu <code>board::init</code> call and before the ZEMI-3 service. Calling export functions at this point is not prohibited, but might cause a deadlock situation if the message ports are not serviced.</p> <p>Multiple <code>-i</code> options can be specified.</p> <p>All the functions are passed the command line arguments specified after the <code>'--'</code> flag. If any function returns a non-zero value, <code>zEmiRun</code> treats it as a signal to abort the run.</p>

TABLE 26 zEmiRun Options

zEmiRun option	Description
<code>-l</code> [<usr_dyn_lib>:]<callback_func>	If a dynamic library (<usr_dyn_lib>) is specified, the function is resolved within that library. Thus, it can be used to designate a specific function. The library is loaded in the global scope and is available to resolve other global symbols. If the function is to be resolved globally, the dynamic library may be left unspecified and must instead be specified with one of the <code>-m</code> , <code>-b</code> or <code>-l</code> options. See the note in the <code>-l</code> option description about using the <code>-fPIC</code> compiler option.
<code>-b</code> [<usr_dyn_lib>:]<callback_func>	Specifies a function (<code>callback_func</code>) to call periodically during the emulation. Although the import processing is active at the time this function is called, deadlock is still possible by calling export functions are called from the callback function. This is a useful entry point for scoreboard or system monitoring operations. Although there are no guarantees about when the callbacks are called, the first call is always performed right before the import processing begins. There is no default. The callback function is called after the ZEMI-3 service shutdown and before the ZeBu <code>board::close</code> call. Multiple <code>-b</code> options can be specified. These functions are not passed any arguments. The return value (<code>int</code>) is interpreted as follows: <ul style="list-style-type: none"> • Zero (0): disable this callback • Non-Zero (N): reschedule this callback after N iterations of the service loop

TABLE 26 zEmiRun Options

zEmiRun option	Description
	<p>If a dynamic library (<usr_dyn_lib>) is specified, the function is resolved within that library. Thus, it can be used to designate a specific function. The library is loaded in the global scope and is available to resolve other global symbols.</p> <p>If the function is to be resolved globally, the dynamic library may be left unspecified and must be specified with one of the -m, -b or -l options.</p> <p>See the note in the -l option description about using the <code>-fPIC</code> compiler option.</p>
<code>-s [<usr_dyn_lib>:]<setup_func></code>	<p>Specifies a setup function (<setup_func>) to call after a ZeBu board “open” call and before a ZeBu board “init” call.</p> <p>There is no default.</p> <p>If a dynamic library (<usr_dyn_lib>) is specified, the function is resolved within that library. Thus it can be used to designate a specific function. The library is loaded in the global scope and is available to resolve other global symbols.</p> <p>If the function is to be resolved globally, the dynamic library may be left unspecified and must instead be specified with one of the -m, -b or -l options.</p> <p>See the note in the -l option description about using the <code>-fPIC</code> compiler option.</p>
<code>-n [threading_model]</code>	<p>Specifies the threading model. Using <code>-n</code> specifies that threads are not used.</p> <p>When <code>-n</code> is specified with one of the following arguments, the specified threading model is used:</p> <ul style="list-style-type: none"> • <code>xtor_with_service</code> • <code>xtor_only[:nb_of_threads]</code> • <code>xtor_only_with_wait[:<timeout_value>]</code> • <code>service_only</code> <p>This option is only available when there is no user main function (see <code>-m</code> option).</p>

TABLE 26 `zEmiRun` Options

<code>zEmiRun</code> option	Description
<code>-r <clock>:<cycle_number></code>	Specifies the number of cycles (<code><cycle_number></code>) to perform on clock <code><clock></code> before exiting. Applied only when there is no user main function (see <code>-m</code> option).
<code>-t <time in seconds></code>	Specifies a time in seconds before exit if no user main function was specified (see <code>-m</code> option description).
<code>-w <timeout></code>	Defines a waitgroup with a timeout value in microseconds. By default, there is no waitgroup.
<code>-y <loop_type></code>	<ul style="list-style-type: none"> Enables the yield in Rx/Tx loops. <code><loop_type></code> can be as follows: <code>rx</code>: the yield option is enabled in Rx loop only <code>tx</code>: the yield option is enabled in Tx loop only <code>all</code>: the yield option is enabled in both Rx and Tx loops <code>off</code> (default) : the yield option is disabled
<code>-o <log_filename></code>	Writes messages to the specified log file. The default log filename is <code>zEmiRun.log</code> .
<code>-h</code>	Prints the list of options with descriptions for <code>zEmiRun</code> .

Note

At least one dynamic library must be specified with either the `-l`, `-m`, `-i`, `-b` or `-s` options.

11.8.3.3 Launching `zRun` with `zEmiRun`

To use the ZeBu runtime and debug features, you can launch `zRun` with `zEmiRun` as follows:

```
$ zRun -testbench "zEmiRun -z ../../zebu.work
                         -m user_testbench.so:my_testbench_function"
```

11.8.3.4 Controlling the Profiling Feature

The profiling feature allows gathering data for runtime statistics, such as, the average number of `driverClock` cycles during which the software waits for a specific import.

These statistics are displayed to the screen and written to the emulation runtime log file.

For example:

```
# info : ZEMI3 Manager Statistics:
    Number of transactors          :          1
    Number of threads managed      :          0
    Total time in manager         : 0.000 s
    Number of iterations in service thread: 0
    Number of singleLoop calls     :          0

Statistics on transactor service loops:
    hw_top.x0 (xtor)           : 0 calls

Statistics on import and export calls:
    import myimport      (hw_top.x0)
        Number of loops       : 199468
        Number of calls       : 199468
        Words (32-bit) from hardware : 1595744 ( 0.000
Mb/s)
        Cycles waited in ZeBu for tx   : 0 ( 0.000
per call)
    import test_end      (hw_top.x0)
        Number of loops       : 1
        Number of calls       : 1
        Words (32-bit) from hardware : 1 ( 0.000
Mb/s)
```

Running Emulation of ZEMI-3 Transactors

```

        Cycles waited in ZeBu for tx      :          0 (  0.000
per call)
        Words (32-bit) to hardware      :          0 (  0.000
Mb/s)
        Cycles waited in ZeBu for rx      :          0 (  0.000
per call)
        export set_max_cycles (hw_top.x0)
        Number of loops      :          0
        Number of calls       :          1
        Words (32-bit) from hardware   :          1 (  0.000
Mb/s)
        Cycles waited in ZeBu for tx      :          0 (  0.000
per call)
        Words (32-bit) to hardware      :          1 (  0.000 Mb/s)
        export myexport      (hw_top.x0)
        Number of loops      :          0
        Number of calls       :         42
        Words (32-bit) to hardware   :        40 (  0.000 Mb/s)

```

Turning the Profiling Feature On

By default, profiling is ON.

Turning the Profiling Feature Off

Profiling requires extra hardware resources. To turn it off, add the `-do_profiling false` option to the UTF file. For example,

```
|zemi3 -module {list_of_zemi3__modules} -do_profiling false
```

11.9 Running Emulation of ZEMI-3 Transactors in a Heterogeneous Environment

The following advanced runtime environments can be used with a ZEMI-3 transactor:

- An existing SystemC testbench (see [Using a SystemC Testbench](#)). This mode gives access to a `ZEMI3Manager` object for emulation control.
- A zcei transactors along with ZEMI-3 transactors (see [Using ZCEI Transactors and ZEMI-3 Transactors](#)).

Note

If you wish to run your ZEMI-3 transactors in a 32-bit environment, contact your local representative.

11.9.1 Using a SystemC Testbench

In a SystemC environment, the runtime is handled by the SystemC kernel. Since `zEmiRun` uses its own scheduler similar to SystemC, they cannot be used together. In a SystemC environment, you must explicitly initialize the ZEMI-3 environment.

In such an environment, your testbench should use a C++ class that initializes the transactors, as described in the following sections.

11.9.1.1 Initializing the ZEMI-3 Environment Using the `ZEMI3Manager` Class

The `ZEMI3Manager` class initializes the ZEMI-3 environment. The interface of this class can be found in the `$ZEBU_ROOT/include/ZEMI3Manager.hh` file.

The following sections describe the procedure for environment initialization.

Creating the `ZEMI3Manager` Object

Create the `ZEMI3Manager` object by calling the `open` method as follows:

```
ZEMI3Manager* ZEMI3Manager::open (const char *zebu_work,  
                                const char *designFeatures,  
                                const char *processName");
```

Running Emulation of ZEMI-3 Transactors in a Heterogeneous Environment

where,

- zebu_work: Path to the working directory.
- designFeatures: Path to the designFeatures file.
- processName: The name of the process.

Note

You can open only one ZEMI3Manager object.

For example:

```
ZEMI3Manager *dm = open("./zebu.work", ". /designFeatures",
"myProcess");
```

Declaring the List of ZEMI-3 Transactors

A list of transactors is automatically generated in the zebu.work directory by `zfw` as `xtor_dpi.lst`. You must specify this list to the runtime environment as follows:

- When using a single-process, that is, only one ZEMI-3 transactor process is defined in your designFeatures file. Add the following line to your testbench:

```
void buildXtorList(const char *xtor_dpi.lst);
```

By default, this command uses the automatically generated `xtor_dpi.lst`.

For example:

```
manager->buildXtorList(xtor_dpi.lst);
```

or

```
manager->buildXtorList();
```

- When using multiple processes, that is, multiple ZEMI-3 transactor processes are defined in your designFeatures file (therefore as many testbenches as the number of processes). You must do the following:
 - a. Manually split this `xtor_dpi.lst` list to create one `.lst` file per process.

- b. Add the following line to each of your testbenches that declare separate lists for the runtime environment:

```
void buildXtorList(const char *<xtor_list_custom_n.lst>);
```

where, `<xtor_list_custom_n.lst>` is the name of the customized transactor list.

For example:

```
manager->buildXtorList(my_xtor_list01);
```

11.9.1.2 Initializing the `ZEMI3Manager` Object

The initialization starts the clocks and launches a thread for each transactor. The launched threads call the transactor imports. Start the initialization as follows:

```
dm->init(); // dm is the ZEMI3Manager object
```

Getting the `ZEBU::Board` Object With a `ZEMI3Manager` Object

When runtime features, such as, access to signals, logic analyzer, or clock control are required, you can retrieve the `ZEBU::Board` object from the `ZEMI3Manager` object as follows:

```
ZEBU::Board* getBoard();
```

For example:

```
Board *z = dm->getBoard();
```

Controlling the Back-to-Back Mode

It is possible to control the back-to-back mode from the `ZEMI3Manager` object.

To do so, use the functions described in [Controlling the Back-to-Back Mode From Your Testbench](#).

11.9.1.3 Handling Errors Inside the `ZEMI3Manager` Object

When errors are detected inside the `ZEMI3Manager` object, an exception is generated. This exception must be captured with a try-catch statement.

For example:

```
try{
    // initialization code here
}

catch (exception &xcp) {
    printf("### aborting %s - fatal error : %s.", argv[0],
xcp.what());
    exit(1);
}

catch (...) {
    printf("### aborting %s - fatal error... ", argv[0]);
    exit(1);
}
```

11.9.1.4 Compiling Libraries for the ZEMI-3 Environment

The final executable must be linked with the following libraries:

- The ZEMI-3 library: `libZebuZEMI3.so` with the `-lZebuZEMI3` option.
- The ZeBu library: `libZebu.so` or `libZebuThreadssafe.so` with the `-lZebu` or `-lZebuThreadssafe` option respectively.

The `libZebuThreadssafe.so` library must be used with a multi-threaded testbench.

It is recommended to use the `libZebuThreadssafe.so` library first if you want to perform your design bring-up safely. After you get the expected behavior, you can perform the bring-up again with the `libZebu.so` library to improve performance.

For example:

```
$ g++ -fPIC -c dpi_ctrl.cc -Izcui.work/zebu.work -I$ZEBU_ROOT/include  
$ g++ -rdynamic -o user_tb *.o -L$ZEBU_ROOT/lib \  
zcui.work/zebu.work/my_xtor_1.so zcui.work/zebu.work/my_xtor_2.so \  
\  
-lZebuZEMI3 -lZebuThreadsafe
```

11.9.2 Using ZCEI Transactors and ZEMI-3 Transactors

It is possible to mix ZEMI-3 transactors and ZCEI transactors. In this case, you have a ZCEI-based environment or a ZEMI-3-based environment. As in standard ZCEI-based environments, all ZeBu initializations are done through the `ZEBU::Board` class.

The programming interface for ZEMI-3 transactors is defined by a base class in the `ZEMI3Xtor.hh` header file located in the `$ZEBU_ROOT/include` directory.

The ZEMI-3 transactor base class allows the following:

- Connecting and disconnecting a ZEMI-3 transactor.
- Serving the imports.
- Starting and stopping back-to-back mode.

11.9.2.1 Using the ZEMI3Xtor Class

A testbench for the ZEMI-3 transactor must start with the following lines:

```
#include "ZEMI3Xtor.hh"  
using namespace ZEBU
```

The `ZEMI3Xtor.hh` file defines the `ZEMI3Xtor` class.

The `ZEMI3Xtor` class represents the ZEMI-3 transactor object. The methods associated with this class are described in the following table.

TABLE 27 ZEMI3Xtor Class Methods

Method	Description
ZEMI3Xtor	Constructor.
~ZEMI3Xtor	Destructor.
init	Initializes the ZEMI-3 transactor.
close	Closes the ZEMI-3 transactor.
needsService	Checks if any ZeBu imports need service.
reset	Restarts the processing of imports/exports.
terminate	Terminates any ongoing processing, including service loop.
setGroup	Specifies a group ID number for the ZEMI-3 transactor.
registerImports	Registers imports in Board::serviceLoop.
unregisterImports	Unregisters imports in Board::serviceLoop.
enableImports	Enables calling the streaming imports.
disableImports	Stops calling the streaming imports.
isImportDisabled	Indicates whether calling the streaming imports is enabled or not.
checkImports	Checks whether there is any message pending for any of the imports.
serviceLoop	Calls the ZEMI-3 transactor service loop.
serviceLoopWithWait	Calls the ZEMI-3 transactor service loop and returns only when an import call occurs or after the specified timeout.
startBackToBack	Starts the back-to-back mode for the ZEMI-3 transactor.
stopBackToBack	Stops the back-to-back mode for the ZEMI-3 transactor.
isBackToBackStarted	Indicates whether the back-to-back mode is started or not.
flushBufferedPorts	Flushes messages of the ports that are automatically buffered.
getXtorInfos	Returns information about the import from which this method is called.

TABLE 27 ZEMI3Xtor Class Methods

Method	Description
getErrorStatus	Indicates whether an error is detected inside the ZEMI-3 transactor.
getExportActiveStatus	Indicates whether an export task from the ZEMI-3 transactor is currently running.

init() Method

This method initializes the ZEMI-3 transactor and activates its ports. It should be called before the ZeBu Board::init initialization.

```
void init (Board *zebu, const char *instancePath) ;
```

where,

- zebu is the pointer to the ZEBU::Board object.
- instancePath is the path to the instance of the ZEMI-3 transactor.

close() Method

This method closes the ZEMI-3 transactor and deactivates its ports.

```
void close () ;
```

needsService() Method

This method checks whether there are any ZeBu imports that need service through a service loop.

```
bool needsService () const;
```

This method returns the following:

- true: Import needs service.
- false: No import needs for service.

reset() Method

This method re-starts the processing of imports/exports after a terminate method.

```
void reset () ;
```

terminate() Method

This method terminates any ongoing processing. This also stops any ongoing service loop.

```
void terminate () ;
```

setGroup() Method

This method sets a group ID number for the ZEMI-3 transactor. The group ID can be used with `Board::serviceLoop` to serve the imports.

```
void setGroup(unsigned int portGroup);
```

where, `portGroup` is the group ID number.

registerImports() Method

This method allows registering imports in `Board::serviceLoop`. Thus, when `Board::serviceloop` is called, the ZEMI-3 transactor's imports are called.

```
void registerImports(unsigned int portGroup);
```

where, `portGroup` is optional. It is the group ID number used in `Board::serviceLoop` if any. It allows associating a specific import to the specified group of transactors.

unregisterImports() Method

This method unregisters the imports registered with `registerImports()`. These imports are no longer called by `Board::serviceLoop`.

```
void unregisterImports();
```

enableImports () Method

This method allows calling the streaming imports if they were previously disabled.

```
void enableImports () ;
```

disableImports () Method

This method stops calling the streaming imports if they were previously enabled.

```
void disableImports () ;
```

isImportDisabled() Method

This method checks whether calling streaming imports is allowed or not.

```
bool isImportDisabled () ;
```

This method returns the following:

- true: Calling is disabled.
- false: Calling is enabled.

checkImports () Method

This method checks whether there is any pending message for any of the imports.

```
bool checkImports () ;
```

This method returns the following:

- true: there is a pending message.
- false: there is no pending message.

serviceLoop () Method

This method calls the pending ZEMI-3 transactor's import functions, if any. The imports requested by the software are executed.

```
void serviceLoop () ;
```

serviceLoopWithWait() Method

This method calls the ZEMI-3 transactor service loop and returns only when one import call is executed or after the specified timeout.

```
void serviceLoopWithWait (int timeout=0) ;
```

where, timeout is the value of the timeout in seconds.

startBackToBack() Method

This method enables Back-to-Back mode for the ZEMI-3 transactor.

```
void startBackToBack () ;
```

stopBackToBack() Method

This method disables Back-to-Back mode for the ZEMI-3 transactor.

```
void stopBackToBack () ;
```

isBackToBackStarted() Method

This method checks whether Back to Back mode is enabled or not.

```
void isBackToBackStarted () ;
```

This method returns the following:

- true when this mode is enabled.
- false when this mode is disabled.

flushBufferedPorts() Method

This method flushes import/export requests pending on the ports that are automatically buffered by the system. It ensures that all pending requests are processed.

```
void flushBufferedPorts () ;
```

getXtorInfos () Method

This method returns the following information about the import, from which it is called, in the `xtorInfos` structure:

- `function`: The name of the import function.
- `scope`: Scope of the function call (for example, `xtor.ins`).
- `filename`: The name of the Verilog file in which the call is coded.
- `line`: Number of the line concerned in the file pointed by `filename`.

The syntax for this method is:

```
xtorInfos* getXtorInfos() ;
```

getErrorStatus () Method

This method indicates if an error is detected inside the ZEMI-3 transactor.

```
bool getErrorStatus () ;
```

This method returns the following:

- `true`: an error is detected.
- `false`: no error detected.

getExportActiveStatus () Method

This method indicates whether an export task is currently running for the ZEMI-3 transactor.

```
bool getExportActiveStatus () ;
```

This method returns the following:

- `true`: an export task is running.
- `false`: no export task is running.

11.9.2.2 Using a ZEMI-3 Transactor in a ZCEI Testbench

You must include in the testbench the `<my_xtor>.h` header file, generated by `zcui` in the `zebu.work` directory. This file declares your transactor interface class, which is derived from the base class `ZEMI3Xtor`. To handle the ZEMI-3 transactor, an object of this transactor class has to be instantiated in the testbench.

For example:

```
namespace ZEMI3_USER
{
    class trans : public ZEMI3Xtor
    {
        public:
            trans();
    };
}
```

Creating the Transactor Object

You must create an object for each instance of the transactor instantiated in the hardware top file. The object must be created between the calls to `ZEBU::Board::open` and `ZEBU::Board::init`.

For example:

```
// Board open
Board *z = Board::open("../zebu.work");

// Creation of the transactor objects
ZEMI3_USER::trans *t0 = new ZEMI3_USER::trans ();
ZEMI3_USER::trans *t1 = new ZEMI3_USER::trans();
```

Initializing a ZEMI-3 Transactor

Use the `init()` method to initialize the transactor. This method connects the transactor's ports.

The `init()` must be called after creation of the transactor object and before `ZEBU::Board::init`. The arguments are the Board object and the transactor instance name (in the hardware top). For details about this method, see [init\(\) Method](#).

For example:

```
// Init of the streamers  
t0->init(z, "trans0");  
t1->init(z, "trans1");  
  
// Init of zebu  
z->init();
```

11.9.2.3 Running a ZEMI-3 Transactor

When interfacing a ZEMI-3 transactor with a ZCEI environment, the testbench explicitly calls the export functions and transactor service-loop method services the import functions.

In a heterogeneous environment, one of the following ways is supported to call the imports:

- Using the transactor local `ZEMI3Xtor::serviceLoop()` call, which directly checks if there are messages available on the message port(s).
- Registering a callback for the import. The callback is executed during the `Board::serviceLoop()` method. The `ZEMI3Xtor::registerImports()` method does the callback registration, if desired.

Thus, you can control and decide how the transactor imports should be handled.

Note

These two modes cannot be used simultaneously for the same transactor.

Using the Local Transactor `serviceLoop`

When called, `ZEMI3Xtor::serviceLoop` checks if an import is called from the hardware part of the transactor. If that is the case, it calls the corresponding user import function. The advantage of this method is that it can be called in a separate thread serving only a specific transactor import. When using an application with one thread for each transactor, `ZEMI3Xtor::serviceLoop` should be used.

For example:

In the following example, there are two transactors and one thread is created for each of them.

```
bool TEST_END = false;

// Thread function: Server of import
void xtorImportServer(ZEMI3Xtor *xtor)
{
    while(!TEST_END) xtor->serviceLoop();
}

int main()
{
    Board *z = Board::open();
    // Initialization: two instances of the same my_zemi3_xtor
    transactor
    ZEMI3Xtor *my_handle0 = new my_zemi3_xtor();
    ZEMI3Xtor *my_handle1 = new my_zemi3_xtor();
    my_handle0->init(z, "my_zemi3_xtor_0");
    my_handle1->init(z, "my_zemi3_xtor_1");
    // Initialization of the zcei transactor
    ZCEI_xtor *my_zcei_xtor_0 = new ZCEI_xtor(z, "my_zcei_xtor_0");
```

```
// ZeBu init
z->init();
// Launching threads for the ZEMI-3 transactors
pthread_t tid1;
pthread_create(&tid1, NULL, (void *(*)(void *))xtorImportServer,
(void *) my_handle0);
pthread_t tid2;
pthread_create(&tid2, NULL, (void *(*)(void *))xtorImportServer,
(void *) my_handle1);
// testbench execution
...
// disconnection
TEST_END = true;
pthread_join(tid1);
pthread_join(tid2);
my_handle0->close();
my_handle1->close();
z->close();
}
```

Using the ZEBU::Board serviceLoop

You must register the imports of each transactor that you wish to be handled by Board::serviceLoop. This provides some flexibility.

Here is an example indicating where the register call should be placed.

```
bool TEST_END = false;
// Thread function: Server of import
void xtorImportServer(Board *z)
{
```

Running Emulation of ZEMI-3 Transactors in a Heterogeneous Environment

```
    while(!TEST_END) z->serviceLoop();  
}  
  
int main()  
{  
    Board *z = Board::open();  
  
    // Initialization: two instances of the same my_zemi3_xtor  
    transactor  
    ZEMI3Xtor * my_handle0 = new my_zemi3_xtor();  
    ZEMI3Xtor * my_handle1 = new my_zemi3_xtor();  
    my_handle0->init(z, "my_zemi3_xtor_0");  
    my_handle0->registerImports ((int)my_handle0);  
    my_handle1->init(z, "zemi3_xtor_1");  
    my_handle1->registerImports ((int)my_handle1);  
    // Initialization of the zcei transactor  
    ZCEI_xtor *zcei_xtor_0 = new ZCEI_xtor(z, "zcei_xtor_0");  
    // ZeBu init  
    z->init();  
    // Launching board thread  
    pthread_t tid;  
    pthread_create(&tid, NULL, (void *(*)(void *))xtorImportServer,  
    (void *)z);  
    // testbench execution  
    ...  
    // disconnection  
    TEST_END = true;  
    pthread_join(tid);
```

```
    my_handle0->close();  
    my_handle1->close();  
    z->close();  
}
```

11.10 Performing Simulation of ZEMI-3 Transactors

A ZEMI-3 transactor and its standalone environment can be simulated with any SystemVerilog-compliant simulator such as VCS.

11.10.1 Writing a Top-Level Wrapper

The connection between the clock ports, the DUT, and the transactor is performed through the same hardware top file.

For example:

```
module hw_top();  
    wire [7:0] mem0_addr;  
    wire [31:0] mem0_din;  
    wire [31:0] mem0_dout;  
    wire mem0_we;  
    wire [7:0] mem1_addr;  
    wire [31:0] mem1_din;  
    wire [31:0] mem1_dout;  
    wire mem1_we;  
    reg clk;  
    initial clk <= 0;  
    always #10 clk <= !clk;  
    dut dut0(
```

Performing Simulation of ZEMI-3 Transactors

```
.clk(clk),
.mem0_addr(mem0_addr),
.mem0_din(mem0_din),
.mem0_dout(mem0_dout),
.mem0_we(mem0_we),
.mem1_addr(mem1_addr),
.mem1_din(mem1_din),
.mem1_dout(mem1_dout),
.mem1_we(mem1_we)
);
mem_xtor mem_xtor_0 (.clk(clk),
.mem0_addr(mem0_addr),
.mem0_din(mem0_din),
.mem0_dout(mem0_dout),
.mem0_we(mem0_we),
.mem1_addr(mem1_addr),
.mem1_din(mem1_din),
.mem1_dout(mem1_dout),
.mem1_we(mem1_we)
);
endmodule
```

11.10.2 Using a Software Entry Point

The testbench function is not available in simulation as it is in emulation.

To create a testbench function in simulation, you can add the following code to the hardware part of the transactor:

```
`ifdef SIMULATION
    import "DPI-C" context task testbench();
    initial begin
        testbench();
        $finish;
    end
`endif
```

11.10.3 Launching the Simulation

The filenames used in this section match the memory transactor example described in [Writing a Top-Level Wrapper](#).

To launch the simulation using VCS, perform the following steps:

1. Create a `mem_xtor.h` header file that includes the prototypes of the export functions:

```
#include <svdpi.h>
extern "C" export (...);
#define XTOR_SCOPE "hw_top.mem_xtor_0"
```

2. Compile the transactor for the HDL simulator, proceed as follows:

```
$ vcs ../src/*.sv ../src/*.v ../src/tb.cc -sverilog -R
```

3. Run the simulation, proceed as follows:

```
$ ./simv
```

12 Hybrid Emulation

Hybrid emulation entails running part of a design on a hardware emulator and another part of the design on a software simulator using Transaction-Level Models (TLM). The ZeBu hybrid emulation solution consists of the following three components:

- A virtual platform that hosts fast models.
- The PCT/TLM2 adapter that acts as an interface between the virtual platform and an AMBA ZeBu transactor.

NOTE: *For any bus other than AMBA, a hardware bridge is recommended instead of the PCT/TLM2 adapter.*

- The DUT platform mapped onto ZeBu.

The following figure displays the components of the ZeBu hybrid emulation.

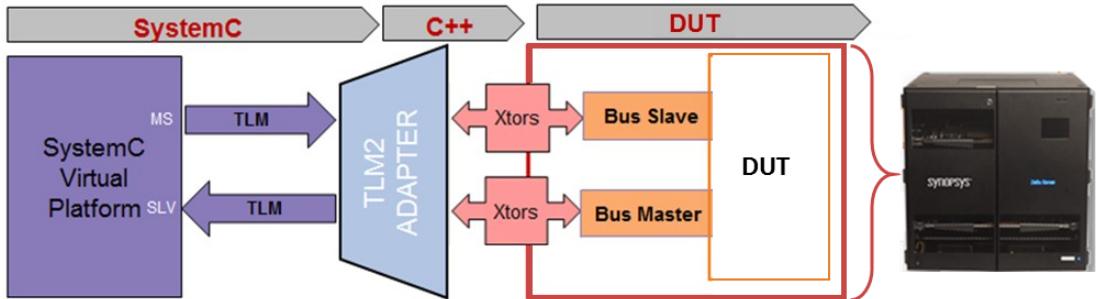


FIGURE 38. ZeBu Hybrid Emulation Platform

This chapter only describes the mapping of the DUT platform on ZeBu. This chapter discusses the following topics:

- *Hardware Top Preparation*
- *UTF Commands for Hybrid Emulation*

12.1 Hardware Top Preparation

This section consists of the following sub-sections:

- *AMBA Transactors*
- *SideBand Signals*
- *LPDDRx Shared Memory*

12.1.1 AMBA Transactors

In the hardware top, one of the AMBA transactors (ACELite, AXI4, AXI3, AHB, or APB) must be instantiated to establish the protocol link with the virtual platform. Cross-module references (XMRs) can be used to connect an AMBA bus directly inside a DUT to the transactor.

12.1.2 SideBand Signals

SideBand signals are provided in the PCT/TLM2 adapter and they connect interrupts between the DUT and the virtual platform. These signals are usually instantiated in the hardware top and connected to the interrupt ports using XMRs or to reset signals in the hybrid platform if required.

12.1.3 LPDDRx Shared Memory

The flow of the LPDDRx shared memory is similar to AMBA transactors. That is, it is instantiated in the hardware top and connected using XMRs.

12.2 UTF Commands for Hybrid Emulation

This section consists of the following sub-sections:

- *Black-Boxing a Design During Synthesis*
- *Defining the zTLM_RESETn Signal*

12.2.1 Black-Boxing a Design During Synthesis

Some design components run in the TLM as fast models, thus, it is useful to black box them to reduce the ZeBu mapping size. To black a Virtual ARM core, use the following command:

```
synthesis -blackbox CORTEXA53
```

12.2.2 Defining the zTLM_RESETn Signal

It is mandatory to declare a reg ZTLM_RESETn signal to control the DUT reset from the TLM. Use the following UTF command to designate the existing Verilog variable as the reg ZTLM_RESETn signal:

```
set_sw_control_signal -hdl_path HWTOP.ZTLM_RESETn_int -name  
ZTLM_RESETn+
```

The hardware top must have a Verilog reg declaration of the ZTLM_RESETn_int attribute.

13 Design Partitioning With zManualPartitioner

ZeBu provides a graphical user interface, **zManualPartitioner** that allows manual partitioning of specific parts of a design. The interface allows drag and drop to manipulate design instances after compilation.

The manual partitioning also exists in **zNetgen**, a Tcl-based ZeBu tool.

This chapter discusses the following topics:

- *User Interface*
- *Managing Black-boxes*
- *Moving Instances Between Partitions*
- *Limitations*

13.1 User Interface

This section explains how to use the **zManualPartitioner** GUI. This section consists of the following sub-sections:

- [*Launching zManualPartitioner*](#)
- [*Loading a Netlist File*](#)
- [*Main Window*](#)
- [*Partition Pane*](#)
- [*Partition Interconnection Matrix Pane*](#)
- [*Instance Interconnection Matrix Pane*](#)
- [*Timing Analysis Pane*](#)
- [*Configuration Window*](#)
- [*Properties Window*](#)
- [*Loading and Generating Mapping Files*](#)

13.1.1 Launching zManualPartitioner

To launch the **zManualPartitioner** GUI, use the following command:

```
$ zManualPartitioner
```

This command opens the following welcome screen:

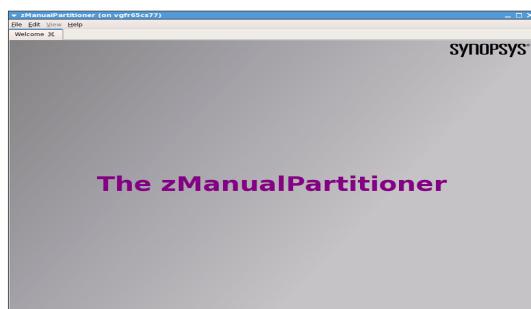


FIGURE 39. ManualPartitioner Welcome Screen

13.1.2 Loading a Netlist File

You can load two types of netlist files into the **zManualPartitioner** GUI:

- EDIF
- Automatic Clustering Interconnection

This section consists of the following sub-sections:

- *Loading an EDIF File*
- *Loading an Automatic Clustering Interconnection File*

13.1.2.1 Loading an EDIF File

To load an EDIF file, perform the following steps:

1. Select **File > Open EDIF** file.
2. Browse to your EDIF file. Extensions accepted are .edf, .edf.gz, and .edif.
3. Click **Open**.

NOTE: You can load only one EDIF file at a time in a **zManualPartitioner** session.

13.1.2.2 Loading an Automatic Clustering Interconnection File

You can load an interconnection file in **zManualPartitioner** if you have compiled your design at zCore-Level with Automatic Clustering enabled.

Note

*You can load multiple Automatic Clustering interconnection files in parallel in a **zManualPartitioner** session.*

To load an Automatic Clustering Interconnection file, perform the following steps:

1. Click **File > Open AC file** to open the following dialog box:

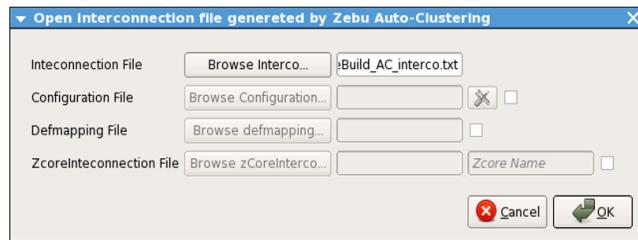


FIGURE 40. Loading an Automatic Clustering Interconnection File

NOTE: To retrieve an Interconnection file for a design, generated at `zCoreBuild` level, you must activate automatic clustering at the `zTopBuild` level.

2. Click **Browse Interco...** to select your Automatic Clustering Interconnection file.

In addition to the Automatic Clustering Interconnection file, you can also load a configuration file, a defmapping file, and a ZcoreInterconnection file to reproduce and modify partitioning results as needed. By default, these fields are inactive in the window. To activate these fields, click the check box next to them (see [Figure 40](#)). For more details on these files, see the following table:

TABLE 28 Additional Files to Modify Partitioning Results

Files	Description	Mandatory /Optional
Integration Files	<p>This is the <code>zCoreBuild_AC_interco.txt</code> file that is generated by automatic clustering after compilation at zCore-level (<code>zCoreBuild</code>). It can also be generated by the system-level compiler (<code>zTopBuild</code>).</p> <p>This file does not require the EDIF file to perform partitioning.</p> <p>It is available in your zCore-level working directory, that is, <code>work.<core_name></code>.</p>	Mandatory
Configuration Files	<p>This is a target configuration file that selects the ZeBu configuration in the <code>\$ZEBU_ROOT/etc/configurations</code> directory.</p> <p>To load a configuration file, select the corresponding check box and click the Browse Configuration button.</p>	Optional
	<p>Note: If you click the  button, the Configuration window appears, which allows you to customize the configuration. For more information, see Configuration Window.</p>	
Defmapping File	<p>This is the <code>zCoreBuild_AC_defmapping.tcl</code> file associated with the <code>zCoreBuild_AC_interco.txt</code> file. It contains the results of a previous partitioning.</p> <p>This file is available in your zCore-level working directory, that is, <code>work.<core_name></code>.</p>	Optional
zCoreInterconnection File	<p>This is the <code>zTopBuild_AC_TOPNL_interco.txt</code> file that describes the interconnection information between all <code>zCores</code>. This file is generated by the system-level compiler (<code>zTopBuild</code>) with the cluster enable <code>-map_inter_core_io</code> command.</p>	Optional

13.1.3 Main Window

The **zManualPartitioner** main window changes depending on the file you load, that is, an EDIF file or an Automatic Clustering Interconnection file.

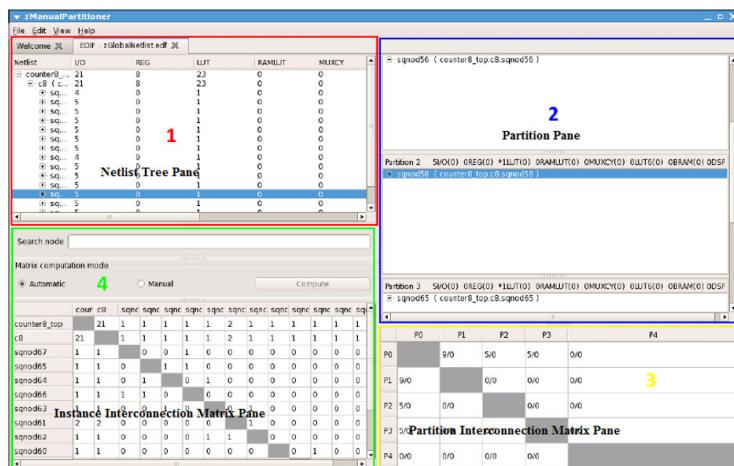


FIGURE 41. EDIF File loaded in zManualPartitioner

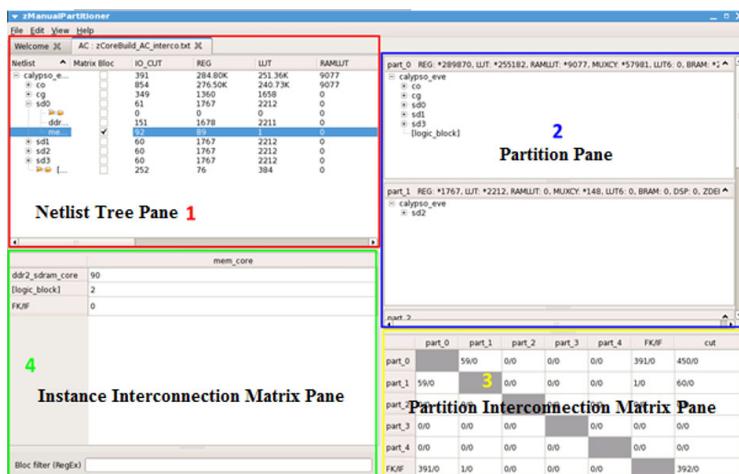


FIGURE 42. Automatic Clustering Interconnection File loaded in zManualPartitioner

User Interface

The main window can be divided into the following four panes, (see [Figure 41](#) and [Figure 42](#)).

- [Netlist Tree Pane](#)
- [Partition Pane](#)
- [Partition Interconnection Matrix Pane](#)
- [Instance Interconnection Matrix Pane](#)

The **Instance Interconnection Matrix** pane is replaced by the **Timing Analysis** pane when you launch the **Timing Analysis** feature. For more details, see [Timing Analysis Pane](#).

13.1.4 Netlist Tree Pane

The **Netlist Tree** pane allows you to view all the available instances in a netlist file. The pane changes depending on the file you load into **zManualPartitioner**.

13.1.4.1 Using an EDIF File

The following figure displays the **Netlist Tree** pane, when you load an EDIF file in **zManualPartitioner**.

Netlist	I/O	REG	LUT	RAMLUT	MUXCY	LUT6	BRAM	DSP	Partition	Matrix Selection
counter8_top	21	8	23	0	0	0	0	0	0	<input checked="" type="checkbox"/>
c8 (counter8)	21	8	23	0	0	0	0	0	0	<input checked="" type="checkbox"/>
+ sqnod67 ...	4	0	1	0	0	0	0	0	0	<input checked="" type="checkbox"/>
+ sqnod66 ...	5	0	1	0	0	0	0	0	0	<input checked="" type="checkbox"/>
+ sqnod65 ...	5	0	1	0	0	0	0	0	3	<input checked="" type="checkbox"/>
+ sqnod64 ...	5	0	1	0	0	0	0	0	0	<input checked="" type="checkbox"/>
+ sqnod63 ...	5	0	1	0	0	0	0	0	0	<input checked="" type="checkbox"/>
+ sqnod62 ...	5	0	1	0	0	0	0	0	1	<input checked="" type="checkbox"/>
+ sqnod61 ...	5	0	1	0	0	0	0	0	0	<input checked="" type="checkbox"/>
+ sqnod60 ...	4	0	1	0	0	0	0	0	0	<input checked="" type="checkbox"/>
+ sqnod59 ...	5	0	1	0	0	0	0	0	0	<input checked="" type="checkbox"/>
+ sqnod58 ...	5	0	1	0	0	0	0	0	2	<input checked="" type="checkbox"/>
+ sqnod57 ...	5	0	1	0	0	0	0	0	0	<input checked="" type="checkbox"/>
+ sqnod56 ...	5	0	1	0	0	0	0	0	1	<input checked="" type="checkbox"/>
+ sqnod55 (...)	5	0	1	0	0	0	0	0	0	<input type="checkbox"/>
+ sqnod54 ...	5	0	1	0	0	0	0	0	0	<input type="checkbox"/>
+ Orini: 102 0	0	0	0	0	0	0	0	0	0	<input type="checkbox"/>

FIGURE 43. Netlist Tree Pane (EDIF File)

The following table describes the columns available in the **Netlist Tree** pane.

TABLE 29 Column Description in the Netlist Tree Pane (EDIF File)

Column	Description
Netlist	Displays the list of the following available items in the netlist in a tree view: <ul style="list-style-type: none"> • The top of a design. • Common instances followed by their module name • Logical gates. • Black box in the original design or in zManualPartitioner. • (For more information on black-box management in zManualPartitioner, see Managing Black-boxes). • Primitives containing logical gates and black-boxes at the same level.
I/O	Displays the number of inputs/outputs.
REG	Displays the number of registers.
LUT	Displays the number of lookup tables.
RAMLUT	Displays the number of RAMLUTs.
MUXCY	Displays the number of multiplexers for Carry Logic.
LUT6	Displays the number of LUT6s.
BRAM	Displays the number of BRAMs.
DSP	Displays the number of DSPs.
Partition	Partition number where an instance is currently located. It is dynamically updated when you move an instance from one partition to another.
Matrix Selection	A Check box that allows you to select as many items as necessary to display in the Instance Interconnection Matrix pane.

User Interface

The following table displays the icons available in the **Netlist Tree** pane.

TABLE 30 Available Icons in the Netlist Tree Pane (EDIF File)

Column	Description
	Black box is already set in the netlist. For more information on black-box management in zManualPartitioner , see Managing Black-boxes .
	Black box set in zManualPartitioner . For more information on black-box management in zManualPartitioner , see Managing Black-boxes .
	Primitives.

13.1.4.2 Using an Automatic Clustering Interconnection File

The following figure displays the Netlist Tree pane when you load an *Automatic Clustering Interconnection* file in **zManualPartitioner**.

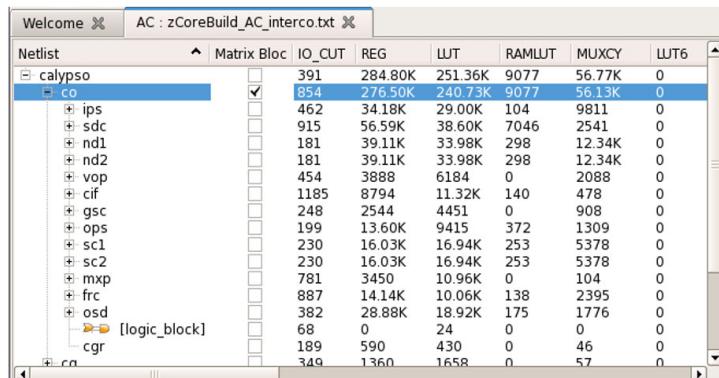


FIGURE 44. Netlist Tree Pane (Automatic Clustering Interconnection File)

The following table describes the columns available in the Netlist Pane when you load *Automatic Clustering Interconnection* file in **zManualPartitioner**.

TABLE 31 Netlist Tree Pane Column Description (Automatic Clustering Interconnection File)

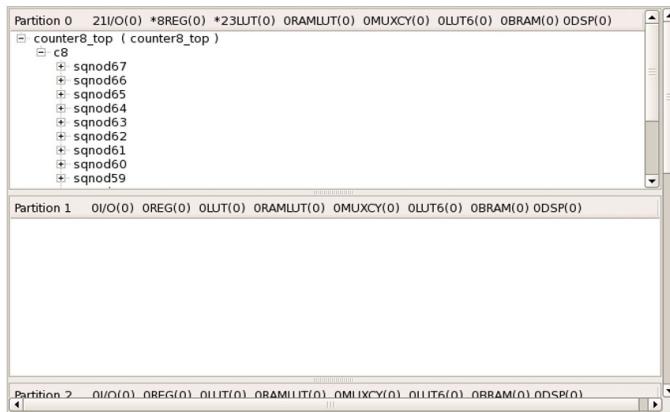
Column	Description
Netlist	Displays the list of the following available items in the netlist organized in a tree view: <ul style="list-style-type: none"> ● Top of a design ● Logic blocks ● Common instances followed by their module name
Matrix Block	A check box that allows you to select an item to display in the Instance Interconnection Matrix pane. Only one item can be selected at a time.
IO_CUT	Displays the number of inputs/outputs.
REG	Displays the number of registers.
LUT	Displays the number of LUTs.
RAMLUT	Displays the number of RAMLUTs.
MUXCY	Displays the number of MUXCYS.
LUT6	Displays the number of LUT6s.
BRAM	Displays the number of BRAMs.
DSP	Displays the number of DSPs.
ZDELAY	Displays the number of zDelays.
ZRM	Displays the number of ZRMs.

13.1.5 Partition Pane

When you load a netlist file in **zManualPartitioner**, both the **Netlist Tree** pane and the **Partition** pane display the file.

The **Partition** pane allows you to view the available partitions and the instance placement. The information in the header displayed in this pane is the same as the information in the columns of the **Netlist Tree** pane (see [Figure 45](#) and [Figure 46](#)).

User Interface

**FIGURE 45.** Partition Pane (An EDIF File)

	part_0	part_1	part_2	part_3	part_4	FK/IF	cut
part_0	58/0	57/0	57/0	57/0	391/0	620/0	
part_1	58/0		2/0	2/0	2/0	1/0	65/0
part_2	57/0	2/0		0/0	0/0	1/0	60/0
part_3	57/0	2/0	0/0		0/0	1/0	60/0
part_4	57/0	2/0	0/0	0/0		1/0	60/0
FK/IF	391/0	1/0	1/0	1/0	1/0		395/0

FIGURE 46. Partition Pane (An Automatic Clustering Interconnection File)

A netlist is loaded in Partition 0 by default. However, you may drag and drop instances from one partition to another in the **Partition** pane.

For more information about moving instances between partitions, see [Moving Instances Between Partitions](#).

13.1.5.1 Adding or Removing Partitions From the Partition Pane

You can add or remove as many partitions as necessary in the **Partition** pane from the **Properties** dialog box.

To add or remove partitions in the **Partition** pane, perform the following steps:

1. From the menu bar, click **Edit > Preferences mapping** to open the **Properties** dialog box.
2. Add or remove partitions using the **Number of partitions** field on the **Properties** dialog box as required.

13.1.5.2 Hiding or Displaying Partitions

You can hide or display as many partitions as necessary in the **Partition** pane by selecting **View > View N**.

A view is a partition in the **Partition** pane and a check symbol next to a view means its corresponding partition is displayed, see the following figure:

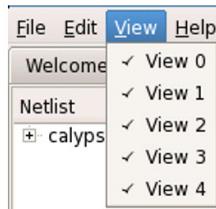


FIGURE 47. View Menu

13.1.6 Partition Interconnection Matrix Pane

The **Partition Interconnection Matrix** pane displays information about the number of cuts between partitions in a table. The table (*Figure 48*) displays one column and one row for each partition.

13.1.6.1 Using an EDIF File

The following figure displays the **Partition Interconnection Matrix** pane when you load an EDIF file.

User Interface

	part_0	part_1	part_2	part_3	part_4	FK/IF	cut
part_0		58/0	57/0	57/0	57/0	391/0	620/0
part_1	58/0		2/0	2/0	2/0	1/0	65/0
part_2	57/0	2/0		0/0	0/0	1/0	60/0
part_3	57/0	2/0	0/0		0/0	1/0	60/0
part_4	57/0	2/0	0/0	0/0		1/0	60/0
FK/IF	391/0	1/0	1/0	1/0	1/0		395/0

FIGURE 48. Partition Interconnection Matrix Pane (EDIF File)

Each cell displays information such as, for instance, 4/0 or 9/0. These figures have the following meaning:

- The first figure is the number of cuts between partitions.
- The second figure is the maximum number of cuts allowed by the target configuration between partitions.

13.1.6.2 Using an Automatic Clustering Interconnection File

The following figure displays the **Partition Interconnection Matrix** pane when you load an *Automatic Clustering Interconnection* file.

	part_0	part_1	part_2	part_3	part_4	FK/IF	cut
part_0		58/0	57/0	57/0	57/0	391/0	620/0
part_1	58/0		2/0	2/0	2/0	1/0	65/0
part_2	57/0	2/0		0/0	0/0	1/0	60/0
part_3	57/0	2/0	0/0		0/0	1/0	60/0
part_4	57/0	2/0	0/0	0/0		1/0	60/0
FK/IF	391/0	1/0	1/0	1/0	1/0		395/0

FIGURE 49. Partition Interconnection Matrix Pane (Automatic Clustering Interconnection File)

Each cell displays information such as, for instance, 57/0 or 58/0. These figures have the following meaning:

- The first figure is the number of cuts between partitions.
- The second figure is the maximum number of cuts allowed by the target configuration between partitions.

The **Cut** column displays the cut between the current partition and the others.

Depending on the results of the automatic clustering, the column **FK/IF** may be displayed to give information on the interconnection between the current partition and the external interface IF.

13.1.7 Instance Interconnection Matrix Pane

The **Instance Interconnection Matrix** pane computes the interconnection between adjacent instances.

13.1.7.1 Using an EDIF File

The matrix consists of one column and one row for each instance selected, using the check box in the **Matrix Selection** column in the **Netlist Tree** pane (see [Figure 50](#)). You can select as many instances, as needed.

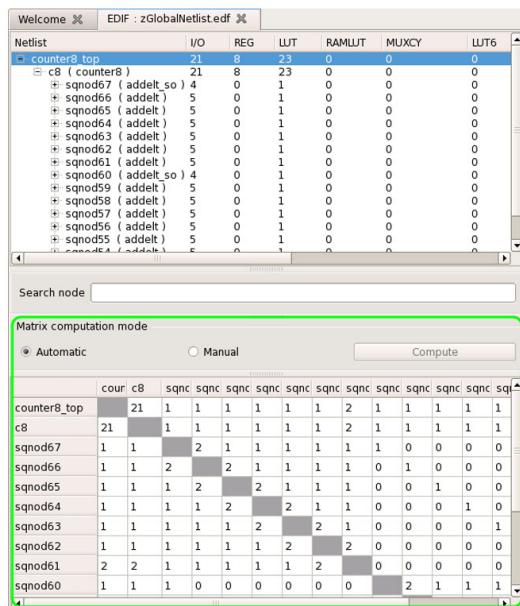


FIGURE 50. Instance Interconnection Matrix Pane (An EDIF File)

The **Matrix Computation Mode** allows you to control how the instance interconnection is calculated:

- **Automatic:** It is automatically calculated every time you must select an instance in the **Netlist Tree** pane.
- **Manual:** You first select all instances from the **Netlist Tree** pane and then click **Compute** for manual calculation.

13.1.7.2 Using an Automatic Clustering Interconnection File

When you load an *Automatic Clustering Interconnection* file, only the interconnection of the selected instance (through the **Matrix Block** check box) and its neighbors can be displayed in the **Instance Interconnection Matrix** pane.

To display the interconnection of an instance, go to the **Netlist Tree** pane and select the check box in the **Matrix Block** column (see [Figure 51](#)).

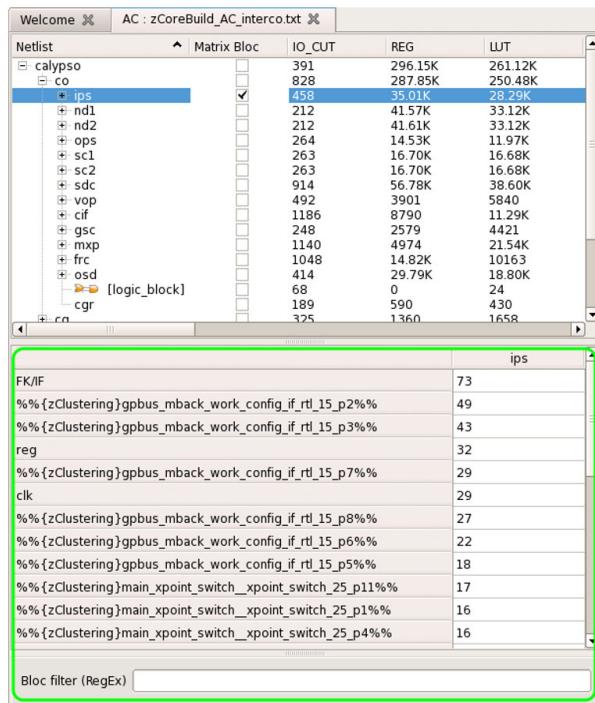


FIGURE 51. Instance Interconnection Matrix Pane (Automatic Clustering Interconnection File)

When an instance has too many adjacent instances, you can filter them using the **Bloc Filter (RegEx)** field.

13.1.8 Timing Analysis Pane

The **Timing Analysis** pane allows you to view the structure of combinational paths, that is, the set of successive instances in the combinational path for each partition.

Note

The timing analysis feature is only available when you use an Automatic Clustering Interconnection file in zManualPartitioner.

User Interface

With a clear view of these paths, you can enhance performance by selecting long combinational paths across a minimum number of partitions.

To display the **Timing Analysis** pane, perform the following steps:

1. Open an *Automatic Clustering Interconnection* file in **zManualPartitioner**.
2. Click **Edit > Timing Analyze** to replace the **Instance Interconnection Matrix** pane with the **Timing Analysis** pane. By default, it is set on the **Timing Analyze/Clk Path view** (see [Figure 52](#)):

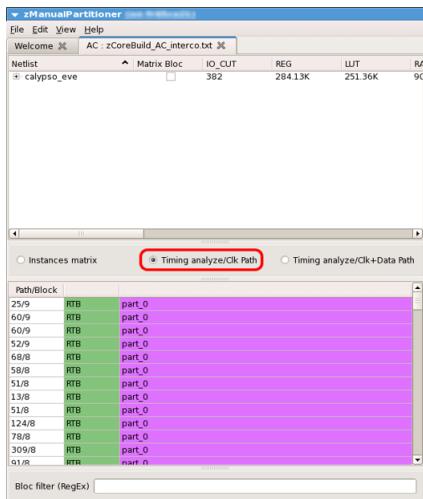
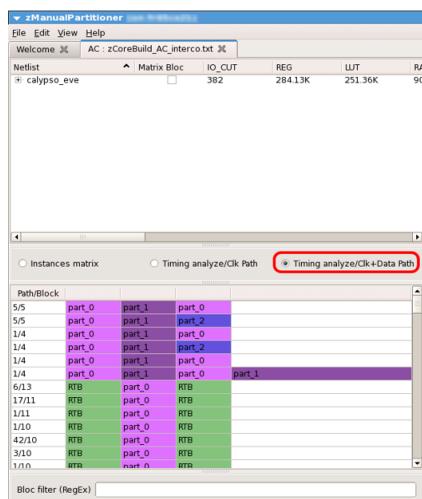


FIGURE 52. Timing Analysis/Clock Path View

3. The **Timing Analyze/Clock+Data Path view** is another view available that displays combinational paths for the data and clock, see the following figure:



The following table displays options available in the **Timing Analysis** pane.

TABLE 32 Available Options in the Timing Analysis Pane

Option	Description
Instances Matrix	Reverts to the Instance Interconnection Matrix pane.
Clock Path View	Displays the combinational path for the clock.
Clock + Data Path View	Displays the combinational path for the data and clock.
Path/Block	<p>The Path is the number of paths represented by the set of path elements.</p> <p>The Block is the number of blocks in the path.</p>

For more information about optimizing timing, see [Performing Timing Analysis](#).

13.1.9 Configuration Window

The **zManualPartitioner** GUI also allows you to select a sample configuration file as a target to map a design with your customized parameters.

From the menu bar, click **Edit > Configuration** to open the **Configuration** window as follows:

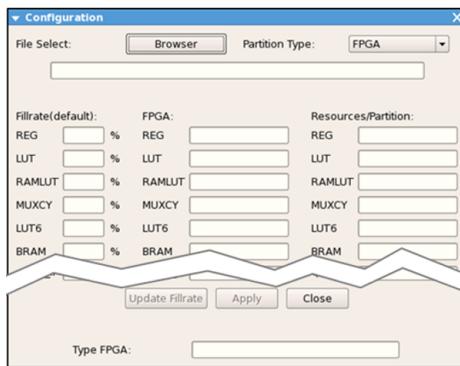


FIGURE 53. Configuration Window

13.1.9.1 Loading a Sample Configuration File

From the **Configuration** window, you can select a specific configuration file from the `$ZEBU_ROOT/etc/configurations` directory.

To select a target, click **Browser** and select your configuration file. The following figure displays the **Configuration** window loaded for an example configuration file.

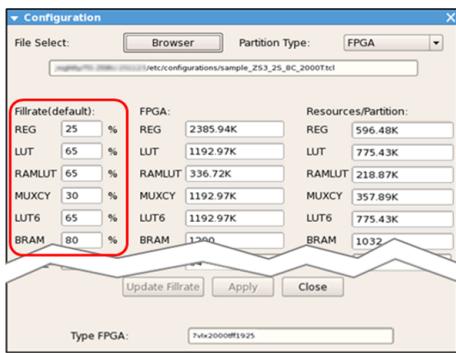


FIGURE 54. Example of Sample Configuration File Loaded

Once a sample configuration file is loaded, you can perform the following actions:

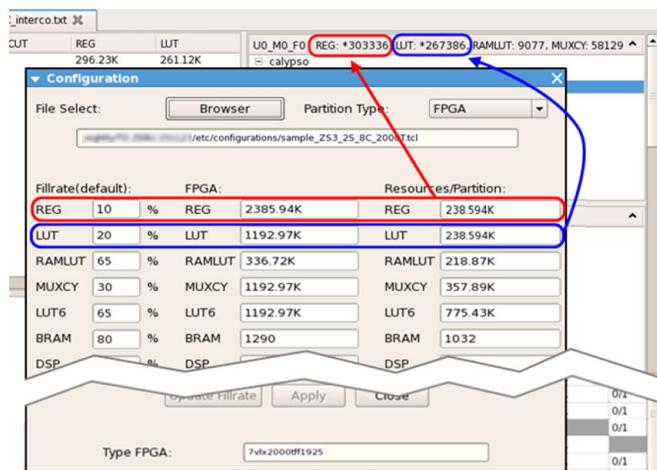
- View various parameters at once.
- Modify filling rates for each resource type.
- See the immediate impact on FPGAs and partitions.

13.1.9.2 Modifying Filling Rates

To modify filling rates in the Configuration window (see [Figure 55](#)), perform the following steps:

1. Modify the resource percentage as required in the **Fillrate (default)** column.
2. Click **Update Fillrate** to modify the values in FPGA and **Resources/Partition** columns.
3. Click **Apply** to validate your parameters.

User Interface

**FIGURE 55.** Modifying Filling Rates in Configuration Window

NOTE: If your set of parameters are not sufficient to map your design, an asterisk is displayed next to the impacted resource, as highlighted in [Figure 55](#) .

13.1.10 Properties Window

The **Properties** window allows you to set various parameters of your **zManualPartitioner** environment.

To open the **Properties** window, click **Edit > Preferences mapping** .

The following table displays available options in the **Properties** window.

TABLE 33 Available Options in the Properties Window

Option	Description
Number of partitions	Specifies the number of partitions to display in the Partition pane.

TABLE 33 Available Options in the Properties Window

Option	Description
Generate log file	Defines the path for the log file containing all user actions.
Information	Defines the columns to display/hide in the Netlist Tree pane.

13.1.11 Loading and Generating Mapping Files

This section consists of the following sub-sections:

- [*Generating a Mapping File*](#)
- [*Loading a Mapping File*](#)

13.1.11.1 Generating a Mapping File

To retrieve partitioning results into a mapping file, perform the following steps:

1. Click **Edit > Generate mapping**.
2. Type a name for the mapping file and click **Save**.

The generation of the mapping file can also be performed with a Tcl command.

13.1.11.2 Loading a Mapping File

You can load a previously saved mapping file into **zManualPartitioner**. To do so, perform the following steps:

1. Click **Edit > Load mapping**.
2. Browse to your mapping file and click **Open**.

Note

You can load a mapping file only with an Automatic Clustering Interconnection file.

13.2 Managing Black-boxes

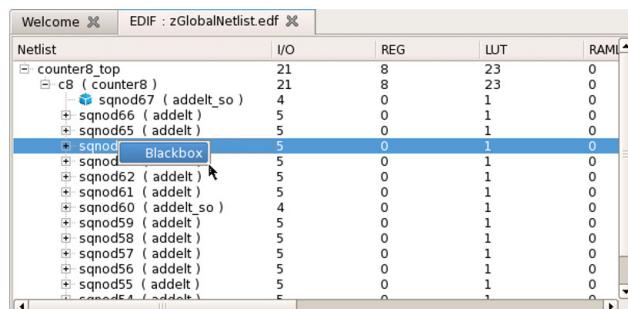
In **zManualPartitioner**, you can manage black boxes in the following ways:

- View black boxes originally defined in the netlist (viewed as )
- Add or remove the black box attribute to any item in the netlist (viewed as )

NOTE: You can manage black boxes only with EDIF files.

13.2.1 Applying a Black Box Attribute

To apply a black box attribute on an item in the netlist, right-click on it and select **Blackbox**.



The screenshot shows the zManualPartitioner software interface with the 'EDIF : zGlobalNetlist.edf' tab active. A table titled 'Netlist' displays various components and their resource usage. One row, specifically 'sqnod54 (Blackbox)', is highlighted with a blue selection bar. The columns are labeled 'I/O', 'REG', 'LUT', and 'RAM'. The 'Blackbox' entry is clearly visible in the 'I/O' column of the selected row.

Netlist	I/O	REG	LUT	RAM
counter8_top	21	8	23	0
c8 (counter8)	21	8	23	0
sqnod67 (addelt_so)	4	0	1	0
sqnod66 (addelt)	5	0	1	0
sqnod65 (addelt)	5	0	1	0
sqnod54 (Blackbox)	5	0	1	0
sqnod53 (addelt)	5	0	1	0
sqnod62 (addelt)	5	0	1	0
sqnod61 (addelt)	5	0	1	0
sqnod60 (addelt_so)	4	0	1	0
sqnod59 (addelt)	5	0	1	0
sqnod58 (addelt)	5	0	1	0
sqnod57 (addelt)	5	0	1	0
sqnod56 (addelt)	5	0	1	0
sqnod55 (addelt)	5	0	1	0
sqnode4 (addelt)	5	0	1	0

FIGURE 56. Applying a Black Box Condition

13.2.1.1 Removing Black Boxing Attribute

To remove a black box attribute and revert an item to its original state, right-click the black box attribute and click **UnBlackbox**.

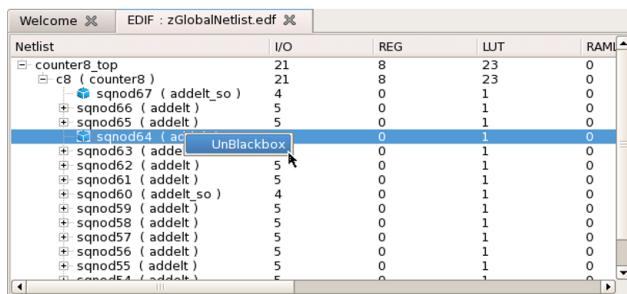


FIGURE 57. Removing a Black Box Condition

13.3 Moving Instances Between Partitions

When you load a netlist file in **zManualPartitioner**, both the **Netlist Tree** pane and the **Partition** pane display the file. By default, the netlist is loaded in Partition 0. However, you can drag and drop instances from one partition to another in the **Partition** pane.

This section consists of the following sub-sections:

- [Using an EDIF File](#)
- [Using an Automatic Clustering Interconnection File](#)
- [Performing Timing Analysis](#)

13.3.1 Using an EDIF File

To move an instance to another partition when you load an EDIF file, perform the following steps:

1. In the **Partition** pane, browse to your netlist and select the instance you want to move.
2. Drag this instance into another partition (Partition 1, see [Figure 58](#)).

Moving Instances Between Partitions

The information about **Impact Analysis** and **Number of Moved LUTs** is dynamically displayed in a tooltip.

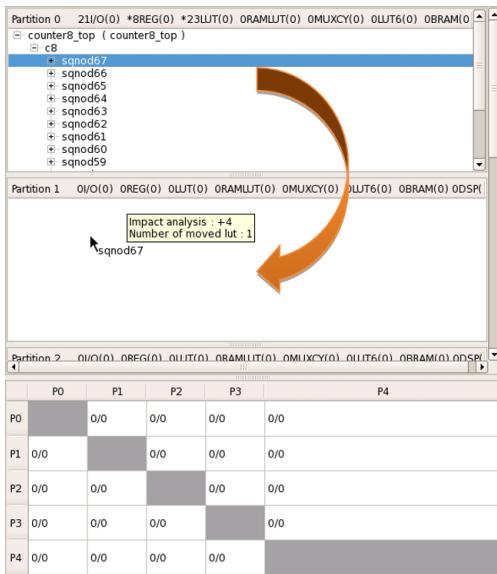
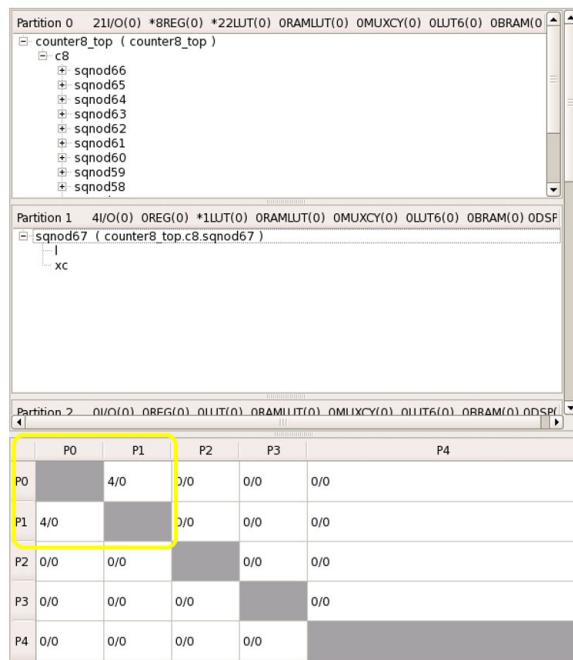


FIGURE 58. Moving Instances in the Partition Pane Using an EDIF File

3. Drop the instance into the new partition (Partition 1).

The result of moving the instance is displayed in the **Partition Interconnection Matrix** window (see [Figure 61](#)).

**FIGURE 59.** Result of Moving Instances in the Partition Pane Using an EDIF File

For more information on the **Partition Interconnection Matrix**, see [Partition Interconnection Matrix Pane](#).

13.3.2 Using an Automatic Clustering Interconnection File

The process for moving an instance to another partition using an *Automatic Clustering Interconnection* file is same as that for an EDIF file, see [Using an EDIF File](#).

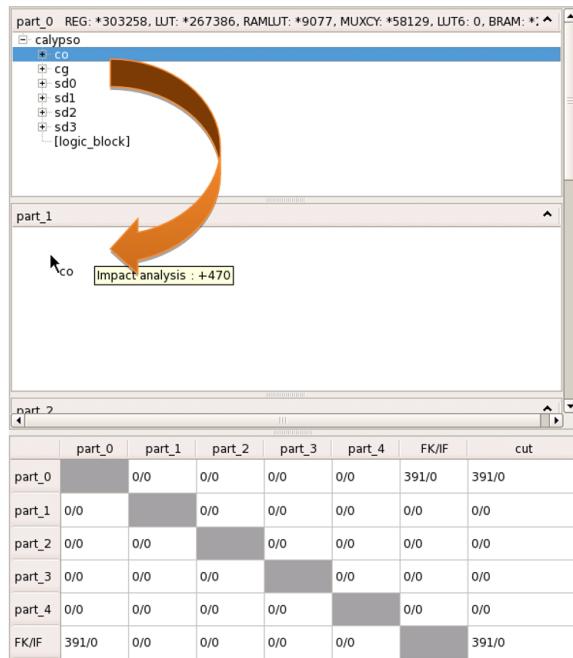
Moving Instances Between Partitions

FIGURE 60. Moving Instances in the Partition Pane Using an Automatic Clustering Interconnection File

The following figure displays result of moving instances in the **Partition** pane using an *Automatic Clustering Interconnection* file.

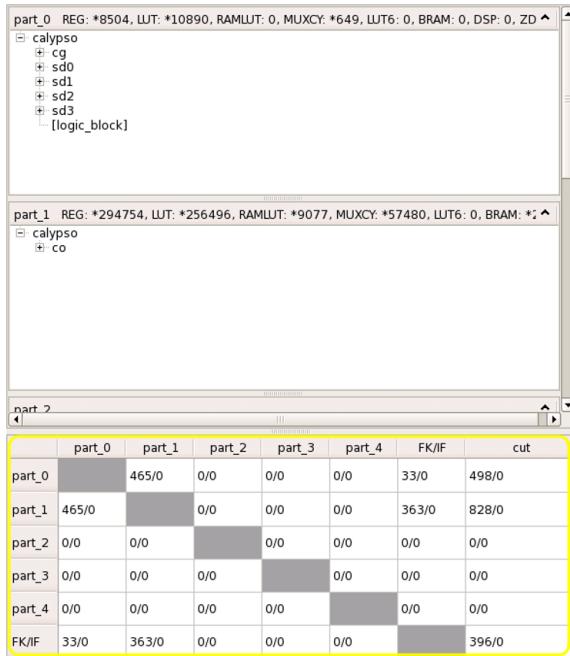


FIGURE 61. Result of Moving Instances in the Partition Pane Using an Automatic Clustering Interconnection File

13.3.3 Performing Timing Analysis

After moving instances across partitions (see [Using an Automatic Clustering Interconnection File](#)), you can modify elements of the combinational paths (that is, a set of successive instances that are in the same partition) and their timing to enhance performance.

Note

The timing analysis feature is only available when you use an Automatic Clustering Interconnection file in zManualPartitioner.

Moving Instances Between Partitions

To modify the content of a combinational path, perform the following steps:

1. Click **Edit > Timing Analyze** to replace the **Instance Interconnection Matrix** pane with the **Timing Analysis** pane.
2. Select your view by selecting **Timing analyze/Clk Path** or **Timing analyze/Clk + Data Path**.
3. Double-click on the path element you want to move to display the **Element Information** window (see *Figure 62*).

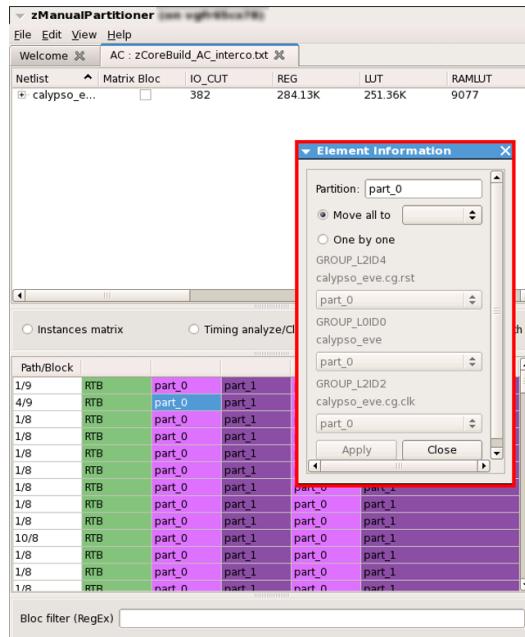


FIGURE 62. Element Information Window

4. In the **Element Information** window, select the method to move by selecting one the following options:
 - Move all to:** Moves the entire combinational path content of a partition to another partition. Selects the destination partition from the drop-down next to it.
 - One by one:** Moves an individual element of a partition to another partition. Selects the destination partition from the drop-down below it.

5. Click **Apply** and then **Close**. The modified element is now in red. It means that it is required to launch timing analysis again (see [Figure 63](#)).

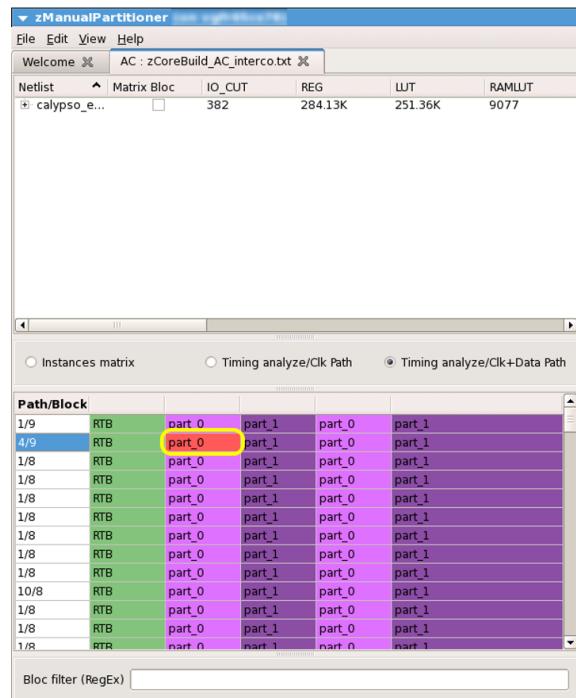


FIGURE 63. Element Modified in the Timing Analysis Pane

6. Select **Edit > Timing Analyze** to refresh the **Instance Interconnection Matrix** pane according to your modifications (see [Figure 64](#)).

Limitations

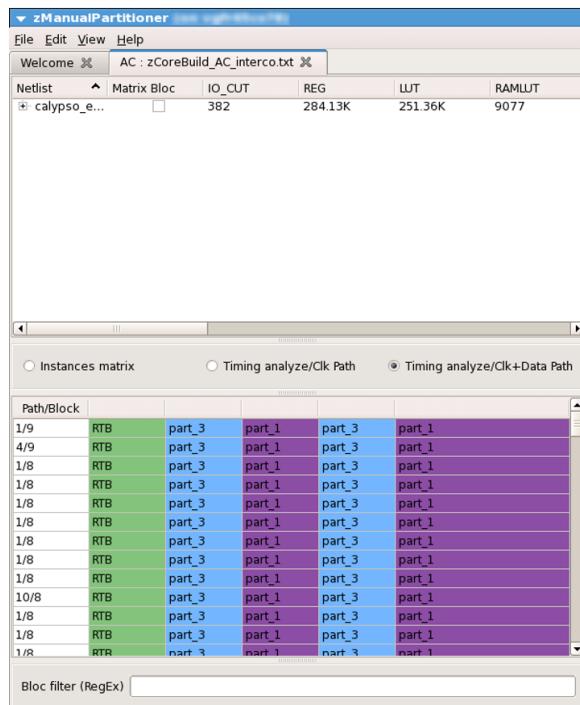


FIGURE 64. Timing Analysis Pane Refreshed

13.4 Limitations

The limitation of **zManualPartitioner** is as follows:

- **zManualPartitioner** is not supported by other partitioners.
- It may be challenging to partition large EDIF files using **zManualPartitioner**.

14 Runtime Performance Analysis With zTune

zTune is an emulation performance analysis tool. When the design is compiled with **zTune** profiling enabled, it generates hardware and software profiles for the design which can be viewed simultaneously using a common web browser. It also helps users to identify emulation performance bottlenecks.

The following are the main features of **zTune**:

- Profiles clock frequencies
- Profiles clock stopping activity
- Profiles software activity
- Interactive GUI for profiling data analysis

Profiling does not have a significant effect on the overall system performance.

To generate **zTune** profiling data, the design must be compiled with the “profile -xtors true” option in the UTF file. The user must also add a few **zTune** API routines (see [zTune Runtime](#)) to their code to configure, and activate the profiling during runtime.

When performance profiling is enabled, a directory is generated during emulation to save the profiling data. This directory is local to the first process that enables the performance monitoring.

After emulation, the user can pass this directory as an argument to **zTune** for post processing and analyze the emulation performance using the web browser of their choice to access the **zTune** GUI (see [Post Processing](#)).

This chapter discusses the following topics:

- [Data Collected](#)
- [Commands/ API Description](#)
- [Compile Time Options](#)
- [zTune Runtime](#)
- [Post Processing](#)
- [Limitations](#)

14.1 Data Collected

There are many ways to measure emulation performance. Most commonly, the performance is measured by comparing the progress of simulated real time to wall clock time. The simulated real time depends on the DUT controlled clocks and frequencies. This chapter describes the types of data collected by **zTune**.

14.1.1 Controlled Clocks

Emulation performance is determined by clock frequencies in the design. During emulation, some components of the design can stop the clocks (Controlled Clocks) and lower the average clock frequency.

Controlled clocks can be stopped at runtime for multiple reasons. Transactors, including co-simulation transactors such as C_COSIM, may stop controlled clocks through a clock controller instance. Controlled clocks may also be stopped by ZeBu features such as Power Management (POMA) or turned on. **zTune** collects data to help you identify the component responsible for stopping the controlled clocks.

14.1.2 Clock Frequency

The average frequency and instantaneous frequency of the DUT's controlled clocks are important parameters to analyze runtime performance. Instantaneous frequency is the clock frequency measured in a small time interval, for example 10 ms. This time interval is called the sampling rate. The following diagram displays average and instantaneous frequencies collected by the profiler:

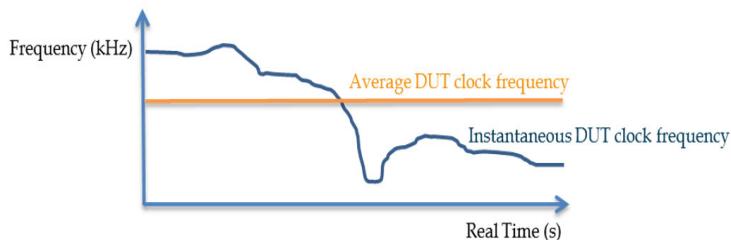


FIGURE 65. Frequency Monitoring

The instantaneous frequency helps identify the time window when a controlled clock

Commands/ API Description

slowdown occurs.

14.1.3 Clock Stopping Activity

Clock stopping slows down the emulation time and results in lower clock frequencies. Hence, it is important to determine the reason for the frequency slowdown. Clock stopping analysis determines the clock controller that stopped the clocks and the moment when the clocks stopped.

This requires obtaining the instantaneous stop status of each clock controller. The instantaneous stop status is defined as the number of driver clock cycles that the `readyForClock` signal is 0 during a small time interval. The following graph helps identify the origin of the clock stopping requests.

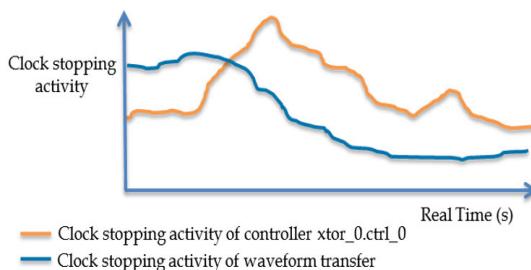


FIGURE 66. Monitoring Clock Stopping Activity

14.2 Commands/ API Description

This section describes the API commands supported by **zTune**.

Note

zTune is supported only in UC 1.0 compilation flow.

14.3 Compile Time Options

To enable/disable **zTune** transactional profiler, add the following command in the UTF file while compiling the design:

```
profile * -xtors true -xtors_params  
{TYPE=ZEMI_GLOBAL|ZEMI_DPI|ZEMI_GLOBAL_DPI|LEGACY}
```

where,

- ZEMI_GLOBAL: Profile global (sampling/internal/dpi/b2b) clock stopping for ZEMI transactors
- ZEMI_DPI: Profile per DPI clock stopping for ZEMI transactors
- ZEMI_GLOBAL_DPI: Profile both global and per DPI clock stopping for ZEMI transactors
- LEGACY: Legacy profiling (per transactor zceiClockControl)

By default, the `profile` command enables profiling for:

- SW stack
- ZCEI controlled clocks
- ZCEI clock control macros

14.4 zTune Runtime

zTune is supported with following zTune API and runtime environments:

- C++ API
- **zEmiRun**
- **zRci**

14.4.1 zTune C++ API

To activate the profiling, the users must add a few lines of code to the test code to initiate and activate the storage of profiling data.

zTune Runtime

zTune can be used with the following zTune C++ runtime API (zTune.hh header):

```
Ztune::Configuration cfg;
Ztune::Init(cfg);
Ztune::Start("<some_dir>");

Board* zBoard = ZEBU::Board::open("zcui.work/zebu.work");

Ztune::Stop();
```

where, configuration is defined in Types.h as follows:

```
/*
 * \brief Configuration for Ztune::Start.
 */
struct ZEBU_Ztune_Configuration
{
    const char *outputPath; // The path of database
    int clockStopEnable; // Enable global clock stop handling
    int debug; // Debug verbose mode
    int dumpLegacyCsv; // Write legacy CSV format with only HW info
    int hwEnable; // Enable HW profiler
    int hwPollDelay_us; // HW monitor thread poll delay in us
```

```
int portTrackingEnable; // Enable transactor thread tracking  
int rtEnable; // Enable real time profiler. If set swEnable is  
ignored.  
int swEnable; // Enable CPU time based profiler  
int swSamplingPeriod_ms; // SW profiler sampling period  
...  
};
```

To stop profiling (before `Board::close()`), use the following command:

```
ztune::Stop();
```

14.4.2 zEmiRun

The following command line option must be used to enable profiling with **zEmiRun**:

```
-u|-ztune [<dump path>] : Enable zTune profiling feature.
```

To start zTune in **zEmiRun**, use the following command:

14.4.3 zRci

```
zEmiRun -ztune <some_dir>
```

To start zTune in **zRci**, use the following command:

```
start_zebu -ztune <some_dir>
```

The following options can be used to enable zTune operations:

```
ztune      # zTune operations.
           ztune -start [path]
           ztune -stop
           ztune -init
           ztune -create_event_marker [name]
           ztune -config <option> [value]
           Options:
                   output_path <path>
                   csv [on|off|1|0]
                   hw_profiler [on|off|1|0]
                   hw_poll_delay [delay in us]
                   port_tracking [on|off|1|0]
                   real_time_profiler [on|off|1|0]
                   cpu_profiler [on|off|1|0]
                   sw_sampling_period [period in ms]
                   system_marker_auto_creation [on|off|1|0]
                   user_markers_max [max]
```

Where,

- [-start [<path>]] : Starts data recording
- [-stop] : Stops data recording
- [-init] : Initializes zTune. It must be called before start_zebu
- [-create_event_marker [<name>]] : Creates an event marker
- [-config <option> [<value>]] : Fetches or sets the **zTune** configuration option

14.5 Post Processing

zTune is available in `$VCS_HOME/bin`. You must use the compatible version of VCS used for compilation and emulation.

```
% $VCS_HOME/bin/zTune <path to profiling database>
```

The profiling data is saved in the directory specified by the `SetOutputPath()` method. The contents of this folder are analyzed by **zTune**. It displays profiling data graphically through a GUI or in a command line mode.

The following figure displays on data analysis performed by zTune.

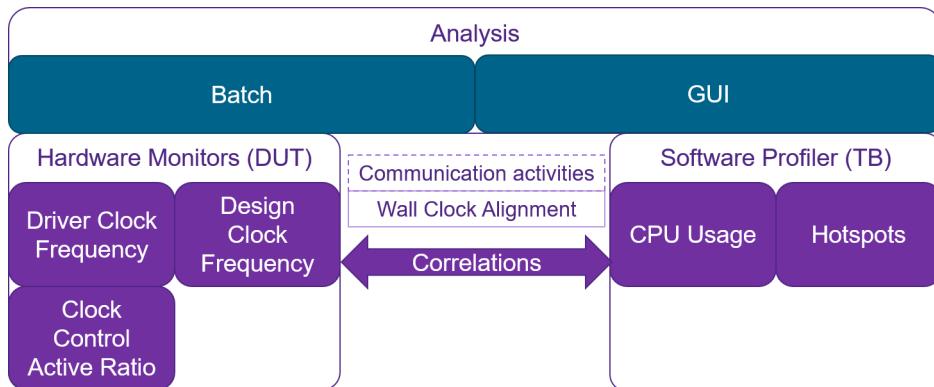


FIGURE 67. Data Analysis in zTune

14.5.1 zTune GUI

The **zTune** post-processing tool requires a modern web browser and Python 2.7 (provided in the ZeBu release) to display the results in the GUI.

A web browser is released at `$VCS_HOME/bin/firefox` (version 44.0.2). Only this version of the Firefox is supported.

14.5.1.1 Launching zTune GUI

To launch **zTune** after emulation runtime, perform the following steps:

1. Start a web server using the following **zTune** script:

```
$ zTune <some_dir>/profile_dir
```

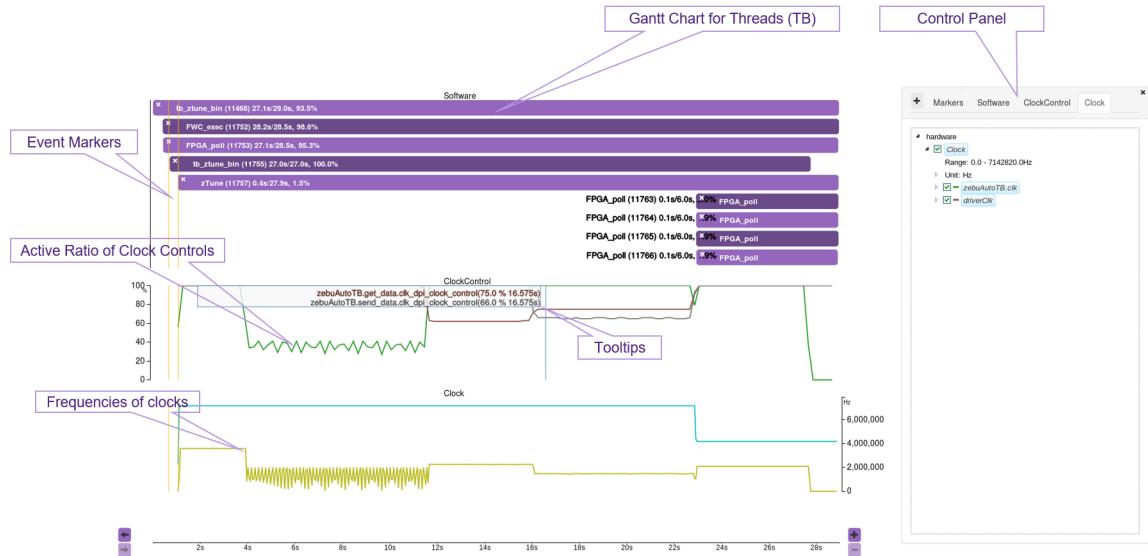
The **zTune** script creates a web server on the machine where the script was running and displays the address on the Unix terminal from where **zTune** was launched from:

```
Loading profile database...
Profile database is load successfully.
Listening on [http://vginthw141:1106]
Web server activities will be logged in collect_web.log
```

2. To access data, open a web browser in Windows or Linux and enter this web server address to launch the **zTune** GUI.

<http://vginthw141:1106>

The **zTune** GUI home page is displayed with the information from the profile directory.

**FIGURE 68.** zTune Initial GUI

The GUI x-axis represents the real emulation time and the bars represent process threads:

- Showing the thread ID number received when the threads were created on the host system.

You can see the clock panel beneath the threads panel. The clock panel shows the graph of the clock signals in the system. In addition, a white graphic representation of the clock-control signals (from the transactors) that had any impact on the emulation is shown beneath the clock control panel.

Beside the graphic representation, you can see a control panel with the following tabs:

- **Software**
- **Clock**
- **ClockControl**
- **Marker**

Each tab controls the content of the respective panels associated with it.

The visible threads, clocks and clock controls can be selected from the list of each

Post Processing

panel.

Note

You can super-impose signals from threads, clocks, and clock controls on each selected panel.

14.5.1.2 GUI Views

The **zTune** GUI presents the following levels of software profile views:

- Overall View
- Thread View
- Function View

Overall View

The initial **zTune** GUI view is also called Overall view. It is a graph with real emulation time on the x-axis. The horizontal bars represent the time during which processes and threads are running.

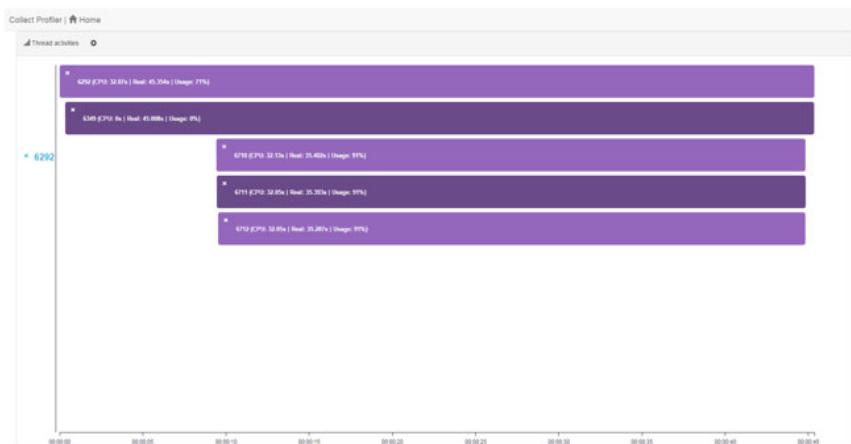


FIGURE 69. Overall View

You can obtain the following information from the overall view:

- **Process ID:** The process ID is displayed on the left side of the view, as displayed in the following figure:



FIGURE 70. Process ID

You can view the command used to launch this process if you hold the pointer over this process ID.

- **Thread Bars:** The thread bars display when the process threads are launched and finished. You can find the following information displayed on the bars in the GUI:

- Thread ID
- CPU Time (in seconds)
- Real Time (in seconds)
- CPU usage (in percentage)

Post Processing



FIGURE 71. Threaded Bars

Thread View

Click a thread in the overall view to open the thread view. The Thread view is a graph with real emulation time on the horizontal axis. It displays a graphical representation of the CPU usage during the lifetime of the selected thread.

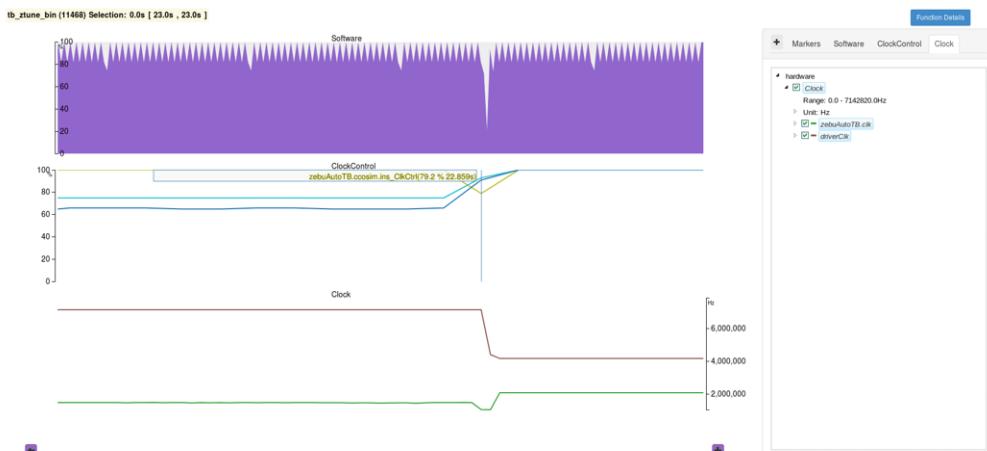


FIGURE 72. Thread View

You can select or deselect any process by clicking any process. If you double-click any process, it opens “Thread View”.

You can select any thread in the design listed in the **Software** tab of the control panel by selecting the check box next to the "thread name/number".

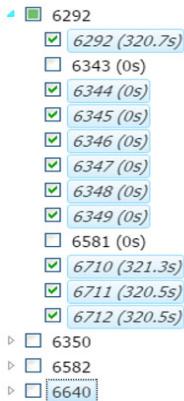
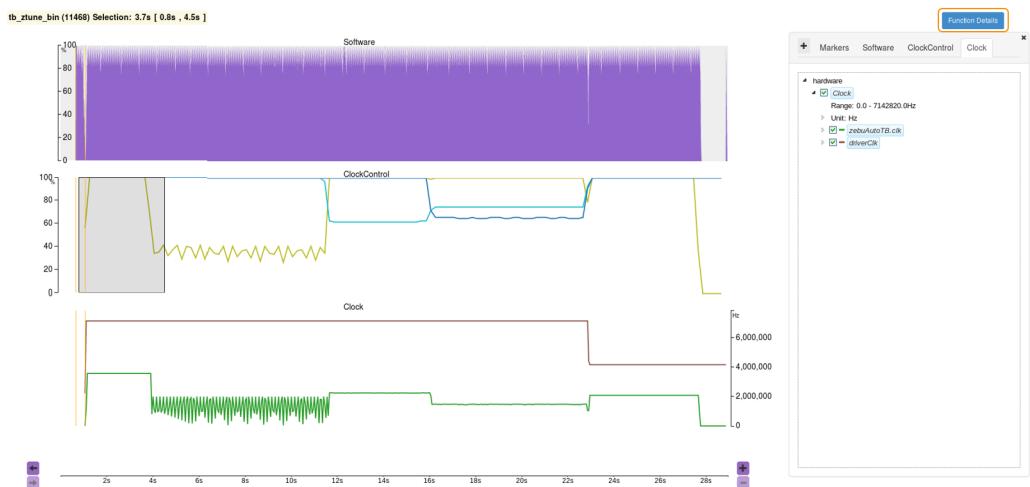


FIGURE 73. Software Profile Configuration Page

Function View (C/C++ testbench)

The function view displays the CPU usage of individual C/C++ functions. You can open the Function View of a thread by selecting a range in the Thread View and then clicking Function Details in the same view. See the following figure:



Post Processing

FIGURE 74. Selecting a Range in Thread View

This opens the Function View for the selected range as displayed in the following figure:

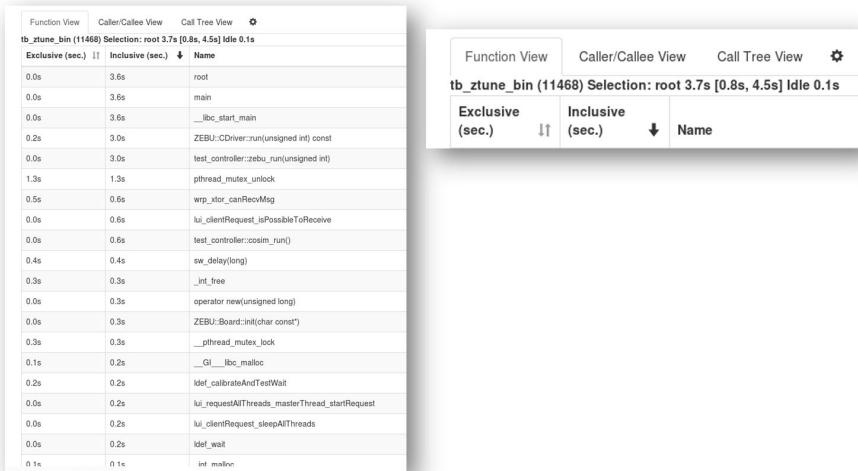


FIGURE 75. Function View

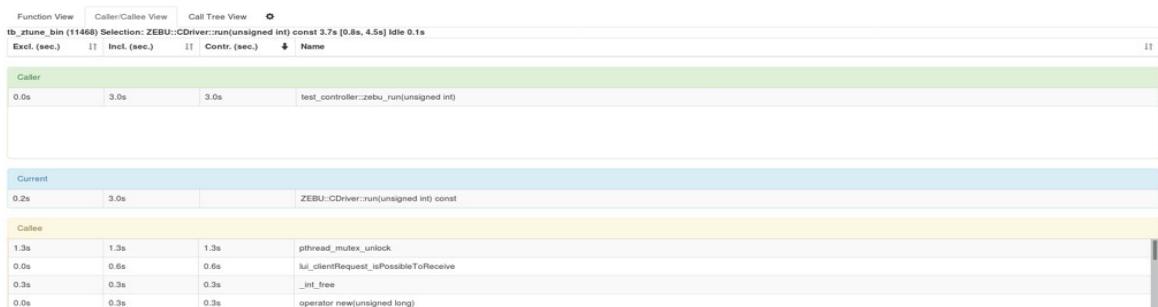
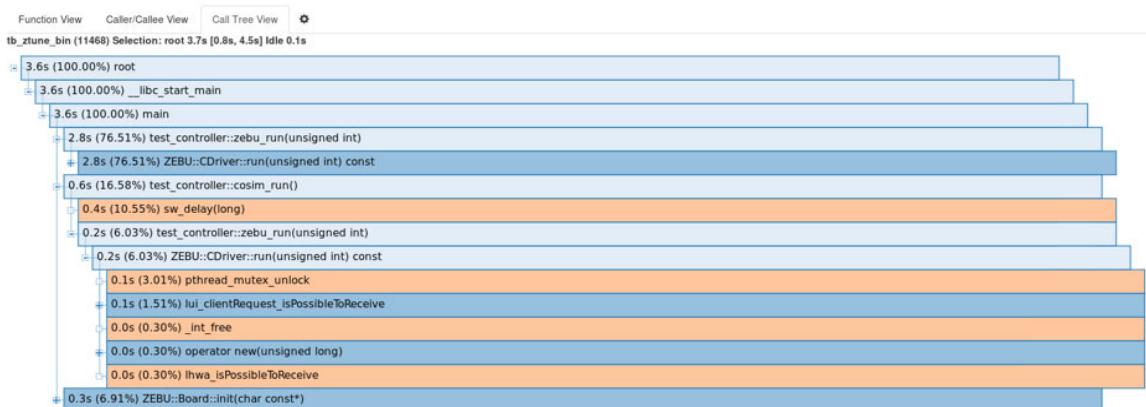
Note

If you do not select any range, the function view displays the entire lifetime of the selected thread.

You can select the range according to the emulation profile data to find out the behavior of any process thread for the time during which performance degrades. The following types of views are available:

- Function View ([Figure 75](#))
- Caller/Callee View
- Call Tree View

You can switch to the required view by clicking the respective view name in the Function View window.

**FIGURE 76.** Caller/Callee View**FIGURE 77.** Call Tree View

Markers

To get the list of commands associated with `report_markers`, use the following:

```
[zTune]$ report_markers --help
usage: report_markers [-h] --name NAME [--interval
BEGIN:END|BEGIN:|:END]
                           [--time-unit {absolute,seconds,cycle-counter}]
```

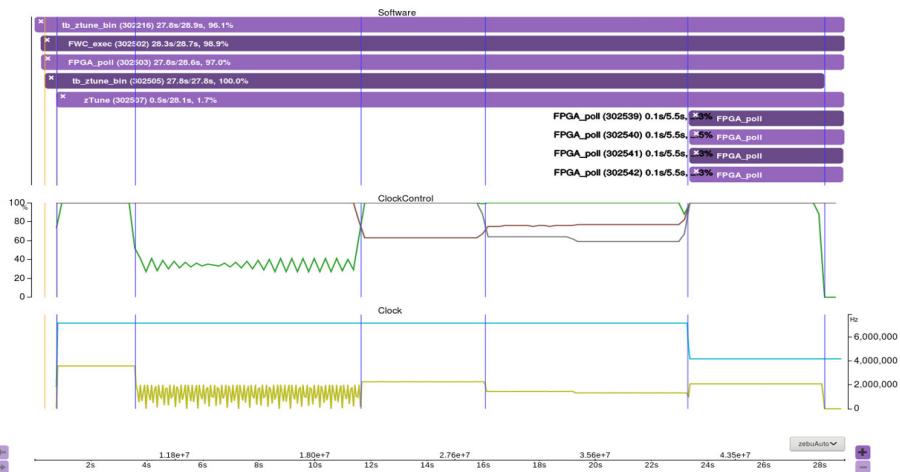
Post Processing

```
[--ref-clk REF_CLK] [--csv [=STDOUT|FILENAME]]  
[--origin {all,System,User}]
```

`report_markers` supports the following optional arguments:

- `-h, --help`: Shows this help message and exits
- `--name NAME, -n NAME`: Displays the marker data name (regexp)
- `--interval BEGIN:END|BEGIN:|:END, -i
BEGIN:END|BEGIN:|:END`
- `--time-unit {absolute,seconds,cycle-counter}, -t
{absolute,seconds,cycle-counter}`: Provides scale of time to display
- `--ref-clk REF_CLK, -r REF_CLK`: Sets reference clock to which to display the cycle count (regexp)
- `--csv [=STDOUT|FILENAME]`: Generates report in the csv format
- `--origin {all,System,User}, -o {all,System,User}`: Enables filter results on origin

In the following zTune GUI, yellow vertical lines represent predefined system markers and blue lines represent user-defined markers.

**FIGURE 78.** Markers View

14.5.2 UPF Event Support in zTune

The UPF event support is added in **zTune** only for ZeBu Server 1, 2, and 3.

Three dedicated UPF signals are collected as follows:

- For UPF power-domain “switching”, see - `upf_pd_change`
- For UPF power-domain “isolation”, see - `upf_isolation`
- For UPF power-domain “retention”, see - `upf_retention`

These signals can be viewed in **zTune**. Each signal is visible only if the related UPF feature is used in the profiled design.

All these signals are visible in the clock control graph.

- ▷ □ = `z_generic_trigger.upf_pd_change`
- ▷ □ = `z_generic_trigger.all_features`
- ▷ □ = `z_generic_trigger.upf_features`
- ▷ □ = `z_generic_trigger.upf_retention`

FIGURE 79. UPF Events

To see consolidated information on a specific type of signal, use the `upf_features` command. The syntax is as follows:

```
upf_features = upf_pd_change | upf_isolation | upf_retention
all_features = upf_features | other features handled by "generic
trigger" transactor
```

Usually **zTune** semantics is used: 0%, clocks are always stopped; 100%, clocks are always running.

14.5.3 System Stoppers Information

In ZeBu Server 1, 2, and 3, three system stoppers (clock control related to the ZeBu infrastructure) are available:

- For FWC stops: `system_stoppers.fwc_stop`
- For transactional memory clock: `system_stoppers.memx_stop`
- For logic analyzer clock stop: `system_stoppers.analyzer_stop`

In ZeBu Server 4, due to a change in clock generation and distribution, only one system stop signal is available for each unit:

- For FWC stops (one per unit with FWC IP): `system_stoppers.fwc_stop.Ui`

For transactional memory stops (`memx`) (one signal per unit in the clock control graph): `system_stoppers.memx_stop.Ui`

14.5.4 zTune in Command Line Mode

zTune can also be launched in command-line mode by specifying a command file or directly using switches (such as, `-proc` or `-func`). In this mode, the profiled data is directly displayed on the terminal output.

For example:

```
$ zTune -proc 0 ztune_dir
```

Lists all the processes monitored by **zTune** sorted by their CPU time.

14.5.4.1 Supported zTune Commands

This section lists the supported commands in **zTune**.

Printing Count Functions

To print count functions sorted by "incl"/"excl" execution time, use the following command:

```
func "incl"/"excl" tid/pid count
```

where,

- "incl" is the execution time of a function, including the time consumed by functions called directly or indirectly.
- "excl" is the time consumed by a function, excluding the time consumed by the functions called directly or indirectly.
- tid/pid should be a valid process thread ID. If 0 or an invalid ID is given, the whole thread/process data is displayed.
- count should be a positive integer number – the number of functions.
- proc count

The proc count command prints information about count threads or processes sorted by the execution time. The count should be non-negative integer; if 0 is given, the whole thread or process data is displayed.

- hw meta

The hw meta command dumps hardware metadata (reference frequency, sample period, monitored hardware components, and its indexes).

- hw index

The hw index command dumps sample information for hardware component index. It retrieves the index from the hw meta command.

Finding Functions

To find the functions, use the following command:

```
$ find_functions -h
```

Use Model

```
find_functions [-h] --name NAME [--limit LIMIT]
                [--time-unit {absolute,seconds,cycle-counter}]
                [--ref-clk REF_CLK]
                [--sort {inclusive,exclusive}]
                [--csv [=STDOUT|FILENAME]]
                [--interval BEGIN:END|BEGIN:|:END]
```

Optional arguments that can be used with `find_functions` follows:

- `-h, --help`: Shows this help message and exits
- `--name NAME, -n NAME`: Function name (regexp)
- `--limit LIMIT, -l LIMIT`: Maximum number of threads to display, when limit is 0, display all threads
- `--time-unit {absolute,seconds,cycle-counter}, -t {absolute,seconds,cycle-counter}`: Time scale to display
- `--ref-clk REF_CLK, -r REF_CLK`: Reference clock to which to display the cycle count (regexp)
- `--sort {inclusive,exclusive}, -s {inclusive,exclusive}`: Sorts elements and accepts the minimum valid string
- `--csv [=STDOUT|FILENAME]`: Report in csv format
- `--interval BEGIN:END|BEGIN:|:END, -i BEGIN:END|BEGIN:|:END`

Example

```
$ find_functions -n zebu --csv
```

This command displays Function Name, Thread ID, Thread Name, Time Range, Inclusive Time, Exclusive Time, Callers.

```
test_controller::zebu_run(unsigned int),2367,tb_ztune_bin,10.06s--  
46.96s,22.99s,0.0s,"test_controller::cosim_run(),main"
```

14.5.4.2 Hardware Description Commands

report_hardware Command

Usage:

```
report_hardware [-h] --name NAME [--category {Clock  
Control,Clock}]  
                [--time-unit {absolute,seconds,cycle-  
counter}]  
                [--value-unit {Hz,kHz,%}] [--ref-clk REF_CLK]  
                [--precision TIME(s|us|ms) | POINTS]  
                [--csv [=STDOUT|FILENAME]]  
                [--interval BEGIN:END|BEGIN:|:END]
```

The following table lists the arguments that can be used with `report_hardware`:

TABLE 34 Arguments Used with `report_hardware`

Arguments	Description
<code>-h, --help</code>	Shows the help message and exit
<code>--name NAME, -n NAME</code>	Hardware data name (regexp) (mandatory).
<code>--category {Clock Control,Clock}, -c {Clock Control,Clock}</code>	Category name
<code>--time-unit {absolute,seconds,cycle- counter}, -t {absolute,seconds,cycle- counter}</code>	Scale of time to display

TABLE 34 Arguments Used with report.hardware

Arguments	Description
--value-unit {Hz,kHz,%}, -v {Hz,kHz,%}	
--ref-clk REF_CLK, -r REF_CLK (regexp)	Reference clock for which the cycle count is displayed
--precision TIME(s us ms) POINTS, -p TIME(s us ms) POINTS	Time between each sample
--csv [=STDOUT FILENAME]	Report in csv format
--interval BEGIN:END BEGIN: :END, -i BEGIN:END BEGIN: :END	Display data only on specified interval. For example: <ul style="list-style-type: none"> • -i 10:20 between 10 and 20 time units (as per -t) • -i 10: between 10 time units and end of data • -i :30 between beginning of data and 30 time units

Output:

```
$ report.hardware --name=driverClk --csv
Time[s],driverClk[Hz]
0,0
0.2,1000
0.4,10000
0.6,10000
...
$ report.hardware --name=ccosim*
Time[s]    hw_top.ccosim_clk[Hz]
hw_top.ccosimDriver.readyForCclock[Hz]
0          0                      0
1          1000                  30
2          3000                  100
```

14.5.4.3 Help Command (zTune -h)

Usage

```
zTune [-h] [--port PORT] [--host HOST] [--process count]
      [--function incl/excl tid count] [--command_file
COMMAND_FILE]
      [--log LOG] [--interactive]
      [--filter_clockControl_percentage
FILTER_CLOCKCONTROL_PERCENTAGE]
      [--filter_clockControl_number
FILTER_CLOCKCONTROL_NUMBER]
      [--cycles_scale]
      database
```

Positional argument, database, can be used with zTune -h to point to the profile database.

The following table lists the optional arguments that can be used with zTune -h.

TABLE 35 Optional Arguments Used with zTune -h

Optional Arguments	Description
-h, --help	Displays the help message and exit
--port PORT, -p PORT the port listens on [Default: 1106]	
--host HOST, -host HOST	The host listens on [Default: vg-vm-vnc15]
--process count, -proc count	Lists count process and threads sorted by CPU time
--function incl/excl tid count, -func incl/excl tid count	Lists functions sorted by incl/excl time in thread/process
--command_file COMMAND_FILE, -cmds COMMAND_FILE	Executes commands in the input command file

TABLE 35 Optional Arguments Used with zTune -h

Optional Arguments	Description
--log LOG, -l LOG	Redirects output into file
--interactive, -i	Interactive mode
--filter_clockControl_percentage FILTER_CLOCKCONTROL_PERCENTAGE, -fclkctrp FILTER_CLOCKCONTROL_PERCENTAGE	Displays clock controls that are lesser than [Default:95]
--filter_clockControl_number FILTER_CLOCKCONTROL_NUMBER, -fclkctrn FILTER_CLOCKCONTROL_NUMBER	Displays clock controls that are not greater than [Default 20]
--cycles_scale, -cyc	Displays clock cycles

14.5.5 CSV Output

An alternative output format is available in Comma Separated Values (CSV) that can be used to display clock frequency and clock stopping data (software profiling is not available in this format). The CSV formation captures ONLY the HW profile.

To enable CSV output, call `Ztune::Start()` with the following configuration (or the C equivalent) :

```
Ztune::Configuration cfg;
cfg.dumpLegacyCsv = true;
```

This generates a directory “ztuneFolder_CSV”. The output folder contains the following CSV files:

- `ztuneClock*.csv`: This file contains the clock frequency data. The first column is current time (in seconds) and remaining columns are the frequencies of the design clocks and driver clock (in Hz).
- `*ztuneData*.csv`: This file contains the clock stopping activity data from transactors (`zceiClockControl` macro) and ZeBu features. There is one such file for each RTB (Interface FPGA is used by the design). The first column is the current time (in seconds) and remaining columns are the percentage of the sampling period during which the clocks controlled by the `zceiClockControl`

macro (or all for ZeBu features) could run. A quantity of 100 percent means the clocks were not stopped (running) during the entire sampling period.

14.6 Limitations

Note

The firmware level logic analyzer is no longer available in ZeBu Server 4 and thus no system_stoppers.analyzer_stop signal is seen.

zTune has the following limitations:

- Automated analysis of results is not supported.
- Hardware and software message exchange activity is not collected.
- Sampling rate cannot be configured.

15 Saving the Design State and Restarting from a Saved State

For fast and effective emulation, typically to skip the OS boot, it is possible to save the design state at a given time, and then restart multiple times from this saved state.

ZeBu offers several methods to perform this:

- Save and Restore saves the state of the DUT. It is the responsibility of the user to save the state of the software testbench, if necessary. This method may be used when the testbench is simple and it is easy for users to define a point from which the software testbench must be restarted.
- Saving the design state is user-controlled in the testbench (**zRci** or C++). It saves the state of FPGAs and the content of design memories.
- Restoring in a later emulation goes directly to the design state as of time when the design was saved.
- Save and Restore must be performed on the same hardware platform type with the same configuration.

This chapter discusses the following topics.

- *Save and Restore - Recommendations*
- *C++ Method for Save and Restore*
- *C++ Example*
- *C Function for Save and Restore*

15.1 Save and Restore - Recommendations

- Only the state of the ZeBu hardware is saved.
 - Restore is done by a separate script.
- Before saving the state of the hardware,
 - Flush and close waveform files (if any), and
 - Flush and close DPI files (if any)

15.2 C++ Method for Save and Restore

To save the hardware state, use the `FastHardwareState` class as follows:

```
FastHardwareState::FastHardwareState();
```

This method is the constructor of the `FastHardwareState` object.

The `FastHardwareState` class provides the following methods:

- To initialize the `FastHardwareState` object, use the following method:

```
void FastHardwareState::initialize(Board *board)
```

where,

- `board` is a pointer to the current `Board` object.
- To initialize and filter the `FastHardwareState` object, use the following method:

```
void FastHardwareState::initialize(Board *board, const Filter  
*filter)
```

C++ Method for Save and Restore

where,

- board is a pointer to the current Board object.
- filter is a pointer to a Filter object and allows to filter the following types of components to save:
 - ◆ internal signals
 - ◆ driver signals
 - ◆ internal
 - ◆ external memories
 - ◆ clocks

- To capture the hardware state into memory, use the following method:

```
void FastHardwareState::capture()
```

- To write the hardware state previously captured on the disk, use the following method:

```
void FastHardwareState::save(const char *filename, bool inParallel
= false)
```

Where,

- filename is the name of the directory in which the hardware state must be saved.
- inParallel specifies if the hardware state must be saved using parallel tasks.

Note

The FastHardwareState::save method cannot be called after calling Board::close.

The FastHardwareState::save method does not release the memory allocated by the call to the capture method.

The following C++ method is available for save and restore:

```
static Board *Board::restoreHardwareState  
    const char *saveDirectory,  
    const char *zebuWorkPath = "./zebu.work",  
    const char *designFile = 0,  
    const char *processName = "default_process")  
    throw(std::exception);  
  
static Board *Board::restoreHardwareStateWithSelectionFile  
    const char *saveDirectory,  
    const char *zebuWorkPath = "./zebu.work",  
    const char *selectionDBPath = 0,  
    const char *designFile = 0,  
    const char *processName = "default_process")  
    throw(std::exception);
```

where,

- `saveDirectory` is the name of the file from which the state of the ZeBu hardware is to be restored.
- `zebuWorkPath` (default `"/zebu.work"`) is the path to the `zebu.work` directory.
- `designFile` (default `"designFeatures"`) is the name of the `designFeatures` file.
- `processName` (default `"default_process"`) is the name identifying this process in the case of a multi-process testbench.
- The `Board::restoreHardwareState()` and `Board::restoreHardwareStateWithSelectionFile()` methods return a pointer to the `Board` object or `NULL` if the open operation failed.

C++ Example

15.3 C++ Example

The following is an example of C++ method for save and restore:

```
FastHardwareState hwState;
initialize( _zebu );
capture();
save( "hw_state" );
clean();
[...]
_zebu = Board::restoreHardwareState( "hw_state",
_zebu_work.c_str() );
```

15.4 C Function for Save and Restore

The following C functions are available for save and restore:

```
ZEBU_Board *ZEBU_restoreHardwareState
    const char *saveDirectory,
    const char *zebuWorkPath,
    const char *designFile,
    const char *processName);

ZEBU_Board *ZEBU_restoreHardwareStateWithSelectionFile
    const char *saveDirectory,
    const char *zebuWorkPath,
    const char* selectionDBPath,
    const char *designFile,
    const char *processName);
```

where,

- `saveDirectory` is the name of the file from which the state of the ZeBu hardware is to be restored.
- `zebuWorkPath` (default "./zebu.work") is the path to the `zebu.work` directory.
- `designFile` (default "designFeatures") is the name of the `designFeatures` file.
- `processName` (default "default_process") is the name identifying this process in the case of a multi-process testbench.
- The `ZEBU_restoreHardwareState()` and `ZEBU_restoreHardwareStateWithSelectionFile()` functions return a pointer to the `Board` structure or `NULL` if the open operation failed.

16 Direct - ICE for ZeBu Server 3

The ZeBu Server 3 Direct-ICE interface connects an emulated DUT to a target system. The features described in this chapter are applicable to a ZeBu Server 3 system with the Direct-ICE interface connected to a 9F/ICE FPGA module.

This section discusses the following topics:

- *Direct-ICE Architecture and Functional Description*
- *Functional Description of Direct-ICE Clocks*
- *Functional Description of Direct-ICE I/O Signals*
- *Physical Description*
- *Writing the Direct-ICE Compilation Script*
- *Compiling for Direct-ICE*
- *Runtime With Direct-ICE*
- *Mechanical Data*

16.1 Direct-ICE Architecture and Functional Description

The Direct-ICE interface is available on the top of the ZeBu Server 3 system from an interconnection board connected to the Direct-ICE module. It provides power-supply, I/O connections, and clock connections to the Direct-ICE target system.

The voltage level on the Direct-ICE interface I/O pins is VCCIO_1 or VCCIO_2. The architecture of the interface is based on two connector banks; each bank is supplied with one of the voltages (VCCIO_1 or VCCIO_2).

The Direct-ICE DDR3 memories for the 9F/ICE FPGA module can be used to map ZRM memory models similar to other memory resources in the ZeBu Server 3. The following figure depicts the architecture of Direct-ICE Interface connected to the 9F/ICE FPGA module in a ZeBu Server 3.

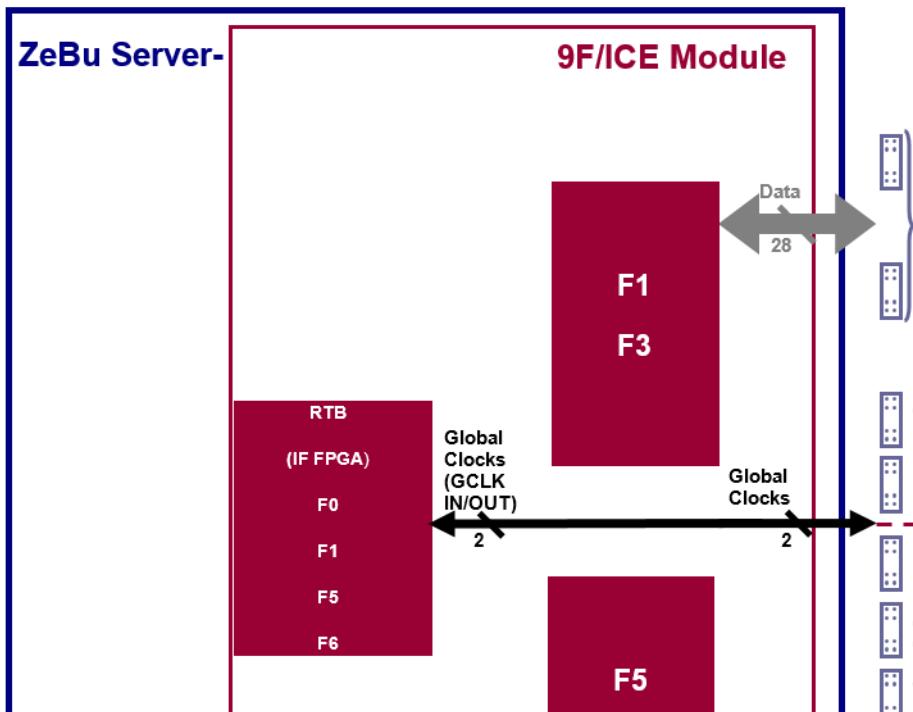


FIGURE 80. Architecture of the Direct-ICE Interface with a 9F/ICE Module

The Direct-ICE interface architecture consists of the following components:

- FPGAs: The 9F/ICE FPGA module includes nine Xilinx Virtex-7 XC7V2000T FPGAs:
 - Nine DUT FPGAs
 - Four interface FPGAs (IF)
- Memories:
 - F0, F1, F6, and F7 have 2 Gigabits DDR3 each
 - F8 has 2x 8 Gigabytes DDR3
- Connections: There are 284 I/O pins allocated as follows:
 - F1, F3, F5 and F7 are standard DUT FPGAs connected to the Direct-ICE interface with 190 I/O pins for F1 and 190 I/O pins for F7; 94 I/O pins for F3 and 94 I/O pins for F5.
 - The IF, F0, F2, F4, F6, and F8 FPGAs are not connected to the Direct-ICE interface.

16.1.1 Power Supply and Interface Voltage Level

The ZeBu Server Direct-ICE interface provides the following power supplies to the target system:

- 12 V power supply connected directly to the ZeBu Server power supply (250 mA max).
- 1.8 V power supply for the two global clocks on the Direct-ICE interface (250 mA max). For more information, see [Global Clocks](#).
- A voltage for each bank (VCCIO_1 for bank 1 and VCCIO_2 for bank 2), programmable in a [1.2V|1.35V|1.5V|1.8V] range, with 200 mA max per bank. The output tolerance of the VCCIO voltage regulators is $\pm 2.5\%$ in the complete operating range.

Note

The same ZeBu Server power supply provides power to the I/O of the FPGAs and the VCCIO, delivered to the target system through the Direct-ICE interface. Therefore, if the external target power system requires more current, in particular, when it drives its output signals with a high drive value, you need an external power supply for your target system.

16.1.1.1 Supported Voltage Standards

The ZeBu Server Direct-ICE interface is compliant with the Low Voltage Complementary Metal Oxide Semiconductor (LVCMOS) and Stub Series Terminated Logic (SSTL) standards.

The following table provides the compliant matrix for each voltage level of the Direct-ICE interface.

TABLE 36 Supported Standard for Direct-ICE Interface

VCCIO voltage	Supported Standards
1.2	LVCMOS1.2
1.35	SSTL1.35
1.5	LVCMOS1.5
1.8	LVCMOS1.8

16.1.1.2 Direct-ICE Voltage Selection

The voltage level of the Direct-ICE interface is user-programmable.

- For compilation: The voltage level is declared in the Direct-ICE Compilation Script with the `zice voltage` command (see [Connecting an External Direct-ICE Target to a Net in the DUT](#)).
- For runtime: The voltage level is declared in the `<my_setup>.zini` file used by `zSetupSystem` during the system setup to generate the `$ZEBU_SYSTEM_DIR/config.dff` file.

Note

Ensure that the voltage value declared for compilation matches the voltage value generated in `$ZEBU_SYSTEM_DIR/config.dff` (By default, the voltage value is 1.2V if no voltage value was declared in `<my_setup>.zini` before the system setup).

To declare the voltage values for bank 1 and bank 2, perform the following steps:

Functional Description of Direct-ICE Clocks

1. In <my_setup>.zini file, navigate to the MISC.DECLARATIONS section present at the end of the <my_setup>.zini file.
2. Un-comment the lines corresponding to the voltage values that you want to declare for bank 1 and bank 2.

Example

To set VCCIO_1 to 1.8V and VCCIO_2 to 1.2V for Direct-ICE on U0.M0, the <my_setup>.zini file is modified in the following code:

```
#$U0.M0.Directice.VccIo_1 = "1.2 V";
#$U0.M0.DirectIce.VccIo_1 = "1.35 V";
#$U0.M0.DirectIce.VccIo_1 = "1.5 V";
$U0.M0.DirectIce.VccIo_1 = "1.8 V";
$U0.M0.DirectIce.VccIo_2 = "1.2 V";
#$U0.M0.DirectIce.VccIo_2 = "1.35 V";
#$U0.M0.DirectIce.VccIo_2 = "1.5 V";
#$U0.M0.Directice.VccIo_2 = "1.8 V";
```

Note

If no voltage value is declared for a given bank in the <my_setup>.zini file, the voltage value is set to 1.2V in \$ZEBU_SYSTEM_DIR/config.dff.

16.2 Functional Description of Direct-ICE Clocks

The clock connections between a ZeBu Server system and a Direct-ICE target system can be performed in different ways, using one of the following clocks:

- Global Output Clocks (gclkout)
- Global Input clocks (gclkin)
- Local Output Clocks (fclkout)

Note

Local Input clocks (f_{clkin}) are not implemented.

16.2.1 Global Clocks

Two Global Output Clocks are connected to the ZeBu Server clock generator (in the IF of the FPGA module).

Note

A Global Clock can only be used as Global Output or Global Input Clock; it cannot be used as a bi-directional clock.

16.2.1.1 Global Output Clocks

Using Global Clocks as Global Output Clocks is recommended when the clock signal:

- is a primary clock generated by the ZeBu Clock Generator, and
- is declared in the DVE file with a `zceiClockPort` or `zClockPort` instantiation

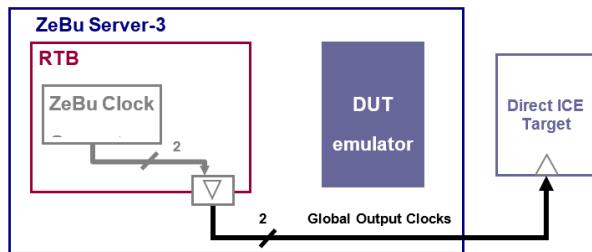


FIGURE 81. Direct-ICE Global Output Clocks Connection

16.2.1.2 Local Output Clocks

Any DUT signal can be connected as a Local Output Clock. If the Local Output Clock is generated in a distant FPGA with respect to the selected Direct-ICE pin, it is routed by the compiler with an optimized skew. However, the latency of the routed signal may reduce the achievable runtime frequency.

16.3 Functional Description of Direct-ICE I/O Signals

16.3.1 Direct-ICE Target Integration With the DUT

The Direct-ICE target system can be integrated with the DUT in different ways:

- The Direct-ICE target system handles the DUT. This configuration is known as an external target.
- The Direct-ICE target system is a sub-module of the DUT, for example, integrating an embedded core in the DUT. This configuration is known as an internal target.
- The Direct-ICE target system includes one or more sub-modules of the DUT and also has a driver part for the DUT.

In addition, the connection between the DUT and the Direct-ICE target system can be made to a net in the design or to a port of a DUT instance. The declaration of connections between DUT signals and the Direct-ICE target signals is always ZeBu centric, as displayed in the following figure.

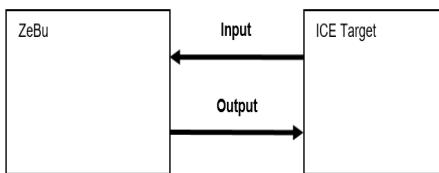


FIGURE 82. ZeBu-Centric I/O Convention

The Direct-ICE interface supports bi-directional and tri-state connections to nets and ports. It is possible to interconnect some pins of the Direct-ICE interface.

16.3.2 Connecting an External Direct-ICE Target to the DUT

An external Direct-ICE target can be connected to a net in the DUT or to DUT top-level ports. The following figure explains about connecting the external Direct-ICE target to the DUT.

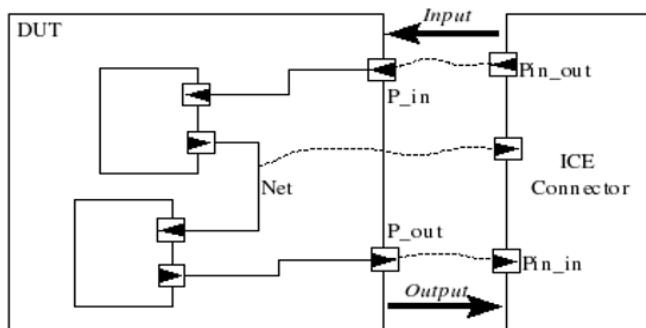


FIGURE 83. Connecting an External Direct-ICE Target to the DUT

16.3.2.1 Connecting an External Direct-ICE Target to a Net in the DUT

When connecting a DUT internal net to an external Direct-ICE target, the following conditions are applied:

- If the signal is driven by the DUT implemented in ZeBu, the connection is declared as an output.
- If the signal is driven by the Direct-ICE target, the connection is declared as an input.

If the signal is driven by a tri-state buffer, this signal is also disconnected and replaced by a binary driver from the Direct-ICE target.

16.3.2.2 Connecting an External Direct-ICE Target to DUT Top-Level Ports

When the Direct-ICE target system handles the DUT, an input port of the design is connected to an output port of the target system (and vice versa) and the connection is declared according to the *ZeBu-centric* convention.

16.3.3 Connecting an Internal Target to the DUT

When the target is a sub-module of the DUT, the direction of the port on the target is the same as on the DUT as displayed in the following figure:

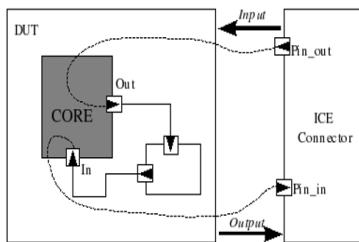


FIGURE 84. Connecting an Internal Target to the DUT

16.3.4 Delays

At compilation time, blocks connected to the Direct-ICE target are delayed by an asynchronous delay to limit race conditions in the design.

Note

The delay step (10, 20, or 40 nanoseconds (ns)) is declared at compilation time and cannot be changed at runtime.

The number of delay steps ranges from 1 to 16. A preset value of delay step is defined when compiling the design; but it can be changed at runtime.

Note

Delay blocks can be disabled at runtime, which means that the delay is set to 0 ns.

In the Direct-ICE Compilation Script, you can declare the absolute delay and the delay step as follows:

- **If only a delay step is specified**, the delay step value is used. By default, the delay block is disabled at runtime (0 ns delay).
- **If only an absolute delay is specified**, the delay step is automatically selected according to the values in [Table 37](#) and the delay is rounded to the nearest

available value. At runtime, the number of delay steps is preset to match the rounded delay.

- **If both are specified**, the delay step is used and the delay is rounded to the nearest available value. At runtime, the number of delay steps is preset to match the rounded delay.

The delay can be set to 0 ns at compilation time so that the signals are not delayed. However, the delays are set by the design for runtime programming. If the delay is declared as 0 ns, the delay step is set to 20 ns.

In the Direct-ICE Compilation Script, you can declare:

- A default delay or delay step for all Direct-ICE connections.
- A specific delay or delay step for individual connections (see *Direct-ICE Delaying of Clock Signals* and *Delayed I/O Signals*).

The appropriate register instances to enable or disable the delay and to control its duration are available at runtime (see *Delay Activation and Programming*).

- For compilation, the delay and the delay step are declared in nanoseconds.
- For runtime programming, the delay can be changed by modifying the number of delay steps. The delay control value can range from 0 to 15, representing a number of delay steps ranging from 1 to 16. The delay can also be disabled to obtain a delay of 0ns.

TABLE 37 Range and Step Values for Delay Programming at Runtime

Compilation Delay	Delay Step	Runtime Programming	Runtime Range
0 ns - 160 ns	10 ns	0 - 15	0 ns - 160 ns
161 ns - 320 ns	20 ns	0 - 15	0 ns - 320 ns
321 ns - 640 ns	40 ns	0 - 15	0 ns - 640 ns

16.3.5 Direct-ICE Sampling Data

The sampling feature can be used for data alignment with respect to any signal in the DUT. This feature can be activated by dedicated commands in the Direct-ICE compilation script.

The declaration of the default settings for data alignment is described in *Declaring the*

Physical Description

Default Settings for Signals

The option for a data alignment on specific connections is described in [Direct-ICE Declaring Signal Connections for an Internal Target](#).

Note

When the sampling feature is enabled in the Direct-ICE compilation script, the register instances to control the feature through dynamic-probes can be accessed at runtime (see [Sampling Feature](#))

16.4 Physical Description

The Direct-ICE interface provides easy-to-use ERNI connectors, which can be used for cable connections or for board-to-board connections. The following figure displays Direct-ICE interface connectors.

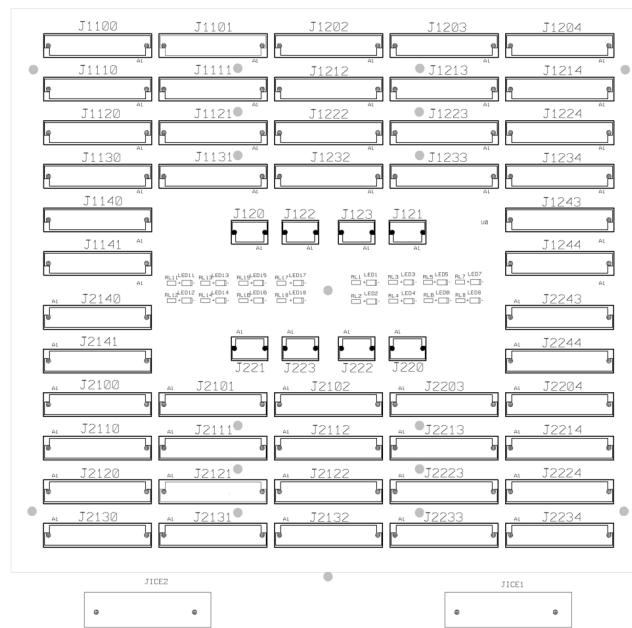


FIGURE 85. Direct-ICE Interface Connectors

16.4.1 Connectors

The following connectors can be used with the Direct-ICE interface:

- **J11xx, J12xx, J21xx, J22xx:** 50-pin connectors are used for data connections.
- **J120:** 12-pin connector provides VCCIO_2 power supply to the target.
- **J121:** 12-pin connector is used for Global Clocks.
- **J122 and J123:** 12-pin connectors provide 12 V power to the target.
- **J220, J221, J222, J223:** unused 12-pin connectors.

16.4.2 LEDs

The following LEDs are available in the Direct-ICE interface:

- **LED1-LED8:** The programmed voltage for each bank is indicated by a lit LED. Each LED indicates 1 of 4 possible voltage values (For more information, see *Direct-ICE LEDs*).
- **LED11-LED18:** unused LEDs.

16.4.3 Direct-ICE Interface Connectors and FPGAs on a Direct-ICE Module

The following figure displays Direct-ICE Interface Connectors versus FPGAs with a 9F/ICE FPGA Module.

Physical Description

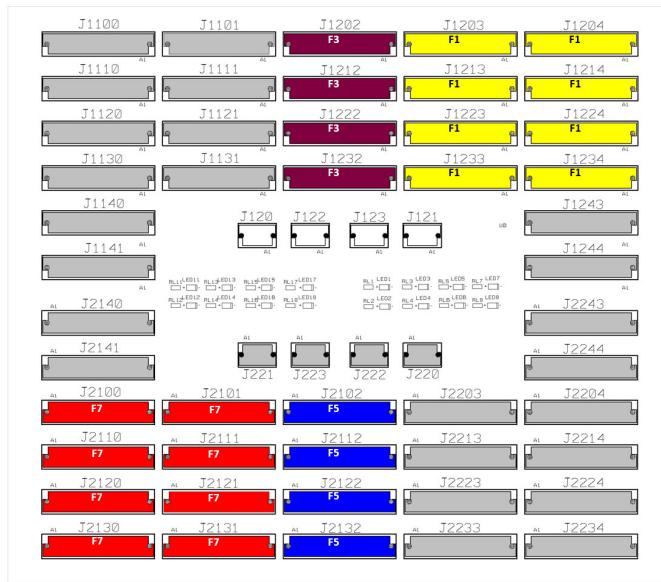


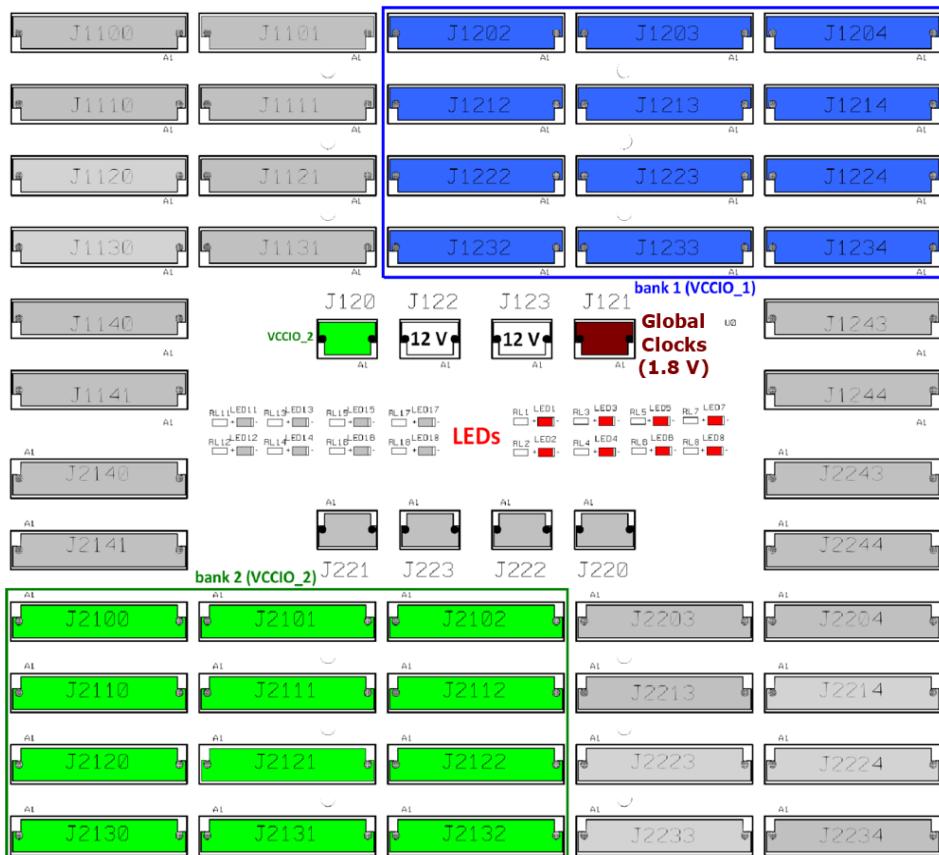
FIGURE 86. Direct-ICE Interface Connectors Versus FPGAs with a 9F/ICE FPGA Module

Note

- ▀ Grayed out connectors are not used with the 9F/ICE FPGA module.
- ▀ The other connectors are color-coded to display which FPGA they are connected to on the Direct-ICE module. For instance, all the Direct-ICE interface connectors indicated in red are connected to FPGA F7 on the Direct-ICE module.

16.4.4 Direct-ICE Banks

The connectors used in the Direct-ICE interface are distributed in two connector banks. Each bank supports one programmable voltage level(VCCIO_1 or VCCIO_2) as displayed in the following figure.

**FIGURE 87.** Direct-ICE Connectors with Bank Distribution for one 9F/ICE Module**Note**

- Grayed out connectors and LEDs are not used with the 9F/ICE module.
- No dedicated 12-pin connector provides VCCIO_1 power supply to the target. VCCIO_1 is provided by all J1xxx data connectors of bank 1 (see [Data Connectors](#))

Physical Description

16.4.5 Direct-ICE Data Connectors Mapping

All used data connectors have 25 I/Os, except:

- **J1232** of VCCIO_1 and **J2102** of VCCIO_2, which have 19 I/Os.
- **J1203** of VCCIO_1 and **J2131** of VCCIO_2, which have 15 I/Os.

The following figure displays the mapping of Direct-ICE data connectors.

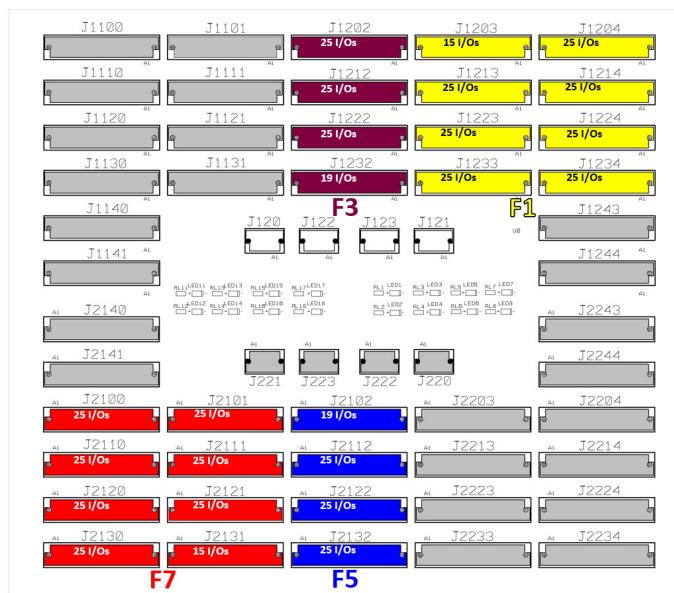


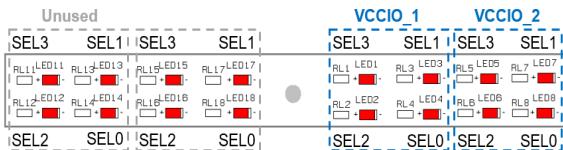
FIGURE 88. Direct-ICE I/Os Mapping

Note

Grayed out connectors and LEDs are not used with the 9F/ICE module.

16.4.6 Direct-ICE LEDs

Direct-ICE LEDs are placed in the middle of the Direct-ICE interface board. When an LED is lit, it indicates the voltage value for one of the connector banks.

**FIGURE 89.** LEDs on the ZeBu Server 3 Direct-ICE Interface board

The eight rightmost LEDs are dedicated to the voltage values for the two banks:

- SEL0 -> 1.8 V
- SEL1 -> 1.5 V
- SEL2 -> 1.35 V
- SEL3 -> 1.2 V

Example:

- If LED1 (VCCIO_1 SEL3) is ON, VCCIO_1 is powered by 1.2V.
- If LED8 (VCCIO_2 SEL0) is ON, VCCIO_2 is powered by 1.8V.

Note

Silkscreen markings on the board for LED1 (IO2_SEL3) and LED2 (IO2_SEL2) are wrong. They should read IO1_SEL3 and IO1_SEL2, respectively.

16.4.7 Direct-ICE Connectors Physical Description

Two types of connectors are present on the ZeBu Server Direct-ICE interface for target system interconnection:

- 50-pin connectors for data I/O
- 12-pin connectors for clocks and 12V power supply

The following table lists the appropriate references (ERNI Part Numbers) for cable connection and board-to-board connection:

Physical Description
TABLE 38 References for ERNI Connectors

Connector Series	ERNI Part Number for 50-pin Data Connector	ERNI Part Number for 12-pin Connector	
ZeBu Server 3 Direct-ICE	ERNI SMC Vertical Male type Q low profile.	244838	244836
Cable Assemblies	ERNI flat cable with Female Connector SMC, Type B, with locking latches.		
	300 mm	173795	173801
	200 mm	173794	173800
	100 mm	173793	173799
Board-to-Board connection	ERNI SMC Vertical Female type B low profile.	154807	154805

The connectors' datasheets are available on the ERNI website: <http://www.erni.com>.

16.4.8 Direct-ICE Clocks Physical Description

This section provides physical description about the Direct-ICE clocks.

16.4.8.1 Global Clocks

Two global clocks are available on the Direct-ICE interface (on dedicated connector J121). For the connector pinout for Global Clocks, see [Global Clocks Connector \(J121\)](#).

16.4.8.2 Local Output Clocks

Local Output Clocks are connected to the pins of the data I/O connectors present on the Direct-ICE interface and not to the dedicated connectors.

16.4.9 Direct-ICE Pinouts

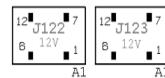
This section provides details about a list of pins/connectors available on the Direct-ICE interface module.

16.4.9.1 12 V Power Connectors

The pinout for the 12 V power connectors J122 and J123 is listed in the following table:

TABLE 39 Pinout for 12V Power Supply Connectors J122 and J123

Pin #	Identification	Pin #	Identification
1	12V	7	GND
2	12V	8	GND
3	12V	9	GND
4	GND	10	12V
5	GND	11	12V
6	GND	12	12V



Note

The maximum available current for all connectors is 250 mA. If your target system requires more current, you need an external power supply.

16.4.9.2 Global Clocks Connector (J121)

The Global Clocks connector (J121) is a 12-pin ERNI connector. It provides the two Global Clock signals and 1.8 V power supply. The following table lists the pinout of the Global Clocks Connector.

Physical Description

TABLE 40 Pinout for Global Clocks Connector (J121)

Pin #	Identification	Pin #	Identification
1	1.8V	7	GND
2	GND	8	Reserved
3	GND	9	Reserved
4	GND	10	GCLK_0 (*)
5	GND	11	GCLK_0 (*)
6	1.8V	12	GND

**Note**

The maximum available current on the 1.8 V power supply is 250 mA, irrespective of the voltage settings for bank 1 and bank 2.

16.4.9.3 VCCIO_2 Connector

VCCIO_2 connector (J120) is a dedicated 12-pin ERNI connector. The following table lists the pinout of the VCCIO_2 connector.

TABLE 41 Pinout for VCCIO_2 Connector (J120)

Pin #	Identification	Pin #	Identification
1	VCCIO_2	7	GND
2	GND	8	Reserved
3	GND	9	Reserved
4	GND	10	Reserved



TABLE 41 Pinout for VCCIO_2 Connector (J120)

Pin #	Identification	Pin #	Identification
5	GND	11	Reserved
6	VCCIO_2	12	GND

Note

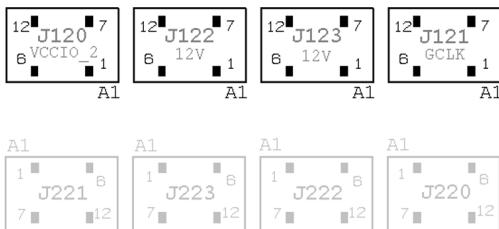
No dedicated 12-pin connector provides VCCIO_1 power supply to the target. All the J1xxx data connectors of bank 1 provide this power supply as described in [Data Connectors](#).

16.4.9.4 Pinout Summary for 12-pin Connectors

The following table lists pinout for 12-pin connectors.

TABLE 42 Pinout for 12-pin Connectors

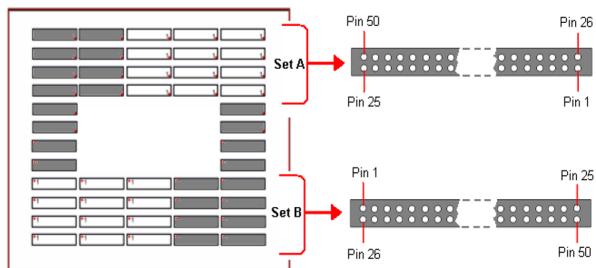
Pin #	J120	J121	J122	J123
1	VCCIO_2	1.8V	12V	12V
2	GND	GND	12V	12V
3	GND	GND	12V	12V
4	GND	GND	GND	GND
5	GND	GND	GND	GND
6	VCCIO_2	1.8V	GND	GND
7	GND	GND	GND	GND
8	Reserved	Reserved	GND	GND
9	Reserved	Reserved	GND	GND
10	Reserved		12V	12V
11	Reserved		12V	12V
12	GND	GND	12V	12V

Physical Description

FIGURE 90. 12-pin Connectors Pinout

16.4.9.5 Data Connectors

The data connectors are grouped in two sets of symmetrically organized connectors:

- Set A is the upper set. All connectors of Set A have the same orientation (pin 1 in the bottom-right corner, marked A1 on the board).
- Set B is the lower set. All connectors of Set B have the same orientation (pin 1 in the top-left corner, marked A1 on the board).


FIGURE 91. Pinout for Data Connectors

The pinout of the data connectors is listed in the following table:

TABLE 43 Generic Pinout Information for Data Connectors

Set	Data Connectors	Data I/O	Supply Voltage VCCIO_1	Ground
A	J1203	26 to 41	1 and 25	2 to 24
	J1232	26 to 45		
	All other connectors	26 to 50		
B	J2102	26 to 45		
	J2131	26 to 41		
	All other connectors	26 to 50		

Note

Only the connectors of set A provide VCCIO on pin 1 and pin 25. The corresponding pins must be left unconnected on the target board on the connectors of set B.

16.4.10 Data and Clock Timings for Direct-ICE Interface

The Direct-ICE target system can be clocked by a Global Output Clock (g_{clkout}) or by a Local Output Clock (f_{clkout}). The following figure illustrates the conditions to be met for timing measurements.

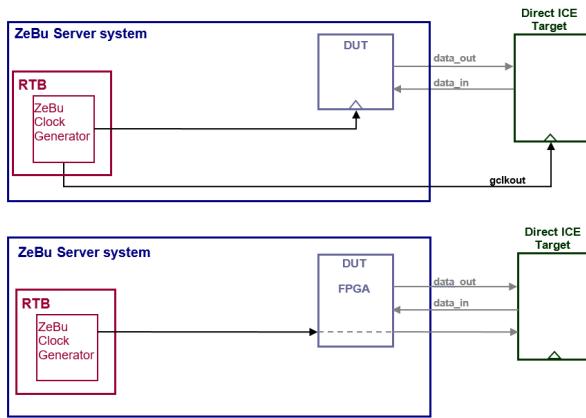
Physical Description


FIGURE 92. Conditions for Timing Measurements With Output Clocks

For characterization purposes, the Local Output Clock is a primary clock that is generated by the ZeBu Clock Generator and goes through a design FPGA.

For data inputs of ZeBu (coming from the target), setup time (t_{Setup}) and hold time (t_{Hold}) are measured and defined as displayed in the following figure:

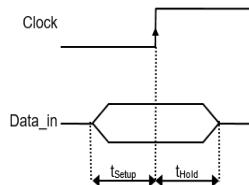


FIGURE 93. Definition of Timings for Input Data

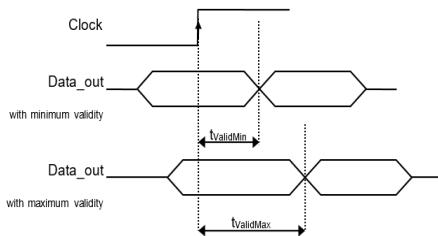
The following table lists timings for each data inputs.

TABLE 44 Timing for Data Inputs

Clock Type	$t_{SetupMin}$	$t_{HoldMin}$
Global Output		0
Clock (gclkout)	$\frac{1}{F} + \Delta t_{input}$	
Local Output		0
Clock (fclkout)	$\frac{1}{F} + \Delta t_{input}$	

where, F is the maximum driverClock frequency computed by Static Timing Analysis and Δt_{input} is the user-defined delay added on the data input.

For data outputs of ZeBu (going to the target), the maximum and minimum delays during which the data is valid after the clock edge are defined as shown in the following figure:

**FIGURE 94.** Definition of Timings for Output Data

Physical Description

The following table lists timings for each data outputs.

TABLE 45 Timing for Data Outputs

Clock Type	$t_{validMin}$	$t_{validMax}$
Global Output Clock (gclkout)	$10\text{ns} + \Delta t_{output}$	$\frac{1}{F} + \Delta t_{output}$
Local Output Clock (fclkout)	$5\text{ns} + \Delta t_{output}$	$\frac{1}{F} + \Delta t_{output}$

where, F is the maximum driverClock frequency computed by **zPar** or **zTime** and Δt_{output} is the user-defined delay added on the data output pin.

Note

The five ns difference between Global and Local Output Clocks for $t_{validMin}$ is due to the hop in the Direct-ICE CPLD.

For output clocks, the maximum skew between two clocks is defined as the delay between the arrivals of two clocks at the Direct-ICE connectors:

TABLE 46 Maximum Skew for Output Clocks

Maximum Clock Skew	
Between 2 Global Output Clocks (gclkout)	3ns
Between 2 Local Output Clocks (fclkout)	3ns
Between Global Output Clocks and Local Output Clocks	15ns

16.5 Writing the Direct-ICE Compilation Script

Before you compile for Direct-ICE, you must write the Direct-ICE Compilation Script. In this Tcl script, you describe the configuration of the Direct-ICE interface and declare the interconnections between the Direct-ICE target system and the DUT mapped in the ZeBu Server 3 system.

Later, the Direct-ICE Compilation Script is added in the UFT project file (see [Compiling for Direct-ICE](#)).

Three types of commands can be used in the Direct-ICE Compilation Script:

- zice commands for global settings.
- zice_clock commands for clock connections.
- zice_io commands for data connections.

The global settings are declared with one of the zice commands listed in [Table 51](#). The global settings apply to all subsequent connection commands (zice_clock and zice_io) until the following zice command is met in the Direct-ICE Compilation Script. Hence, the order in which zice commands and connection commands appear is very important. Any new zice command affects the next connection commands. The connection commands sharing the same global settings can be declared in any order since the consistency of connections is automatically verified by the compiler.

16.5.1 Direct-ICE Compilation Script Commands

This section describes the list of commands used in the Direct-ICE Compilation Script.

16.5.1.1 Global Commands

The following table describes the global commands.

TABLE 47 Global Commands (`zice`)

Commands	Description	See Section
<code>zice voltage</code>	Identifies a connector bank and declares a voltage for the bank.	<i>Declaring the Voltage Level</i>
<code>zice default (*)</code>	Applies a default parameter to all subsequent <code>zice_clock</code> and <code>zice_io</code> commands.	<i>Declaring the Default Settings for Signals</i>
<code>zice shield_ios</code>	Configures I/O shielding for the Direct-ICE module currently selected in zCui .	<i>Disabling the Direct-ICE Interface Output Connections</i>

(*) Settings declared with these commands can be overridden by `zice_clock` and/or `zice_io` commands.

16.5.1.2 Clock Commands

The following table describes the clock commands.

TABLE 48 Clock Commands (`zice_clock`)

Command	Description
<code>zice_clock gclkout</code>	Connects a Global Output Clock (generated by ZeBu) to the Direct-ICE target.

TABLE 48 Clock Commands (`zice_clock`)

Command	Description
<code>zice_clock gclkin</code>	Connects a Global Input Clock (coming from the Direct-ICE target to the ZeBu clock generator). This clock is routed as low-skew into the design FPGAs of a single unit.
<code>zice_clock fclkout</code>	Connects a Local Output Clock (from a Direct-ICE FPGA) to the Direct-ICE target.

16.5.1.3 Data Commands

The following table describes the data commands.

TABLE 49 Data Commands (`zice_io`)

Command	Description
<code>zice_io input</code>	Connects a DUT signal as an input. Connects a DUT signal as an output of the design when <code>-target_path</code> is specified.
<code>zice_io output</code>	Connects a DUT signal as an output. Connects a DUT signal as an input of the design when <code>-target_path</code> is specified.
<code>zice_io inout</code>	Connects a DUT signal as bi-directional.
<code>zice_io vcc</code>	Drives a Direct-ICE interface pin to one.
<code>zice_io gnd</code>	Drives a Direct-ICE interface pin to zero.

16.5.2 Direct-ICE Compilation Script options

This section provides details about the script options available in the Direct-ICE Compilation Script.

16.5.2.1 Strictly Global Options

The following table lists the options that can only be used with global commands.

TABLE 50 Strictly Global Options (`zice` only)

zice Command	Option	Parameter	Description	See Section
voltage	-vcc_bank	<VCCIO_1 VCCIO_2>	Specifies the name of the bank for which a declared voltage is affected.	<i>Declaring the Voltage Level</i>
	-value	<1.2v 1.35v 1.5v 1.8v 1200mV 1350mV 150mV 1800mV>	Declares a voltage value for power connectors and for a bank of I/O connectors.	
shiel_ios	-shielding_enable	<yes no>	Adds the logic so that I/O shielding can be activated at runtime.	<i>Disabling the Direct-ICE Interface Output Connections</i>
	-shielding_active	<yes no>	When shielding is enabled, activates the shielding when the run starts.	

16.5.2.2 Overridable Default Options

Any value declared with `zice` is overridden if it is declared with `zice_clock` or `zice_io` at the runtime. Similarly, any value declared with `zice_clock` or `zice_io` is overridden when it is declared with `zice` afterwards. The following table lists the options that can be overridden during the runtime.

TABLE 51 Overridable Default Options (`zice` and `zice_clock / zice_io`)

zice Command	Option	Parameter	Description	Applicable for <code>zice_clock</code>	Applicable for <code>zice_io</code>
default	<code>-delay (1)</code>	<code><delay_value></code> Range: 0 ns – 640 ns	Adds a delay (with the declared duration) on all Local Output Clocks and all I/O signals.	fclkout	input output inout
default	<code>-delay_en (1)</code>	<code><delay_value></code> Range: 0 ns – 640 ns	Adds a delay (with the declared duration) on the enable pin of all I/O signals.		inout
default	<code>-delay_data (1)</code>	<code><delay_value></code> Range: 0 ns – 640 ns	Adds a delay (with the declared duration) on the output pin of all I/O signals.		inout
default	<code>-delay_fclk_out (1)</code>	<code><delay_value></code> Range: 0 ns – 640 ns	Adds a delay (with the declared duration) on all Local Output Clocks.		

Writing the Direct-ICE Compilation Script

TABLE 51 Overridable Default Options (`zice` and `zice_clock / zice_io`)

zice Command	Option	Parameter	Description	Applicable for <code>zice_clock</code>	Applicable for <code>zice_io</code>
default	-delay_step (2)	<10ns 20ns 40 ns>	Adds a delay (with the declared delay step) on all Local Output Clocks and all I/O signals.	fclkout	input output inout
default	-slew	<SLOW FAST>	Sets a slew on all Local Output Clocks and all I/O signals.	fclkout	output inout
default	-drive	<2 4 6 8 12 16 24>	Sets connection strength on all Local Output Clocks and all I/O signals.	fclkout	output inout
default	-sampling (3)	<reg latch pinmux idr odd r>	Aligns data on all I/O signals.		input output inout
default	-sampling_en (3)	<reg latch>	Aligns data on the enable pin of all I/O signals.		inout
default	-sampling_data (3)	<reg latch>	Aligns data on the output pin of all I/O signals.		inout

16.5.2.3 Strictly Clock/Data Options

The following table lists the options that are only applicable for clock/data.

TABLE 52 Strictly Clock and/or Data Options (`zice_clock / zice_io`)

Option	Parameter	Description	Applicable for <code>zice_clock</code>	Applicable for <code>zice_io</code>
<code>-con_loc</code>	<code><con_id>.<pin></code>	Identifies a pin on a connector of the Direct-ICE interface.	fclkout gclkout gclkin	input output inout gnd vcc
<code>-dve_source</code>	<code><sig_name></code>	Sets a signal specified in the DVE file for connection to the Direct-ICE target.	fclkout gclkout	Output
<code>-inv</code>		Inverts the polarity of the connected signal.		input output
<code>-loop</code>	<code><loop_id></code>	Defines a loop label. The connected input and output signals must use the same label.		input output
<code>-net</code>	<code><net_name></code>	Identifies the hierarchical name of the wire to connect.	fclkout	input output inout
<code>-pin</code>	<code><port_name></code>	Identifies the port to connect.	fclkout gclkin	input output inout

TABLE 52 Strictly Clock and/or Data Options (`zice_clock` / `zice_io`)

Option	Parameter	Description	Applicable for <code>zice_clock</code>	Applicable for <code>zice_io</code>
<code>-pull</code>	<code><keeper pullup pulldown></code>	Sets a resolution function for tri-state pads.	fclkout	output inout
<code>-target_path</code>	<code><instance_path></code>	Identifies the hierarchical path to an internal target (blackbox).	fclkout	input output inout

16.5.3 Configuring the Direct-ICE Interface

Direct-ICE default settings can be declared with the `zice` commands described in this section. All the options, which can be used with `zice` commands, are listed in [Table 50](#) and [Table 51](#).

16.5.3.1 Declaring the Voltage Level

The voltage level for each bank of the Direct-ICE interface must be declared using the `voltage` option. There is no default setting for the voltage level. The compiler exits with an error if this command is not specified.

Syntax

```
zice voltage -vcc_bank <bank_name> -value <voltage_value>
```

where:

- `<bank_name>` is `VCCIO_1` or `VCCIO_2`.
- `<voltage_value>`: 1.2 V, 1.35 V, 1.5 V, or 1.8 V.

Example

To set bank 1 (VCCIO_1) to 1.8 V and bank 2 (VCCIO_2) to 1.5 V:

```
zice voltage -vcc_bank VCCIO_1 -value 1800mV  
zice voltage -vcc_bank VCCIO_2 -value 1.5v
```

Note

*Verify that the voltage value declared here is the same as the one specified in the \$ZEBU_SYSTEM_DIR/config.dff file generated during system setup by **zSetupSystem** (for details, see [Direct-ICE Voltage Selection](#)).*

16.5.3.2 Declaring the Default Settings for Signals

In the Direct-ICE compilation script, you can define the following default settings for the connections to the Direct-ICE interface:

- Compilation constraints
- Default Delay
- Tri-state resolution
- Output data sampling

Note

It is recommended to add the zice default settings at the beginning of your Direct-ICE compilation script. It avoids conflicting sequences with individual settings in the connection commands. In case of conflicting command sequences, the latest value is considered.

Default Compilation Constraint

For output signals, you can force some compilation constraints (similar to Xilinx ISE constraints).

Syntax

```
zice default -slew <slew_mode> -drive <drive_value>
```

where,

Writing the Direct-ICE Compilation Script

- <slew_mode> can be FAST (produces a faster output; but increases noise and power consumption) or SLOW. When the <slew_mode> is not declared, by default, FAST is used.
- <drive_value> selects the output drive strength. Possible values are: 2 , 4 , 6 , 8 , 12 , 16 , or 24. Ensure that you use high drive values with care so as to limit the overall current consumption and possible overheating of the Direct-ICE FPGAs. When the default value is not declared, 4 is used by default. For the appropriate output drive strength, see the Xilinx Virtex-7 FPGA documentation.

Example

To set the default for slew_mode to "SLOW" and the default drive_value to 8 for all signals:

```
zice default -slew SLOW -drive 8
```

Note

The same ZeBu Server power supply feeds the I/O of the Direct-ICE FPGAs and the VCCIO delivered to the target system through the Direct-ICE interface. Therefore, if your target system requires more current, in particular, when it drives its output signals with a high drive_value, you need an external power supply for your target system.

Default Delay

For synchronization purposes, a given signal is delayed.

By default, the delay can be declared for the following:

- All I/O signals (input, output, and bi-directional), using the -delay option.
When the delay must be reduced for bi-directional signals, you can use:
 - The -delay_en option (enable pin of all bi-directional signals).
 - The -delay_data option (output pin of all bi-directional signals).
- Local Output Clocks (using the -delay_fclkout option).
By default, the delay step can be set for the following:
- All I/O signals (input, output, and bi-directional), using the -delay_step option with or without -delay.

When the delay must be reduced for bi-directional signals, you can use the `-delay_step` option with:

- The `-delay_en` option (enable pin of all bi-directional signals).
- The `-delay_data` option (output pin of all bi-directional signals).
- Local Output Clocks: `-delay_step` cannot be declared separately.

Note

- *The `-delay/-delay_en/-delay_data` commands have an effect on subsequent `zice_io` commands. On the other hand `-delay_fclkout` has an effect on subsequent `zice_clock_fclkout` commands.*
 - *-delay_step can be declared in several commands but only the delay step value declared in the last command is taken into account.*
 - *When you set the delay without setting the delay step, the latter is automatically inferred (most likely to 20 ns).*
 - *The delay declared for compilation impacts the range for runtime programming (see [Table 53](#))*
-

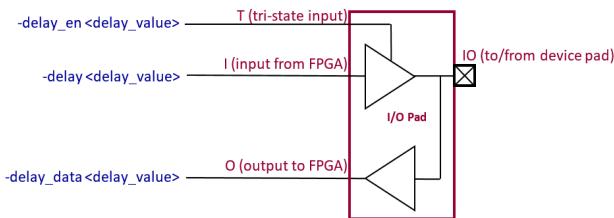
Syntax

```
zice default -delay <delay_value> -delay_step <step_value>
zice default -delay_en <delay_value> -delay_step <step_value>
zice default -delay_data <delay_value> -delay_step <step_value>
zice default -delay_fclkout <delay_value> -delay_step <step_value>
```

where, `<delay_value>` is an integer value in nanosecond (maximum: 640 ns) and `<step_value>` can be 10 ns, 20 ns, or 40 ns.

The following figure displays delay options for an I/O pad:

Writing the Direct-ICE Compilation Script

**FIGURE 95.** Delay Options for I/O Pad

The following table lists the step values and its range for delay programming at runtime.

TABLE 53 Range and Step Values for Delay Programming at Runtime

Compilation Delay	Delay Step	Runtime Programming	Runtime Range
0 – 160 ns	10 ns	0 - 15	0 – 160 ns
161 – 320 ns	20 ns	0 - 15	0 – 320 ns
321 – 640 ns	40 ns	0 - 15	0 – 640 ns

If the delay is set to 0 ns:

- The signals are not been delayed.
- The default delay step is set to 20ns.
- The delay can be programmed at runtime (For more information, see [Delay Activation and Programming](#)).

Note

- ▀ If the delay is not compatible with the delay step, the delay step is assumed to be correct and the delay is adjusted accordingly.
- ▀ If delay and delay step values declared with the zice_io and zice_clock commands, override the values declared with the zice default command, if any.

Direct-ICE Default Data Sampling

Sampling consists of aligning data with respect to a reference clock and inserting a latch or a register in one of the following items (based on declaration in the Direct-ICE compilation script):

- All data (using the `-sampling` option).
- Enable pin of all bi-directional data (using the `-sampling_en` option).
- Output pin of all bi-directional data (using the `-sampling_data` option).

Note

`-sampling`, `-sampling_en`, and `-sampling_data` have an effect on subsequent `zice_io` commands, if any. A user clock (instantiated in the DVE file) must also be declared in the same command.

Syntax

```
zice default -sampling <sampling_mode> -sampling_clock  
<clock_name>  
zice default -sampling_en <sampling_mode> -sampling_clock  
<clock_name>  
zice default -sampling_data <sampling_mode> -sampling_clock  
<clock_name>
```

where:

- `<sampling_mode>`: its value can be `reg` or `latch`.
- `<clock_name>` is a primary clock declared in the DVE file.

To use the sampling feature on a specific signal, use the `zice_io` command.

For Example:

To align data with reference clock `ck1`, insert a latch as follows:

```
zice default -sampling_data latch -sampling_clock ck1
```

16.5.3.3 Disabling the Direct-ICE Interface Output Connections

For debugging purposes, disable the output connections of the Direct-ICE interface.

Syntax:

To disable the output connections of the Direct-ICE interface, enable shielding as follows:

```
zice shield_ios -shielding_enable yes -shielding_active yes
```

where,

- `-shielding_enable yes`: Enables shielding.
- `-shielding_active yes`: Activates shielding when the emulation starts.

To make the shielding feature inactive, set `-shielding_active` to no:

```
zice shield_ios -shielding_enable yes -shielding_active no
```

To disable the shielding feature, set `-shielding_enable` to no:

```
zice shield_ios -shielding_enable no
```

The registers are available in the runtime database (one for each Direct-ICE FPGA) to control the shielding feature for each Direct-ICE FPGA.

```
DirectIce.io_shielding.<fpga_name>.shield_enable
```

where, `<fpga_name>` is UX_MX_FX (Unit, Module, and FPGA IDs).

These registers can be selected in **zSelectProbes** for runtime programming from **zRun** or from the user testbench. For more details, see [Control of the Direct-ICE Shielding](#).

16.5.4 General Rules for Signal Connections

The signal connection section of the Direct-ICE compilation script defines the matching between the logical wires of the DUT and the physical pin locations on the Direct-ICE interface. A connection (input, output, or bi-directional) is done to a net, a port of the DUT, or a DVE source.

16.5.4.1 Location of a Pin on the Direct-ICE Interface

The location of a global clock pin (in `zice_clock` commands) or an I/O pin (in `zice_io` commands) on the Direct-ICE interface is identified by
`<con_id>. <pin>`.

where, `<con_id>` is the connector ID, and `<pin>` is the pin ID on the connector.

16.5.4.2 Location of a Global Clock pin on the Direct-ICE interface

For global clock pins: `<con_id> = J121`; `<pin> = 10 or 11`.

For example:

```
zice_clock gclkin -dve_source clk -con_loc J121.10
```

16.5.4.3 Location of an I/O pin on the Direct-ICE interface

For I/O pins: `<con_id> = Jxxxx` (`xxxx`: four digits); `<pin>` is two-digit ID.

For example:

```
zice_io input -net {dut.b_core2.di1[0]} -con_loc J1204.26
```

16.5.4.4 Signal Direction

The direction of a signal is always ZeBu-centric:

- **Input:** Establishes a connection in ICE to ZeBu direction.
- **Output:** Establishes a connection in ZeBu to ICE direction.
- **Inout:** Establishes a bi-directional connection between ZeBu and ICE.

Writing the Direct-ICE Compilation Script

When declaring an input, the signal can only be driven by the Direct-ICE target system. The other drivers for this signal in the design are automatically disconnected and a warning is displayed (and stored in the system-level compiler log file). Only tri-state signals can be connected as bi-directional nets.

Note

- When Direct-ICE pins are not connected in the Direct-ICE compilation script, they are not driven by the Direct-ICE FPGAs (tri-state pins).
- When the Direct-ICE feature is not used in a compilation, the related FPGA pins are not powered and remain configured in tri-state mode. Therefore, you can proceed with a connected and active target while running a non-Direct-ICE emulation.

16.5.4.5 Using the `-pin` Option Alone

When the `-pin` option is used alone, the parameter passed with this option is `<top_name>.<port_name>` or `<top_name>.<clock_name>` (hierarchical path to the port or to the clock from the top).

In the following figure:

- For p1 (port in the DUT), use: `-pin top.path.p1`
- For p2 (port of the DUT), use: `-pin top.p2`

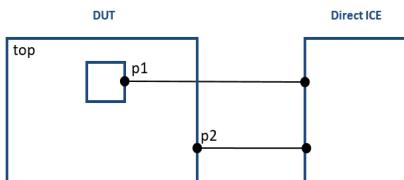


FIGURE 96. Using the `-pin` Option Alone

16.5.4.6 Using the `-pin` Option With `-target_path`

When the `-target_path` option and the `-pin` option are used in the same command, the parameter passed with the `-pin` option is `<port_name>` for I/O signals and `<clock_name>` for clocks (with no hierarchical path).

The `-target_path` option provides the path of the blackbox which is substituted by the target.

In the following figure for p1 (port of the blackbox), use: `-target_path blackbox -pin p1`

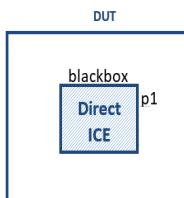


FIGURE 97. Using the `-pin` Option with `-target_path`

16.5.4.7 Bi-directional Signals

The signals declared as bi-directional (using the `zice inout` commands) in the Direct-ICE compilation script must be tri-state signals.

- When one of the tri-state drivers in the design is enabled, the bi-directional signal behaves like an output and drives the target.
- When no driver of the design is enabled, the bi-directional signal behaves like an input and the value of the Direct-ICE target is read by the design.
- When neither the design nor the Direct-ICE target is driving, the value of the resolution function (see *Tri-State Resolution on I/O Signals*) is read.

16.5.5 Connecting Clocks

The clock connections can be declared with the `zice_clock` commands. The options that can be used with `zice_clock` commands are listed in [Table 51](#) and [Table 52](#).

16.5.5.1 Direct-ICE Delaying of Clock Signals

The delay of clocks can be set by default (only on local output clocks) using the zice default -delay_fclkout command. It can also be declared on individual clocks using the zice_clock command.

These settings override the corresponding default setting described in [Default Delay](#).

Syntax

```
zice_clock fclkout -dve_source <clock_name> -con_loc
<con_id>.<pin>
  -delay <delay_value> -delay_step <step_value>
zice_clock fclkout -pin <clock_name> -con_loc <con_id>.<pin>
  -delay <delay_value> -delay_step <step_value>
zice_clock fclkout -net <clock_name> -con_loc <con_id>.<pin>
  -delay <delay_value> -delay_step <step_value>
```

where,

- <delay_value> is an integer value in nanoseconds (maximum: 640 ns).
- <step_value> is can be 10 ns, 20 ns, or 40 ns.

Note

If the delay is not compatible with the delay step, the delay step is assumed to be correct and the delay is adjusted accordingly.

Examples

```
zice_clock fclkout -con_loc J1204.27 -net dut.counter.cnt\[0\]
  -delay 20ns -delay_step 40ns
```

16.5.5.2 Compilation Constraints on Local Output Clocks

To force compilation constraints (similar to Xilinx ISE constraints) on local output

clocks, use the following syntax:

```
zice_clock fclkout -dve_source <clock_name> -con_loc  
<con_id>. <pin>  
    -slew <slew_mode> -drive <drive_value>  
zice_clock fclkout -pin <clock_name> -con_loc <con_id>. <pin>  
    -slew <slew_mode> -drive <drive_value>  
zice_clock fclkout -net <clock_name> -con_loc <con_id>. <pin>  
    -slew <slew_mode> -drive <drive_value>
```

where,

- <slew_mode> can be FAST (produces a faster output but increases noise and power consumption) or SLOW. By default, the <slew_mode> is declared as FAST.
- <drive_value> selects the output drive strength in mA. Possible values are: 2, 4, 6, 8, 12, 16, or 24. Ensure that you use high -drive values with care to limit the overall current consumption and possible overheating of the Direct-ICE FPGAs. When the default is not declared, a value of 4 is used. For the appropriate output drive strength, see the *Xilinx Virtex-7 FPGA* documentation.

Note

These settings override the corresponding default setting described in Default Delay.

Example

To declare slew and drive constraints (SLOW and 8) for local output clock top.sig_clk connected to pin J2100_30:

```
zice_clock fclkout -pin top.sig_clk -con_loc J2100.30 -slew SLOW -  
drive 8
```

16.5.5.3 Tri-State Resolution on Local Output Clocks

When shielding is active (see [Configuring the Direct-ICE Interface](#)), local output clocks are in high impedance and a resolution function can be declared with the **-pull** option. If there is no resolution function, the default configuration of the Xilinx FPGA Place and Route software is used.

Syntax

```
zice_clock fclkout -pin <clock_name> -con_loc <con_id>.<pin>
  -pull <resolution>
zice_clock fclkout -net <clock_name> -con_loc <con_id>.<pin>
  -pull <resolution>
```

where, **<resolution>** is the resolution function. Possible values are **keeper**, **pullup**, or **pulldown**.

For Example,

To declare a pullup tri-state resolution for local output clock **top.sig_clk** connected to pin **J2100_30**:

```
zice_clock fclkout -pin top.sig_clk -con_loc J2100.30 -pull pullup
```

16.5.6 Connecting I/O Signals

The data connections can be declared with the **zice_io** commands. The options that can be used with the **zice_io** commands are listed in [Table 51](#) and [Table 52](#).

Note

Some of the option settings described in this section can override corresponding default settings declared with the zice command.

16.5.6.1 Declaring Signal Connections for an External Target

To declare an input, output, or bi-directional connection between the Direct-ICE target system and an internal net of the DUT, use the following syntax:

```
zice_io input -net <net_name> -con_loc <con_id>.<pin>
zice_io output -net <net_name> -con_loc <con_id>.<pin>
zice_io inout -net <net_name> -con_loc <con_id>.<pin>
```

where, <net_name> is the hierarchical name of the wire to connect.

For Example,

To connect an internal net {dut.b_core2.di1[0]} as a Direct-ICE input to pin J1204.26:

```
zice_io input -net {dut.b_core2.di1[0]} -con_loc J1204.26
```

16.5.6.2 Declaring Signals When Connecting to a Port of the DUT

To declare an input, output, or bi-directional connection between the Direct-ICE target system and a port of the DUT (or in the DUT), use the following syntax:

```
zice_io input -pin <path_from_top>.<port_name> -con_loc
<con_id>.<pin>
zice_io output -pin <path_from_top>.<port_name> -con_loc
<con_id>.<pin>
zice_io inout -pin <path_from_top>.<port_name> -con_loc
<con_id>.<pin>
```

where,

- <path_from_top> is the hierarchical path to a pin in the DUT, where:
 - <path_from_top> = <top_name>.<instance_path>
 - <instance_path> = [instance_name.] *
- <port_name> is a port at the top of the design.

Examples

To connect top port top.DATA_ICE_IN to pin J1204.26:

```
zice_io input -pin top.DATA_ICE_IN -con_loc J1204.26
```

To connect internal port top.inst1.inst2.DATA_ICE_IN to pin J1204.27:

```
zice_io input -pin top.inst1.inst2.DATA_ICE_IN -con_loc J1204.27
```

16.5.7 Direct-ICE Declaring Signal Connections for an Internal Target

To declare a connection between the Direct-ICE target system and the port of an instance in the DUT, use the following syntax:

```
zice_io input -target_path <instance_path> -pin <port_name>
-con_loc <con_id>.<pin>
zice_io output -target_path <instance_path> -pin <port_name>
-con_loc <con_id>.<pin>
zice_io inout -target_path <instance_path> -pin <port_name>
-con_loc <con_id>.<pin>
```

where,

- <instance_path> is the hierarchical path to the blackbox module, which represents the internal target.
- <port_name> is the name of a port at the target interface, that is, a port at the interface of the instance specified by -target_path (internal target path).

For Example,

To connect I and J ports of an internal target whose hierarchical path is dut.zice_target to pins J1204.26 and J1204.27:

```
zice_io input -target_path dut.zice_target -pin I -con_loc J1204.26
zice_io inout -target_path dut.zice_target -pin J -con_loc J1204.27
```

For more information, see [Connecting an Internal Target to the DUT](#).

16.5.7.1 Forcing Data Pins

You can force a data pin of the Direct-ICE interface to 0 (using the `zice_io gnd` command) or to 1 (using the `zice_io vcc` command). When you force a data pin to 1, it is forced to the same voltage as the one of the bank to which the declared connector belongs:

```
zice_io gnd -con_loc <con_id>.<pin>
zice_io vcc -con_loc <con_id>.<pin>
```

Examples

To force pin J1204.26 to ground:

```
zice_io gnd -con_loc J1204.26
```

To force pin J1204.27 to the voltage of the bank to which connector J1204 belongs:

```
zice_io vcc -con_loc J1204.27
```

16.5.7.2 Interconnecting Pins

You can interconnect several pins of the Direct-ICE interface if they are assumed to be connected to the same FPGA. A group of interconnected pins is called a loop and several loops can be defined in your Direct-ICE compilation script using the `-loop <loop_id>` option.

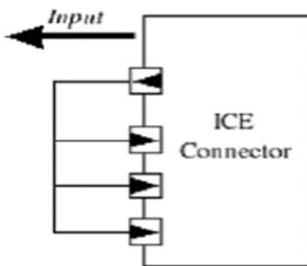
Writing the Direct-ICE Compilation Script


FIGURE 98. Interconnecting Direct-ICE Target Pins

Each loop has one driver from the Direct-ICE target, and connected to one or more sinks (Direct-ICE target inputs). However, the declaration of the interconnected pins is *ZeBu-centric* although there is no connection to the DUT.

You can invert a sink of the loop using the [-inv] option in the command line according to your project requirement.

```
zice_io input -loop <loop_id> -con_loc <con_id>.<pin>
zice_io output -loop <loop_id> -con_loc <con_id>.<pin>
```

where, <loop_id> is the name of a group of interconnected pins.

For Example,

To redirect the signal coming through pin J1204.26 to outgoing pin J1202.31:

```
zice_io input -loop loop1 -con_loc J1204.26
zice_io output -loop loop1 -con_loc J1202.31
```

16.5.7.3 Delayed I/O Signals

Individual data signals can be delayed using a declared delay or delay step.

```
zice_io input -con_loc <con_id>.<pin> [-net <net_name>|-pin
<top_name>.

<port_name>|-target_path <instance_path> -pin <port_name>]
-delay <delay_value> -delay_step <step_value>

zice_io output -con_loc <con_id>.<pin> [-net <net_name>|-pin
<top_name>.

<port_name>|-target_path <instance_path> -pin <port_name>|-dve_source
<sig_name>] -delay <delay_value> -delay_step <step_value>

zice_io inout -con_loc <con_id>.<pin> [-net <net_name>|-pin
<top_name>.

<port_name>|-target_path <instance_path> -pin <port_name>]
-delay <delay_value> -delay_step <step_value>

zice_io inout -con_loc <con_id>.<pin> [-net <net_name>|-pin
<top_name>.

<port_name>|-target_path <instance_path> -pin <port_name>]
-delay_en <delay_value> -delay_step <step_value>

zice_io inout -con_loc <con_id>.<pin> [-net <net_name>|-pin
<top_name>.

<port_name>|-target_path <instance_path> -pin <port_name>]
-delay_data <delay_value> -delay_step <step_value>
```

where,

- <delay_value> is a delay in nanoseconds (maximum: 640 ns).
- <step_value> is a delay step of 10 ns, 20 ns, or 40 ns.

Note

If the delay is not compatible with the delay step, the delay step is assumed to be correct and the delay is adjusted accordingly.

Examples

To define the default delay and delay step:

```
zice default -delay 40ns -delay_step 20ns
```

To override the default delay and delay step:

```
zice_io output -con_loc J1204.27 -net dut.counter.cnt\[0\]  
-delay 80ns -delay_step 40ns
```

16.5.7.4 Compilation Constraints on Data Output Signals

For data output signals, you can force some compilation constraints (similar to Xilinx ISE constraints).

These settings override the corresponding default setting described in [Default Delay](#).

```
zice_io output -con_loc <con_id>.<pin> [-net <net_name>|-pin  
<top_name>.  
    <port_name>|-target_path <instance_path> -pin <port_name>] |-  
dve_source  
    <sig_name>] -slew <slew_mode> -drive <drive_value>  
zice_io inout -con_loc <con_id>.<pin> [-net <net_name>|-pin  
<top_name>.  
    <port_name>|-target_path <instance_path> -pin <port_name>]  
    -slew <slew_mode> -drive <drive_value>
```

where,

- <slew_mode> can be FAST (produces a faster output but increases noise and power consumption) or SLOW. By default, <slew_mode> is declared as FAST.
- <drive_value> selects the output drive strength in mA. The possible values are: 2, 4, 6, 8, 12, 16, or 24. Ensure that you use high –drive values with care to limit the overall current consumption and possible overheating of the Direct-ICE FPGAs. When the default value is not declared, a value of 4 is used. For the appropriate output drive strength, see the Xilinx Virtex-7 FPGA documentation.

For Example,

To declare slew and drive constraints SLOW and eight for a signal (identified in the DVE file as `rst_value`) and connected to pin J1204.30:

```
zice_io output -dve_source rst_value -con_loc J1204.30 -slew SLOW -  
drive 8
```

16.5.7.5 Tri-State Resolution on I/O Signals

Tri-State Resolution on Data Output Signals

When the shielding is active, the output signals are in high impedance and a resolution function can be declared with the `-pull` option. Without a resolution, the default configuration of the Xilinx FPGA Place and Route software is used.

```
zice_io output -con_loc <con_id>.<pin> [-net <net_name>|-pin  
<top_name>.  
 <port_name>|-target_path <instance_path> -pin <port_name>] |-  
dve_source  
 <sig_name>] -pull <resolution>
```

where, `<resolution>` is the resolution function. The possible values are `keeper`, `pullup`, or `pulldown`.

For Example,

To declare a pullup tri-state resolution on signal `top.fromIce`:

```
zice_io output -con_loc J1204.26 -net top.fromIce -pull pullup
```

Tri-State Resolution on Bi-directional Signals

Bi-directional signals are in high impedance in the following cases:

- Shielding is active.
- Shielding is not active but the design and Direct-ICE are not driving.

In both the cases, a resolution function can be declared with the `-pull` option. If there is no resolution function, the default configuration of the Xilinx FPGA Place and

Writing the Direct-ICE Compilation Script

Route software is used.

```
zice_io inout -con_loc <con_id>.<pin> [-net <net_name>|-pin  
<top_name>.  
    <port_name>|-target_path <instance_path> -pin <port_name>]  
-pull <resolution>
```

where, <resolution> is the resolution function. The possible values are keeper, pullup, or pulldown.

For Example,

To declare a pullup tri-state resolution on signal top.toIce:

```
zice_io inout -con_loc J1204.27 -net top.toIce -pull pullup
```

16.6 Compiling for Direct-ICE

To compile for Direct-ICE, add the description script in the UTF:

```
ztopbuild -advanced_command {set_zice_target {<path>/dice.tcl}  
UNIT0.MODO {ERNI_DIRECT}}
```

The commands available in the dice.tcl are listed as follows:

```
zice voltage -vcc_bank VCCIO_2 -value 1.8V  
zice voltage -vcc_bank VCCIO_1 -value 1.8V  
zice_io output -pin {hw_top.S0.pattern[0]} -con_loc J1213.26  
zice_io input -pin {hw_top.R0.pattern[0]} -con_loc J2121.26  
zice_io output -pin {hw_top.S0.pattern[1]} -con_loc J1213.27  
zice_io input -pin {hw_top.R0.pattern[1]} -con_loc J2121.27  
zice_io output -pin {hw_top.S0.pattern[2]} -con_loc J1213.28  
zice_io input -pin {hw_top.R0.pattern[2]} -con_loc J2121.28  
zice_io output -pin {hw_top.S0.pattern[3]} -con_loc J1213.29  
zice_io input -pin {hw_top.R0.pattern[3]} -con_loc J2121.29  
zice_io output -pin {hw_top.S0.pattern[4]} -con_loc J1213.30  
zice_io input -pin {hw_top.R0.pattern[4]} -con_loc J2121.30  
zice_io output -pin {hw_top.S0.pattern[5]} -con_loc J1213.31  
zice_io input -pin {hw_top.R0.pattern[5]} -con_loc J2121.31  
zice_io output -pin {hw_top.S0.pattern[6]} -con_loc J1213.32  
zice_io input -pin {hw_top.R0.pattern[6]} -con_loc J2121.32  
zice_io output -pin {hw_top.S0.pattern[7]} -con_loc J1213.33  
zice_io input -pin {hw_top.R0.pattern[7]} -con_loc J2121.33  
zice_io output -pin {hw_top.S0.pattern[8]} -con_loc J1213.34  
zice_io input -pin {hw_top.R0.pattern[8]} -con_loc J2121.34  
zice_io output -pin {hw_top.S0.pattern[9]} -con_loc J1213.35  
zice_io input -pin {hw_top.R0.pattern[9]} -con_loc J2121.35
```

Compiling for Direct-ICE

```
zice_io output -pin {hw_top.S0.pattern[10]} -con_loc J1213.36
zice_io input -pin {hw_top.R0.pattern[10]} -con_loc J2121.36
zice_io output -pin {hw_top.S0.pattern[11]} -con_loc J1213.37
zice_io input -pin {hw_top.R0.pattern[11]} -con_loc J2121.37
zice_io output -pin {hw_top.S0.pattern[12]} -con_loc J1213.38
zice_io input -pin {hw_top.R0.pattern[12]} -con_loc J2121.38
zice_io output -pin {hw_top.S0.pattern[13]} -con_loc J1213.39
zice_io input -pin {hw_top.R0.pattern[13]} -con_loc J2121.39
zice_io output -pin {hw_top.S0.pattern[14]} -con_loc J1213.40
zice_io input -pin {hw_top.R0.pattern[14]} -con_loc J2121.40
zice_io output -pin {hw_top.S0.pattern[15]} -con_loc J1213.41
zice_io input -pin {hw_top.R0.pattern[15]} -con_loc J2121.41
zice_io output -pin {hw_top.S0.pattern[16]} -con_loc J1213.42
zice_io input -pin {hw_top.R0.pattern[16]} -con_loc J2121.42
zice_io output -pin {hw_top.S0.pattern[17]} -con_loc J1213.43
zice_io input -pin {hw_top.R0.pattern[17]} -con_loc J2121.43
zice_io output -pin {hw_top.S0.pattern[18]} -con_loc J1213.44
zice_io input -pin {hw_top.R0.pattern[18]} -con_loc J2121.44
zice_io output -pin {hw_top.S0.pattern[19]} -con_loc J1213.45
zice_io input -pin {hw_top.R0.pattern[19]} -con_loc J2121.45
zice_io output -pin {hw_top.S0.pattern[20]} -con_loc J1213.46
zice_io input -pin {hw_top.R0.pattern[20]} -con_loc J2121.46
zice_io output -pin {hw_top.S0.pattern[21]} -con_loc J1213.47
zice_io input -pin {hw_top.R0.pattern[21]} -con_loc J2121.47
zice_io output -pin {hw_top.S0.pattern[22]} -con_loc J1213.48
zice_io input -pin {hw_top.R0.pattern[22]} -con_loc J2121.48
zice_io output -pin {hw_top.S0.pattern[23]} -con_loc J1213.49
zice_io input -pin {hw_top.R0.pattern[23]} -con_loc J2121.49
zice_io output -pin {hw_top.S0.pattern[24]} -con_loc J1213.50
```

```
zice_io input -pin {hw_top.R0.pattern[24]} -con_loc J2121.50
zice_io output -pin {hw_top.S1.pattern[0]} -con_loc J1232.26
zice_io input -pin {hw_top.R1.pattern[0]} -con_loc J2102.26
zice_io output -pin {hw_top.S1.pattern[1]} -con_loc J1232.27
zice_io input -pin {hw_top.R1.pattern[1]} -con_loc J2102.27
zice_io output -pin {hw_top.S1.pattern[2]} -con_loc J1232.28
zice_io input -pin {hw_top.R1.pattern[2]} -con_loc J2102.28
zice_io output -pin {hw_top.S1.pattern[3]} -con_loc J1232.29
zice_io input -pin {hw_top.R1.pattern[3]} -con_loc J2102.29
zice_io output -pin {hw_top.S1.pattern[4]} -con_loc J1232.30
zice_io input -pin {hw_top.R1.pattern[4]} -con_loc J2102.30
zice_io output -pin {hw_top.S1.pattern[5]} -con_loc J1232.31
zice_io input -pin {hw_top.R1.pattern[5]} -con_loc J2102.31
zice_io output -pin {hw_top.S1.pattern[6]} -con_loc J1232.32
zice_io input -pin {hw_top.R1.pattern[6]} -con_loc J2102.32
zice_io output -pin {hw_top.S1.pattern[7]} -con_loc J1232.33
zice_io input -pin {hw_top.R1.pattern[7]} -con_loc J2102.33
zice_io output -pin {hw_top.S1.pattern[8]} -con_loc J1232.34
zice_io input -pin {hw_top.R1.pattern[8]} -con_loc J2102.34
zice_io output -pin {hw_top.S1.pattern[9]} -con_loc J1232.35
zice_io input -pin {hw_top.R1.pattern[9]} -con_loc J2102.35
zice_io output -pin {hw_top.S1.pattern[10]} -con_loc J1232.36
zice_io input -pin {hw_top.R1.pattern[10]} -con_loc J2102.36
zice_io output -pin {hw_top.S1.pattern[11]} -con_loc J1232.37
zice_io input -pin {hw_top.R1.pattern[11]} -con_loc J2102.37
zice_io output -pin {hw_top.S1.pattern[12]} -con_loc J1232.38
zice_io input -pin {hw_top.R1.pattern[12]} -con_loc J2102.38
zice_io output -pin {hw_top.S1.pattern[13]} -con_loc J1232.39
zice_io input -pin {hw_top.R1.pattern[13]} -con_loc J2102.39
```

Compiling for Direct-ICE

```
zice_io output -pin {hw_top.S1.pattern[14]} -con_loc J1232.40
zice_io input -pin {hw_top.R1.pattern[14]} -con_loc J2102.40
zice_io output -pin {hw_top.S1.pattern[15]} -con_loc J1232.41
zice_io input -pin {hw_top.R1.pattern[15]} -con_loc J2102.41
zice_io output -pin {hw_top.S1.pattern[16]} -con_loc J1232.42
zice_io input -pin {hw_top.R1.pattern[16]} -con_loc J2102.42
zice_io output -pin {hw_top.S1.pattern[17]} -con_loc J1232.43
zice_io input -pin {hw_top.R1.pattern[17]} -con_loc J2102.43
zice_io output -pin {hw_top.S1.pattern[18]} -con_loc J1232.44
zice_io input -pin {hw_top.R1.pattern[18]} -con_loc J2102.44
zice_io output -pin {hw_top.S2.pattern[0]} -con_loc J1222.26
zice_io input -pin {hw_top.R2.pattern[0]} -con_loc J2112.26
zice_io output -pin {hw_top.S2.pattern[1]} -con_loc J1222.27
zice_io input -pin {hw_top.R2.pattern[1]} -con_loc J2112.27
zice_io output -pin {hw_top.S2.pattern[2]} -con_loc J1222.28
zice_io input -pin {hw_top.R2.pattern[2]} -con_loc J2112.28
zice_io output -pin {hw_top.S2.pattern[3]} -con_loc J1222.29
zice_io input -pin {hw_top.R2.pattern[3]} -con_loc J2112.29
zice_io output -pin {hw_top.S2.pattern[4]} -con_loc J1222.30
zice_io input -pin {hw_top.R2.pattern[4]} -con_loc J2112.30
zice_io output -pin {hw_top.S2.pattern[5]} -con_loc J1222.31
zice_io input -pin {hw_top.R2.pattern[5]} -con_loc J2112.31
zice_io output -pin {hw_top.S2.pattern[6]} -con_loc J1222.32
zice_io input -pin {hw_top.R2.pattern[6]} -con_loc J2112.32
zice_io output -pin {hw_top.S2.pattern[7]} -con_loc J1222.33
zice_io input -pin {hw_top.R2.pattern[7]} -con_loc J2112.33
zice_io output -pin {hw_top.S2.pattern[8]} -con_loc J1222.34
zice_io input -pin {hw_top.R2.pattern[8]} -con_loc J2112.34
zice_io output -pin {hw_top.S2.pattern[9]} -con_loc J1222.35
```

```
zice_io input -pin {hw_top.R2.pattern[9]} -con_loc J2112.35
zice_io output -pin {hw_top.S2.pattern[10]} -con_loc J1222.36
zice_io input -pin {hw_top.R2.pattern[10]} -con_loc J2112.36
zice_io output -pin {hw_top.S2.pattern[11]} -con_loc J1222.37
zice_io input -pin {hw_top.R2.pattern[11]} -con_loc J2112.37
zice_io output -pin {hw_top.S2.pattern[12]} -con_loc J1222.38
zice_io input -pin {hw_top.R2.pattern[12]} -con_loc J2112.38
zice_io output -pin {hw_top.S2.pattern[13]} -con_loc J1222.39
zice_io input -pin {hw_top.R2.pattern[13]} -con_loc J2112.39
zice_io output -pin {hw_top.S2.pattern[14]} -con_loc J1222.40
zice_io input -pin {hw_top.R2.pattern[14]} -con_loc J2112.40
zice_io output -pin {hw_top.S2.pattern[15]} -con_loc J1222.41
zice_io input -pin {hw_top.R2.pattern[15]} -con_loc J2112.41
zice_io output -pin {hw_top.S2.pattern[16]} -con_loc J1222.42
zice_io input -pin {hw_top.R2.pattern[16]} -con_loc J2112.42
zice_io output -pin {hw_top.S2.pattern[17]} -con_loc J1222.43
zice_io input -pin {hw_top.R2.pattern[17]} -con_loc J2112.43
zice_io output -pin {hw_top.S2.pattern[18]} -con_loc J1222.44
zice_io input -pin {hw_top.R2.pattern[18]} -con_loc J2112.44
zice_io output -pin {hw_top.S2.pattern[19]} -con_loc J1222.45
zice_io input -pin {hw_top.R2.pattern[19]} -con_loc J2112.45
zice_io output -pin {hw_top.S2.pattern[20]} -con_loc J1222.46
zice_io input -pin {hw_top.R2.pattern[20]} -con_loc J2112.46
zice_io output -pin {hw_top.S2.pattern[21]} -con_loc J1222.47
zice_io input -pin {hw_top.R2.pattern[21]} -con_loc J2112.47
zice_io output -pin {hw_top.S2.pattern[22]} -con_loc J1222.48
zice_io input -pin {hw_top.R2.pattern[22]} -con_loc J2112.48
zice_io output -pin {hw_top.S2.pattern[23]} -con_loc J1222.49
zice_io input -pin {hw_top.R2.pattern[23]} -con_loc J2112.49
```

Compiling for Direct-ICE

```
zice_io output -pin {hw_top.S2.pattern[24]} -con_loc J1222.50
zice_io input -pin {hw_top.R2.pattern[24]} -con_loc J2112.50
zice_io output -pin {hw_top.S3.pattern[0]} -con_loc J1202.26
zice_io input -pin {hw_top.R3.pattern[0]} -con_loc J2132.26
zice_io output -pin {hw_top.S3.pattern[1]} -con_loc J1202.27
zice_io input -pin {hw_top.R3.pattern[1]} -con_loc J2132.27
zice_io output -pin {hw_top.S3.pattern[2]} -con_loc J1202.28
zice_io input -pin {hw_top.R3.pattern[2]} -con_loc J2132.28
zice_io output -pin {hw_top.S3.pattern[3]} -con_loc J1202.29
zice_io input -pin {hw_top.R3.pattern[3]} -con_loc J2132.29
zice_io output -pin {hw_top.S3.pattern[4]} -con_loc J1202.30
zice_io input -pin {hw_top.R3.pattern[4]} -con_loc J2132.30
zice_io output -pin {hw_top.S3.pattern[5]} -con_loc J1202.31
zice_io input -pin {hw_top.R3.pattern[5]} -con_loc J2132.31
zice_io output -pin {hw_top.S3.pattern[6]} -con_loc J1202.32
zice_io input -pin {hw_top.R3.pattern[6]} -con_loc J2132.32
zice_io output -pin {hw_top.S3.pattern[7]} -con_loc J1202.33
zice_io input -pin {hw_top.R3.pattern[7]} -con_loc J2132.33
zice_io output -pin {hw_top.S3.pattern[8]} -con_loc J1202.34
zice_io input -pin {hw_top.R3.pattern[8]} -con_loc J2132.34
zice_io output -pin {hw_top.S3.pattern[9]} -con_loc J1202.35
zice_io input -pin {hw_top.R3.pattern[9]} -con_loc J2132.35
zice_io output -pin {hw_top.S3.pattern[10]} -con_loc J1202.36
zice_io input -pin {hw_top.R3.pattern[10]} -con_loc J2132.36
zice_io output -pin {hw_top.S3.pattern[11]} -con_loc J1202.37
zice_io input -pin {hw_top.R3.pattern[11]} -con_loc J2132.37
zice_io output -pin {hw_top.S3.pattern[12]} -con_loc J1202.38
zice_io input -pin {hw_top.R3.pattern[12]} -con_loc J2132.38
zice_io output -pin {hw_top.S3.pattern[13]} -con_loc J1202.39
```

```
zice_io input -pin {hw_top.R3.pattern[13]} -con_loc J2132.39
zice_io output -pin {hw_top.S3.pattern[14]} -con_loc J1202.40
zice_io input -pin {hw_top.R3.pattern[14]} -con_loc J2132.40
zice_io output -pin {hw_top.S3.pattern[15]} -con_loc J1202.41
zice_io input -pin {hw_top.R3.pattern[15]} -con_loc J2132.41
zice_io output -pin {hw_top.S3.pattern[16]} -con_loc J1202.42
zice_io input -pin {hw_top.R3.pattern[16]} -con_loc J2132.42
zice_io output -pin {hw_top.S3.pattern[17]} -con_loc J1202.43
zice_io input -pin {hw_top.R3.pattern[17]} -con_loc J2132.43
zice_io output -pin {hw_top.S3.pattern[18]} -con_loc J1202.44
zice_io input -pin {hw_top.R3.pattern[18]} -con_loc J2132.44
zice_io output -pin {hw_top.S3.pattern[19]} -con_loc J1202.45
zice_io input -pin {hw_top.R3.pattern[19]} -con_loc J2132.45
zice_io output -pin {hw_top.S3.pattern[20]} -con_loc J1202.46
zice_io input -pin {hw_top.R3.pattern[20]} -con_loc J2132.46
zice_io output -pin {hw_top.S3.pattern[21]} -con_loc J1202.47
zice_io input -pin {hw_top.R3.pattern[21]} -con_loc J2132.47
zice_io output -pin {hw_top.S3.pattern[22]} -con_loc J1202.48
zice_io input -pin {hw_top.R3.pattern[22]} -con_loc J2132.48
zice_io output -pin {hw_top.S3.pattern[23]} -con_loc J1202.49
zice_io input -pin {hw_top.R3.pattern[23]} -con_loc J2132.49
zice_io output -pin {hw_top.S3.pattern[24]} -con_loc J1202.50
zice_io input -pin {hw_top.R3.pattern[24]} -con_loc J2132.50
zice_io output -pin {hw_top.S4.pattern[0]} -con_loc J1212.26
zice_io input -pin {hw_top.R4.pattern[0]} -con_loc J2122.26
zice_io output -pin {hw_top.S4.pattern[1]} -con_loc J1212.27
zice_io input -pin {hw_top.R4.pattern[1]} -con_loc J2122.27
zice_io output -pin {hw_top.S4.pattern[2]} -con_loc J1212.28
zice_io input -pin {hw_top.R4.pattern[2]} -con_loc J2122.28
```

Compiling for Direct-ICE

```
zice_io output -pin {hw_top.S4.pattern[3]} -con_loc J1212.29
zice_io input -pin {hw_top.R4.pattern[3]} -con_loc J2122.29
zice_io output -pin {hw_top.S4.pattern[4]} -con_loc J1212.30
zice_io input -pin {hw_top.R4.pattern[4]} -con_loc J2122.30
zice_io output -pin {hw_top.S4.pattern[5]} -con_loc J1212.31
zice_io input -pin {hw_top.R4.pattern[5]} -con_loc J2122.31
zice_io output -pin {hw_top.S4.pattern[6]} -con_loc J1212.32
zice_io input -pin {hw_top.R4.pattern[6]} -con_loc J2122.32
zice_io output -pin {hw_top.S4.pattern[7]} -con_loc J1212.33
zice_io input -pin {hw_top.R4.pattern[7]} -con_loc J2122.33
zice_io output -pin {hw_top.S4.pattern[8]} -con_loc J1212.34
zice_io input -pin {hw_top.R4.pattern[8]} -con_loc J2122.34
zice_io output -pin {hw_top.S4.pattern[9]} -con_loc J1212.35
zice_io input -pin {hw_top.R4.pattern[9]} -con_loc J2122.35
zice_io output -pin {hw_top.S4.pattern[10]} -con_loc J1212.36
zice_io input -pin {hw_top.R4.pattern[10]} -con_loc J2122.36
zice_io output -pin {hw_top.S4.pattern[11]} -con_loc J1212.37
zice_io input -pin {hw_top.R4.pattern[11]} -con_loc J2122.37
zice_io output -pin {hw_top.S4.pattern[12]} -con_loc J1212.38
zice_io input -pin {hw_top.R4.pattern[12]} -con_loc J2122.38
zice_io output -pin {hw_top.S4.pattern[13]} -con_loc J1212.39
zice_io input -pin {hw_top.R4.pattern[13]} -con_loc J2122.39
zice_io output -pin {hw_top.S4.pattern[14]} -con_loc J1212.40
zice_io input -pin {hw_top.R4.pattern[14]} -con_loc J2122.40
zice_io output -pin {hw_top.S4.pattern[15]} -con_loc J1212.41
zice_io input -pin {hw_top.R4.pattern[15]} -con_loc J2122.41
zice_io output -pin {hw_top.S4.pattern[16]} -con_loc J1212.42
zice_io input -pin {hw_top.R4.pattern[16]} -con_loc J2122.42
zice_io output -pin {hw_top.S4.pattern[17]} -con_loc J1212.43
```

```
zice_io input -pin {hw_top.R4.pattern[17]} -con_loc J2122.43
zice_io output -pin {hw_top.S4.pattern[18]} -con_loc J1212.44
zice_io input -pin {hw_top.R4.pattern[18]} -con_loc J2122.44
zice_io output -pin {hw_top.S4.pattern[19]} -con_loc J1212.45
zice_io input -pin {hw_top.R4.pattern[19]} -con_loc J2122.45
zice_io output -pin {hw_top.S4.pattern[20]} -con_loc J1212.46
zice_io input -pin {hw_top.R4.pattern[20]} -con_loc J2122.46
zice_io output -pin {hw_top.S4.pattern[21]} -con_loc J1212.47
zice_io input -pin {hw_top.R4.pattern[21]} -con_loc J2122.47
zice_io output -pin {hw_top.S4.pattern[22]} -con_loc J1212.48
zice_io input -pin {hw_top.R4.pattern[22]} -con_loc J2122.48
zice_io output -pin {hw_top.S4.pattern[23]} -con_loc J1212.49
zice_io input -pin {hw_top.R4.pattern[23]} -con_loc J2122.49
zice_io output -pin {hw_top.S4.pattern[24]} -con_loc J1212.50
zice_io input -pin {hw_top.R4.pattern[24]} -con_loc J2122.50
zice_io output -pin {hw_top.S5.pattern[0]} -con_loc J1233.26
zice_io input -pin {hw_top.R5.pattern[0]} -con_loc J2101.26
zice_io output -pin {hw_top.S5.pattern[1]} -con_loc J1233.27
zice_io input -pin {hw_top.R5.pattern[1]} -con_loc J2101.27
zice_io output -pin {hw_top.S5.pattern[2]} -con_loc J1233.28
zice_io input -pin {hw_top.R5.pattern[2]} -con_loc J2101.28
zice_io output -pin {hw_top.S5.pattern[3]} -con_loc J1233.29
zice_io input -pin {hw_top.R5.pattern[3]} -con_loc J2101.29
zice_io output -pin {hw_top.S5.pattern[4]} -con_loc J1233.30
zice_io input -pin {hw_top.R5.pattern[4]} -con_loc J2101.30
zice_io output -pin {hw_top.S5.pattern[5]} -con_loc J1233.31
zice_io input -pin {hw_top.R5.pattern[5]} -con_loc J2101.31
zice_io output -pin {hw_top.S5.pattern[6]} -con_loc J1233.32
zice_io input -pin {hw_top.R5.pattern[6]} -con_loc J2101.32
```

Compiling for Direct-ICE

```
zice_io output -pin {hw_top.S5.pattern[7]} -con_loc J1233.33
zice_io input -pin {hw_top.R5.pattern[7]} -con_loc J2101.33
zice_io output -pin {hw_top.S5.pattern[8]} -con_loc J1233.34
zice_io input -pin {hw_top.R5.pattern[8]} -con_loc J2101.34
zice_io output -pin {hw_top.S5.pattern[9]} -con_loc J1233.35
zice_io input -pin {hw_top.R5.pattern[9]} -con_loc J2101.35
zice_io output -pin {hw_top.S5.pattern[10]} -con_loc J1233.36
zice_io input -pin {hw_top.R5.pattern[10]} -con_loc J2101.36
zice_io output -pin {hw_top.S5.pattern[11]} -con_loc J1233.37
zice_io input -pin {hw_top.R5.pattern[11]} -con_loc J2101.37
zice_io output -pin {hw_top.S5.pattern[12]} -con_loc J1233.38
zice_io input -pin {hw_top.R5.pattern[12]} -con_loc J2101.38
zice_io output -pin {hw_top.S5.pattern[13]} -con_loc J1233.39
zice_io input -pin {hw_top.R5.pattern[13]} -con_loc J2101.39
zice_io output -pin {hw_top.S5.pattern[14]} -con_loc J1233.40
zice_io input -pin {hw_top.R5.pattern[14]} -con_loc J2101.40
zice_io output -pin {hw_top.S5.pattern[15]} -con_loc J1233.41
zice_io input -pin {hw_top.R5.pattern[15]} -con_loc J2101.41
zice_io output -pin {hw_top.S5.pattern[16]} -con_loc J1233.42
zice_io input -pin {hw_top.R5.pattern[16]} -con_loc J2101.42
zice_io output -pin {hw_top.S5.pattern[17]} -con_loc J1233.43
zice_io input -pin {hw_top.R5.pattern[17]} -con_loc J2101.43
zice_io output -pin {hw_top.S5.pattern[18]} -con_loc J1233.44
zice_io input -pin {hw_top.R5.pattern[18]} -con_loc J2101.44
zice_io output -pin {hw_top.S5.pattern[19]} -con_loc J1233.45
zice_io input -pin {hw_top.R5.pattern[19]} -con_loc J2101.45
zice_io output -pin {hw_top.S5.pattern[20]} -con_loc J1233.46
zice_io input -pin {hw_top.R5.pattern[20]} -con_loc J2101.46
zice_io output -pin {hw_top.S5.pattern[21]} -con_loc J1233.47
```

```
zice_io input -pin {hw_top.R5.pattern[21]} -con_loc J2101.47
zice_io output -pin {hw_top.S5.pattern[22]} -con_loc J1233.48
zice_io input -pin {hw_top.R5.pattern[22]} -con_loc J2101.48
zice_io output -pin {hw_top.S5.pattern[23]} -con_loc J1233.49
zice_io input -pin {hw_top.R5.pattern[23]} -con_loc J2101.49
zice_io output -pin {hw_top.S5.pattern[24]} -con_loc J1233.50
zice_io input -pin {hw_top.R5.pattern[24]} -con_loc J2101.50
zice_io output -pin {hw_top.S6.pattern[0]} -con_loc J1223.26
zice_io input -pin {hw_top.R6.pattern[0]} -con_loc J2111.26
zice_io output -pin {hw_top.S6.pattern[1]} -con_loc J1223.27
zice_io input -pin {hw_top.R6.pattern[1]} -con_loc J2111.27
zice_io output -pin {hw_top.S6.pattern[2]} -con_loc J1223.28
zice_io input -pin {hw_top.R6.pattern[2]} -con_loc J2111.28
zice_io output -pin {hw_top.S6.pattern[3]} -con_loc J1223.29
zice_io input -pin {hw_top.R6.pattern[3]} -con_loc J2111.29
zice_io output -pin {hw_top.S6.pattern[4]} -con_loc J1223.30
zice_io input -pin {hw_top.R6.pattern[4]} -con_loc J2111.30
zice_io output -pin {hw_top.S6.pattern[5]} -con_loc J1223.31
zice_io input -pin {hw_top.R6.pattern[5]} -con_loc J2111.31
zice_io output -pin {hw_top.S6.pattern[6]} -con_loc J1223.32
zice_io input -pin {hw_top.R6.pattern[6]} -con_loc J2111.32
zice_io output -pin {hw_top.S6.pattern[7]} -con_loc J1223.33
zice_io input -pin {hw_top.R6.pattern[7]} -con_loc J2111.33
zice_io output -pin {hw_top.S6.pattern[8]} -con_loc J1223.34
zice_io input -pin {hw_top.R6.pattern[8]} -con_loc J2111.34
zice_io output -pin {hw_top.S6.pattern[9]} -con_loc J1223.35
zice_io input -pin {hw_top.R6.pattern[9]} -con_loc J2111.35
zice_io output -pin {hw_top.S6.pattern[10]} -con_loc J1223.36
zice_io input -pin {hw_top.R6.pattern[10]} -con_loc J2111.36
```

Compiling for Direct-ICE

```
zice_io output -pin {hw_top.S6.pattern[11]} -con_loc J1223.37
zice_io input -pin {hw_top.R6.pattern[11]} -con_loc J2111.37
zice_io output -pin {hw_top.S6.pattern[12]} -con_loc J1223.38
zice_io input -pin {hw_top.R6.pattern[12]} -con_loc J2111.38
zice_io output -pin {hw_top.S6.pattern[13]} -con_loc J1223.39
zice_io input -pin {hw_top.R6.pattern[13]} -con_loc J2111.39
zice_io output -pin {hw_top.S6.pattern[14]} -con_loc J1223.40
zice_io input -pin {hw_top.R6.pattern[14]} -con_loc J2111.40
zice_io output -pin {hw_top.S6.pattern[15]} -con_loc J1223.41
zice_io input -pin {hw_top.R6.pattern[15]} -con_loc J2111.41
zice_io output -pin {hw_top.S6.pattern[16]} -con_loc J1223.42
zice_io input -pin {hw_top.R6.pattern[16]} -con_loc J2111.42
zice_io output -pin {hw_top.S6.pattern[17]} -con_loc J1223.43
zice_io input -pin {hw_top.R6.pattern[17]} -con_loc J2111.43
zice_io output -pin {hw_top.S6.pattern[18]} -con_loc J1223.44
zice_io input -pin {hw_top.R6.pattern[18]} -con_loc J2111.44
zice_io output -pin {hw_top.S6.pattern[19]} -con_loc J1223.45
zice_io input -pin {hw_top.R6.pattern[19]} -con_loc J2111.45
zice_io output -pin {hw_top.S6.pattern[20]} -con_loc J1223.46
zice_io input -pin {hw_top.R6.pattern[20]} -con_loc J2111.46
zice_io output -pin {hw_top.S6.pattern[21]} -con_loc J1223.47
zice_io input -pin {hw_top.R6.pattern[21]} -con_loc J2111.47
zice_io output -pin {hw_top.S6.pattern[22]} -con_loc J1223.48
zice_io input -pin {hw_top.R6.pattern[22]} -con_loc J2111.48
zice_io output -pin {hw_top.S6.pattern[23]} -con_loc J1223.49
zice_io input -pin {hw_top.R6.pattern[23]} -con_loc J2111.49
zice_io output -pin {hw_top.S6.pattern[24]} -con_loc J1223.50
zice_io input -pin {hw_top.R6.pattern[24]} -con_loc J2111.50
zice_io output -pin {hw_top.S7.pattern[0]} -con_loc J1204.26
```

```
zice_io input -pin {hw_top.R7.pattern[0]} -con_loc J2130.26
zice_io output -pin {hw_top.S7.pattern[1]} -con_loc J1204.27
zice_io input -pin {hw_top.R7.pattern[1]} -con_loc J2130.27
zice_io output -pin {hw_top.S7.pattern[2]} -con_loc J1204.28
zice_io input -pin {hw_top.R7.pattern[2]} -con_loc J2130.28
zice_io output -pin {hw_top.S7.pattern[3]} -con_loc J1204.29
zice_io input -pin {hw_top.R7.pattern[3]} -con_loc J2130.29
zice_io output -pin {hw_top.S7.pattern[4]} -con_loc J1204.30
zice_io input -pin {hw_top.R7.pattern[4]} -con_loc J2130.30
zice_io output -pin {hw_top.S7.pattern[5]} -con_loc J1204.31
zice_io input -pin {hw_top.R7.pattern[5]} -con_loc J2130.31
zice_io output -pin {hw_top.S7.pattern[6]} -con_loc J1204.32
zice_io input -pin {hw_top.R7.pattern[6]} -con_loc J2130.32
zice_io output -pin {hw_top.S7.pattern[7]} -con_loc J1204.33
zice_io input -pin {hw_top.R7.pattern[7]} -con_loc J2130.33
zice_io output -pin {hw_top.S7.pattern[8]} -con_loc J1204.34
zice_io input -pin {hw_top.R7.pattern[8]} -con_loc J2130.34
zice_io output -pin {hw_top.S7.pattern[9]} -con_loc J1204.35
zice_io input -pin {hw_top.R7.pattern[9]} -con_loc J2130.35
zice_io output -pin {hw_top.S7.pattern[10]} -con_loc J1204.36
zice_io input -pin {hw_top.R7.pattern[10]} -con_loc J2130.36
zice_io output -pin {hw_top.S7.pattern[11]} -con_loc J1204.37
zice_io input -pin {hw_top.R7.pattern[11]} -con_loc J2130.37
zice_io output -pin {hw_top.S7.pattern[12]} -con_loc J1204.38
zice_io input -pin {hw_top.R7.pattern[12]} -con_loc J2130.38
zice_io output -pin {hw_top.S7.pattern[13]} -con_loc J1204.39
zice_io input -pin {hw_top.R7.pattern[13]} -con_loc J2130.39
zice_io output -pin {hw_top.S7.pattern[14]} -con_loc J1204.40
zice_io input -pin {hw_top.R7.pattern[14]} -con_loc J2130.40
```

Compiling for Direct-ICE

```
zice_io output -pin {hw_top.S7.pattern[15]} -con_loc J1204.41
zice_io input -pin {hw_top.R7.pattern[15]} -con_loc J2130.41
zice_io output -pin {hw_top.S7.pattern[16]} -con_loc J1204.42
zice_io input -pin {hw_top.R7.pattern[16]} -con_loc J2130.42
zice_io output -pin {hw_top.S7.pattern[17]} -con_loc J1204.43
zice_io input -pin {hw_top.R7.pattern[17]} -con_loc J2130.43
zice_io output -pin {hw_top.S7.pattern[18]} -con_loc J1204.44
zice_io input -pin {hw_top.R7.pattern[18]} -con_loc J2130.44
zice_io output -pin {hw_top.S7.pattern[19]} -con_loc J1204.45
zice_io input -pin {hw_top.R7.pattern[19]} -con_loc J2130.45
zice_io output -pin {hw_top.S7.pattern[20]} -con_loc J1204.46
zice_io input -pin {hw_top.R7.pattern[20]} -con_loc J2130.46
zice_io output -pin {hw_top.S7.pattern[21]} -con_loc J1204.47
zice_io input -pin {hw_top.R7.pattern[21]} -con_loc J2130.47
zice_io output -pin {hw_top.S7.pattern[22]} -con_loc J1204.48
zice_io input -pin {hw_top.R7.pattern[22]} -con_loc J2130.48
zice_io output -pin {hw_top.S7.pattern[23]} -con_loc J1204.49
zice_io input -pin {hw_top.R7.pattern[23]} -con_loc J2130.49
zice_io output -pin {hw_top.S7.pattern[24]} -con_loc J1204.50
zice_io input -pin {hw_top.R7.pattern[24]} -con_loc J2130.50
zice_io output -pin {hw_top.S8.pattern[0]} -con_loc J1224.26
zice_io input -pin {hw_top.R8.pattern[0]} -con_loc J2110.26
zice_io output -pin {hw_top.S8.pattern[1]} -con_loc J1224.27
zice_io input -pin {hw_top.R8.pattern[1]} -con_loc J2110.27
zice_io output -pin {hw_top.S8.pattern[2]} -con_loc J1224.28
zice_io input -pin {hw_top.R8.pattern[2]} -con_loc J2110.28
zice_io output -pin {hw_top.S8.pattern[3]} -con_loc J1224.29
zice_io input -pin {hw_top.R8.pattern[3]} -con_loc J2110.29
zice_io output -pin {hw_top.S8.pattern[4]} -con_loc J1224.30
```

```
zice_io input -pin {hw_top.R8.pattern[4]} -con_loc J2110.30
zice_io output -pin {hw_top.S8.pattern[5]} -con_loc J1224.31
zice_io input -pin {hw_top.R8.pattern[5]} -con_loc J2110.31
zice_io output -pin {hw_top.S8.pattern[6]} -con_loc J1224.32
zice_io input -pin {hw_top.R8.pattern[6]} -con_loc J2110.32
zice_io output -pin {hw_top.S8.pattern[7]} -con_loc J1224.33
zice_io input -pin {hw_top.R8.pattern[7]} -con_loc J2110.33
zice_io output -pin {hw_top.S8.pattern[8]} -con_loc J1224.34
zice_io input -pin {hw_top.R8.pattern[8]} -con_loc J2110.34
zice_io output -pin {hw_top.S8.pattern[9]} -con_loc J1224.35
zice_io input -pin {hw_top.R8.pattern[9]} -con_loc J2110.35
zice_io output -pin {hw_top.S8.pattern[10]} -con_loc J1224.36
zice_io input -pin {hw_top.R8.pattern[10]} -con_loc J2110.36
zice_io output -pin {hw_top.S8.pattern[11]} -con_loc J1224.37
zice_io input -pin {hw_top.R8.pattern[11]} -con_loc J2110.37
zice_io output -pin {hw_top.S8.pattern[12]} -con_loc J1224.38
zice_io input -pin {hw_top.R8.pattern[12]} -con_loc J2110.38
zice_io output -pin {hw_top.S8.pattern[13]} -con_loc J1224.39
zice_io input -pin {hw_top.R8.pattern[13]} -con_loc J2110.39
zice_io output -pin {hw_top.S8.pattern[14]} -con_loc J1224.40
zice_io input -pin {hw_top.R8.pattern[14]} -con_loc J2110.40
zice_io output -pin {hw_top.S8.pattern[15]} -con_loc J1224.41
zice_io input -pin {hw_top.R8.pattern[15]} -con_loc J2110.41
zice_io output -pin {hw_top.S8.pattern[16]} -con_loc J1224.42
zice_io input -pin {hw_top.R8.pattern[16]} -con_loc J2110.42
zice_io output -pin {hw_top.S8.pattern[17]} -con_loc J1224.43
zice_io input -pin {hw_top.R8.pattern[17]} -con_loc J2110.43
zice_io output -pin {hw_top.S8.pattern[18]} -con_loc J1224.44
zice_io input -pin {hw_top.R8.pattern[18]} -con_loc J2110.44
```

Compiling for Direct-ICE

```
zice_io output -pin {hw_top.S8.pattern[19]} -con_loc J1224.45
zice_io input -pin {hw_top.R8.pattern[19]} -con_loc J2110.45
zice_io output -pin {hw_top.S8.pattern[20]} -con_loc J1224.46
zice_io input -pin {hw_top.R8.pattern[20]} -con_loc J2110.46
zice_io output -pin {hw_top.S8.pattern[21]} -con_loc J1224.47
zice_io input -pin {hw_top.R8.pattern[21]} -con_loc J2110.47
zice_io output -pin {hw_top.S8.pattern[22]} -con_loc J1224.48
zice_io input -pin {hw_top.R8.pattern[22]} -con_loc J2110.48
zice_io output -pin {hw_top.S8.pattern[23]} -con_loc J1224.49
zice_io input -pin {hw_top.R8.pattern[23]} -con_loc J2110.49
zice_io output -pin {hw_top.S8.pattern[24]} -con_loc J1224.50
zice_io input -pin {hw_top.R8.pattern[24]} -con_loc J2110.50
zice_io output -pin {hw_top.S9.pattern[0]} -con_loc J1234.26
zice_io input -pin {hw_top.R9.pattern[0]} -con_loc J2100.26
zice_io output -pin {hw_top.S9.pattern[1]} -con_loc J1234.27
zice_io input -pin {hw_top.R9.pattern[1]} -con_loc J2100.27
zice_io output -pin {hw_top.S9.pattern[2]} -con_loc J1234.28
zice_io input -pin {hw_top.R9.pattern[2]} -con_loc J2100.28
zice_io output -pin {hw_top.S9.pattern[3]} -con_loc J1234.29
zice_io input -pin {hw_top.R9.pattern[3]} -con_loc J2100.29
zice_io output -pin {hw_top.S9.pattern[4]} -con_loc J1234.30
zice_io input -pin {hw_top.R9.pattern[4]} -con_loc J2100.30
zice_io output -pin {hw_top.S9.pattern[5]} -con_loc J1234.31
zice_io input -pin {hw_top.R9.pattern[5]} -con_loc J2100.31
zice_io output -pin {hw_top.S9.pattern[6]} -con_loc J1234.32
zice_io input -pin {hw_top.R9.pattern[6]} -con_loc J2100.32
zice_io output -pin {hw_top.S9.pattern[7]} -con_loc J1234.33
zice_io input -pin {hw_top.R9.pattern[7]} -con_loc J2100.33
zice_io output -pin {hw_top.S9.pattern[8]} -con_loc J1234.34
```

```
zice_io input -pin {hw_top.R9.pattern[8]} -con_loc J2100.34
zice_io output -pin {hw_top.S9.pattern[9]} -con_loc J1234.35
zice_io input -pin {hw_top.R9.pattern[9]} -con_loc J2100.35
zice_io output -pin {hw_top.S9.pattern[10]} -con_loc J1234.36
zice_io input -pin {hw_top.R9.pattern[10]} -con_loc J2100.36
zice_io output -pin {hw_top.S9.pattern[11]} -con_loc J1234.37
zice_io input -pin {hw_top.R9.pattern[11]} -con_loc J2100.37
zice_io output -pin {hw_top.S9.pattern[12]} -con_loc J1234.38
zice_io input -pin {hw_top.R9.pattern[12]} -con_loc J2100.38
zice_io output -pin {hw_top.S9.pattern[13]} -con_loc J1234.39
zice_io input -pin {hw_top.R9.pattern[13]} -con_loc J2100.39
zice_io output -pin {hw_top.S9.pattern[14]} -con_loc J1234.40
zice_io input -pin {hw_top.R9.pattern[14]} -con_loc J2100.40
zice_io output -pin {hw_top.S9.pattern[15]} -con_loc J1234.41
zice_io input -pin {hw_top.R9.pattern[15]} -con_loc J2100.41
zice_io output -pin {hw_top.S9.pattern[16]} -con_loc J1234.42
zice_io input -pin {hw_top.R9.pattern[16]} -con_loc J2100.42
zice_io output -pin {hw_top.S9.pattern[17]} -con_loc J1234.43
zice_io input -pin {hw_top.R9.pattern[17]} -con_loc J2100.43
zice_io output -pin {hw_top.S9.pattern[18]} -con_loc J1234.44
zice_io input -pin {hw_top.R9.pattern[18]} -con_loc J2100.44
zice_io output -pin {hw_top.S9.pattern[19]} -con_loc J1234.45
zice_io input -pin {hw_top.R9.pattern[19]} -con_loc J2100.45
zice_io output -pin {hw_top.S9.pattern[20]} -con_loc J1234.46
zice_io input -pin {hw_top.R9.pattern[20]} -con_loc J2100.46
zice_io output -pin {hw_top.S9.pattern[21]} -con_loc J1234.47
zice_io input -pin {hw_top.R9.pattern[21]} -con_loc J2100.47
zice_io output -pin {hw_top.S9.pattern[22]} -con_loc J1234.48
zice_io input -pin {hw_top.R9.pattern[22]} -con_loc J2100.48
```

Compiling for Direct-ICE

```
zice_io output -pin {hw_top.S9.pattern[23]} -con_loc J1234.49
zice_io input -pin {hw_top.R9.pattern[23]} -con_loc J2100.49
zice_io output -pin {hw_top.S9.pattern[24]} -con_loc J1234.50
zice_io input -pin {hw_top.R9.pattern[24]} -con_loc J2100.50
zice_io output -pin {hw_top.S10.pattern[0]} -con_loc J1203.26
zice_io input -pin {hw_top.R10.pattern[0]} -con_loc J2131.26
zice_io output -pin {hw_top.S10.pattern[1]} -con_loc J1203.27
zice_io input -pin {hw_top.R10.pattern[1]} -con_loc J2131.27
zice_io output -pin {hw_top.S10.pattern[2]} -con_loc J1203.28
zice_io input -pin {hw_top.R10.pattern[2]} -con_loc J2131.28
zice_io output -pin {hw_top.S10.pattern[3]} -con_loc J1203.29
zice_io input -pin {hw_top.R10.pattern[3]} -con_loc J2131.29
zice_io output -pin {hw_top.S10.pattern[4]} -con_loc J1203.30
zice_io input -pin {hw_top.R10.pattern[4]} -con_loc J2131.30
zice_io output -pin {hw_top.S10.pattern[5]} -con_loc J1203.31
zice_io input -pin {hw_top.R10.pattern[5]} -con_loc J2131.31
zice_io output -pin {hw_top.S10.pattern[6]} -con_loc J1203.32
zice_io input -pin {hw_top.R10.pattern[6]} -con_loc J2131.32
zice_io output -pin {hw_top.S10.pattern[7]} -con_loc J1203.33
zice_io input -pin {hw_top.R10.pattern[7]} -con_loc J2131.33
zice_io output -pin {hw_top.S10.pattern[8]} -con_loc J1203.34
zice_io input -pin {hw_top.R10.pattern[8]} -con_loc J2131.34
zice_io output -pin {hw_top.S10.pattern[9]} -con_loc J1203.35
zice_io input -pin {hw_top.R10.pattern[9]} -con_loc J2131.35
zice_io output -pin {hw_top.S10.pattern[10]} -con_loc J1203.36
zice_io input -pin {hw_top.R10.pattern[10]} -con_loc J2131.36
zice_io output -pin {hw_top.S10.pattern[11]} -con_loc J1203.37
zice_io input -pin {hw_top.R10.pattern[11]} -con_loc J2131.37
zice_io output -pin {hw_top.S10.pattern[12]} -con_loc J1203.38
```

```
zice_io input -pin {hw_top.R10.pattern[12]} -con_loc J2131.38
zice_io output -pin {hw_top.S10.pattern[13]} -con_loc J1203.39
zice_io input -pin {hw_top.R10.pattern[13]} -con_loc J2131.39
zice_io output -pin {hw_top.S10.pattern[14]} -con_loc J1203.40
zice_io input -pin {hw_top.R10.pattern[14]} -con_loc J2131.40
zice_io output -pin {hw_top.S11.pattern[0]} -con_loc J1214.26
zice_io input -pin {hw_top.R11.pattern[0]} -con_loc J2120.26
zice_io output -pin {hw_top.S11.pattern[1]} -con_loc J1214.27
zice_io input -pin {hw_top.R11.pattern[1]} -con_loc J2120.27
zice_io output -pin {hw_top.S11.pattern[2]} -con_loc J1214.28
zice_io input -pin {hw_top.R11.pattern[2]} -con_loc J2120.28
zice_io output -pin {hw_top.S11.pattern[3]} -con_loc J1214.29
zice_io input -pin {hw_top.R11.pattern[3]} -con_loc J2120.29
zice_io output -pin {hw_top.S11.pattern[4]} -con_loc J1214.30
zice_io input -pin {hw_top.R11.pattern[4]} -con_loc J2120.30
zice_io output -pin {hw_top.S11.pattern[5]} -con_loc J1214.31
zice_io input -pin {hw_top.R11.pattern[5]} -con_loc J2120.31
zice_io output -pin {hw_top.S11.pattern[6]} -con_loc J1214.32
zice_io input -pin {hw_top.R11.pattern[6]} -con_loc J2120.32
zice_io output -pin {hw_top.S11.pattern[7]} -con_loc J1214.33
zice_io input -pin {hw_top.R11.pattern[7]} -con_loc J2120.33
zice_io output -pin {hw_top.S11.pattern[8]} -con_loc J1214.34
zice_io input -pin {hw_top.R11.pattern[8]} -con_loc J2120.34
zice_io output -pin {hw_top.S11.pattern[9]} -con_loc J1214.35
zice_io input -pin {hw_top.R11.pattern[9]} -con_loc J2120.35
zice_io output -pin {hw_top.S11.pattern[10]} -con_loc J1214.36
zice_io input -pin {hw_top.R11.pattern[10]} -con_loc J2120.36
zice_io output -pin {hw_top.S11.pattern[11]} -con_loc J1214.37
zice_io input -pin {hw_top.R11.pattern[11]} -con_loc J2120.37
```

Compiling for Direct-ICE

```
zice_io output -pin {hw_top.S11.pattern[12]} -con_loc J1214.38
zice_io input -pin {hw_top.R11.pattern[12]} -con_loc J2120.38
zice_io output -pin {hw_top.S11.pattern[13]} -con_loc J1214.39
zice_io input -pin {hw_top.R11.pattern[13]} -con_loc J2120.39
zice_io output -pin {hw_top.S11.pattern[14]} -con_loc J1214.40
zice_io input -pin {hw_top.R11.pattern[14]} -con_loc J2120.40
zice_io output -pin {hw_top.S11.pattern[15]} -con_loc J1214.41
zice_io input -pin {hw_top.R11.pattern[15]} -con_loc J2120.41
zice_io output -pin {hw_top.S11.pattern[16]} -con_loc J1214.42
zice_io input -pin {hw_top.R11.pattern[16]} -con_loc J2120.42
zice_io output -pin {hw_top.S11.pattern[17]} -con_loc J1214.43
zice_io input -pin {hw_top.R11.pattern[17]} -con_loc J2120.43
zice_io output -pin {hw_top.S11.pattern[18]} -con_loc J1214.44
zice_io input -pin {hw_top.R11.pattern[18]} -con_loc J2120.44
zice_io output -pin {hw_top.S11.pattern[19]} -con_loc J1214.45
zice_io input -pin {hw_top.R11.pattern[19]} -con_loc J2120.45
zice_io output -pin {hw_top.S11.pattern[20]} -con_loc J1214.46
zice_io input -pin {hw_top.R11.pattern[20]} -con_loc J2120.46
zice_io output -pin {hw_top.S11.pattern[21]} -con_loc J1214.47
zice_io input -pin {hw_top.R11.pattern[21]} -con_loc J2120.47
zice_io output -pin {hw_top.S11.pattern[22]} -con_loc J1214.48
zice_io input -pin {hw_top.R11.pattern[22]} -con_loc J2120.48
zice_io output -pin {hw_top.S11.pattern[23]} -con_loc J1214.49
zice_io input -pin {hw_top.R11.pattern[23]} -con_loc J2120.49
zice_io output -pin {hw_top.S11.pattern[24]} -con_loc J1214.50
zice_io input -pin {hw_top.R11.pattern[24]} -con_loc J2120.50
```

16.7 Runtime With Direct-ICE

16.7.1 Specific Information in the ZeBu Runtime Database

Depending on the activation of options in the Direct-ICE compilation script, some specific information is added to the ZeBu runtime database for controlling the following options of the Direct-ICE interface:

- *Dynamic-Probes on the Direct-ICE interface*
- *Delay Activation and Programming*
- *Control of the Direct-ICE Shielding*
- *Sampling Feature*

The elements for runtime control of these features are available in a specific hierarchical level, DirectIce:



FIGURE 99. `zSelectProbes` Display for Direct-ICE Debugging Elements

16.7.2 Dynamic-Probes on the Direct-ICE interface

By default, all the Direct-ICE interface signals can be selected as dynamic-probes at runtime using `zSelectProbes`. In `zSelectProbes`, the DirectIce.probes hierarchical level can be used to add some Direct-ICE signals to the dynamic-probes list for debugging purposes. For performance purposes, you can remove the dynamic-probes that are not necessary for your current run.

Note

When accessing dynamic-probes in the design, the design clocks are stopped in ZeBu, which may not be supported by the Direct-ICE target system.

Runtime With Direct-ICE

16.7.2.1 Dynamic Probes Labeled With a Pin Location

The following figures display dynamic-probes with a pin location.

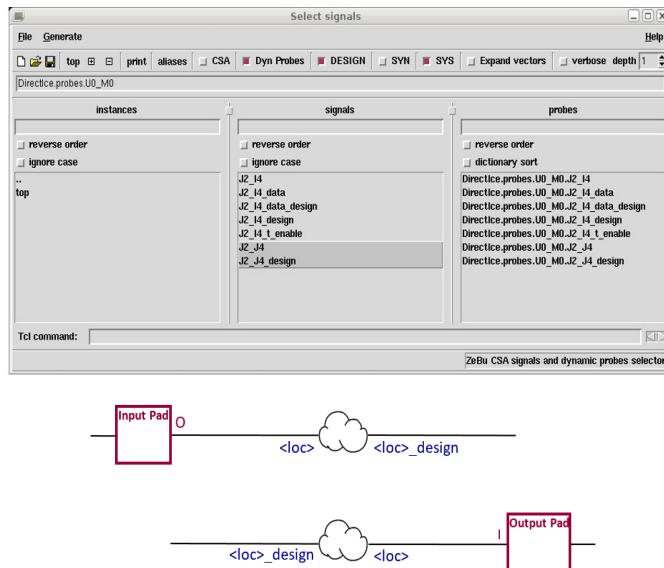
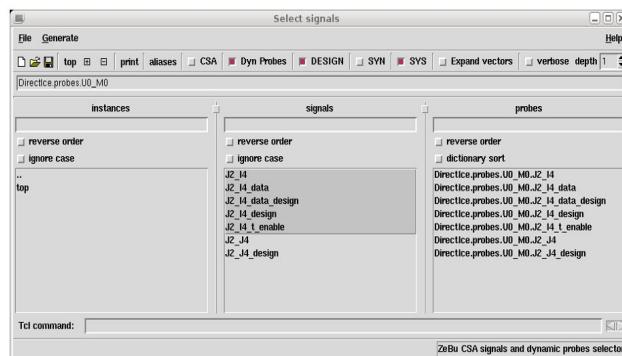
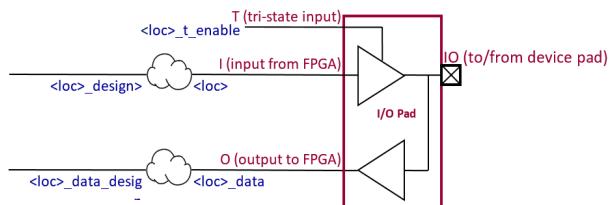


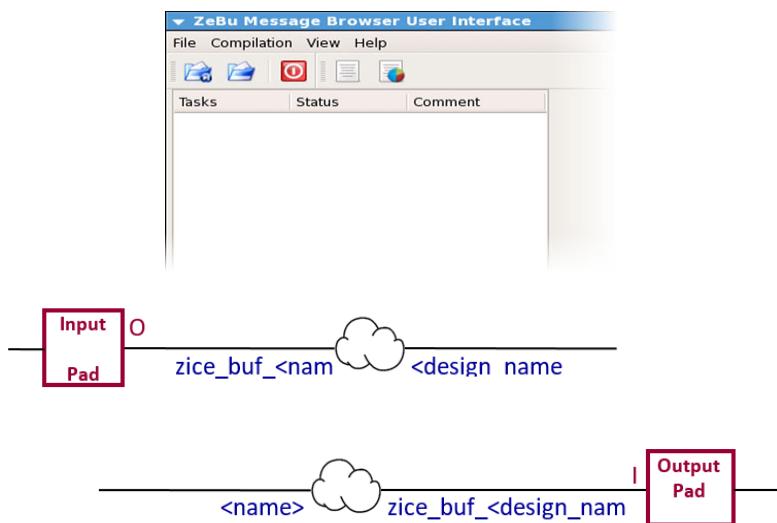
FIGURE 100. Dynamic-Probes on Mono-directional Signals (pin location label)



**FIGURE 101.** Dynamic-Probes on Bi-directional Signals (pin location label)

16.7.2.2 Dynamic Probes Labeled With Design Object Names

The following figures display the dynamic-probes with a design object name label. In the screen examples used in this section, the design object name label is `fromIce`.

**FIGURE 102.** Dynamic-Probes on Mono-directional Signals (design object name)

Runtime With Direct-ICE

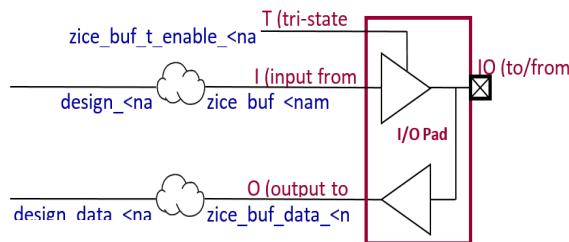
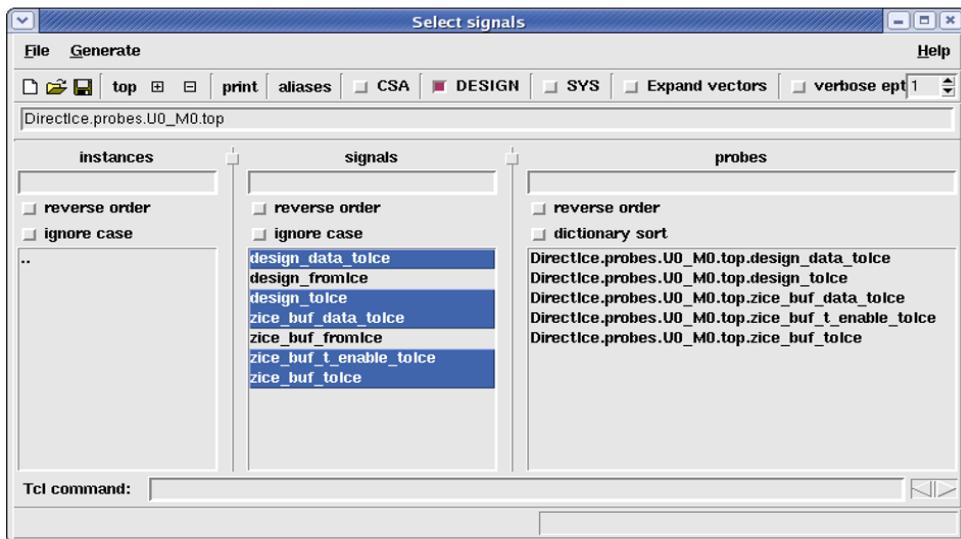


FIGURE 103. Dynamic-Probes on Bi-directional Signals (Design Object Name)

16.7.3 Delay Activation and Programming

The ZeBu runtime database includes the appropriate instances to enable or disable the delay and control its duration.

In the `DirectIce.delays` hierarchical level, there are two instances to control the delay: `delay_enable` and `delay_value`.

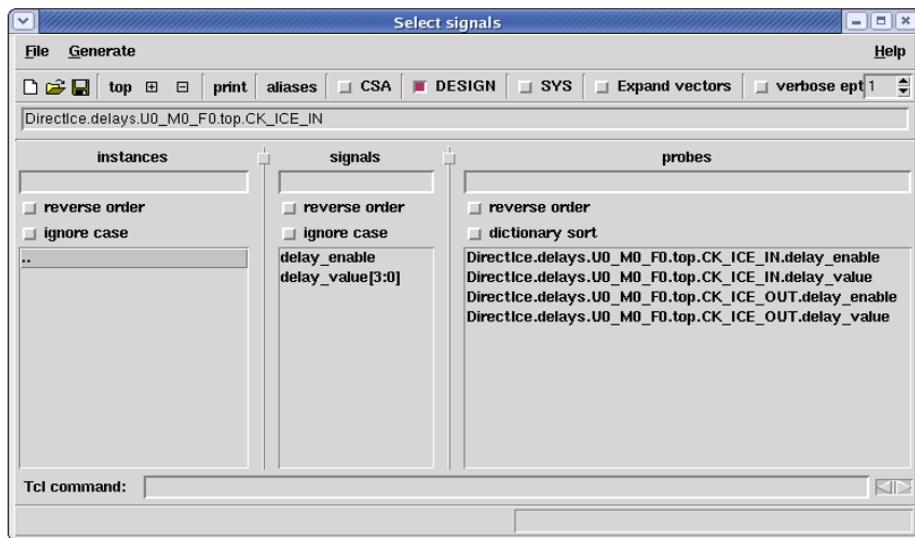


FIGURE 104. Delays Activation and Programming

In the signal names of `DirectIce.delays`, each delayed connection is listed with its delay name:

- `delay_value[3 : 0]`: Delay in the number of delay programming steps (between 0 and 15). As the delay is declared in nanoseconds (ns) in the Direct-ICE Compilation Script, you must check the number of required delay programming steps in [Table 47](#).
- `delay_enable`: Set to 1 to activate the delay; set to 0 to disable the delay.

In `zSelectProbes` you can select these instances for runtime programming with `zRun` or from the user testbench.

16.7.4 Control of the Direct-ICE Shielding

For each Direct-ICE FPGA, the ZeBu runtime database contains an instance in the `DirectIce.io_shielding` hierarchical level to control the Direct-ICE shielding from `zSelectProbes`. This instance is named as the corresponding Direct-ICE FPGA (`Ux_My_Fz`).

Runtime With Direct-ICE

Note

Shielding for Global Output Clocks (GCLK) is not supported in this release.

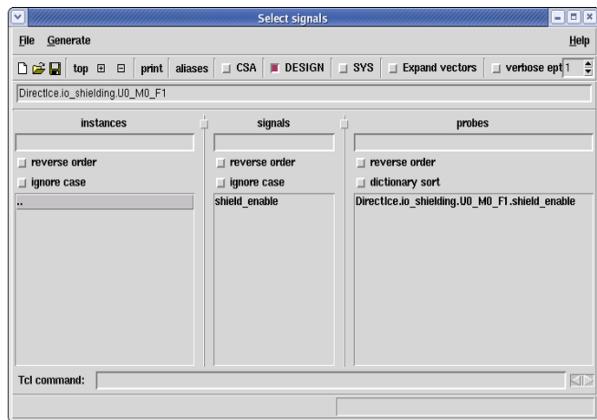


FIGURE 105. Direct-ICE Shielding Control

In each Direct-ICE FPGA, a register named `shield_enable` can be selected as a dynamic- probe to control ICE shielding at runtime (from `zRun` or from user testbench).

The default setting for this register depends on declaration of the `zice shield_ios` command in the Direct-ICE compilation script.

For more details, see [Configuring the Direct-ICE Interface](#).

16.7.5 Sampling Feature

The sampling feature is used for data alignment. The ZeBu runtime database includes the appropriate instances to enable/disable the sampling feature on mono-directional and bi-directional signals.

16.7.5.1 Sampling on Mono-directional Signals

When sampling is used on mono-directional signals, `sampler_enable` is the only instance available in the `DirectIce.sampling` hierarchical level to control the sampling feature.

The following figure displays sampling on mono-directional signals.

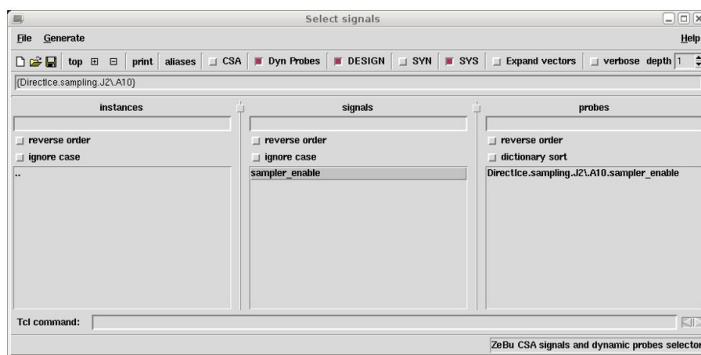


FIGURE 106. Sampling on Mono-directional Signals

If the sampling feature was enabled at compilation time, `sampler_enable` is:

- set to 1 to activate sampling at runtime (default).
- set to 0 to disable sampling at runtime.

16.7.5.2 Sampling on Bi-directional Signals

When sampling is used on bi-directional signals, the following instances are available in the `DirectIce.sampling` hierarchical level to control the sampling feature:

- `sampler_enable`
- `en_sampler_enable`
- `data_sampler_enable`

The following figure displays the sampling on bi-directional signals.

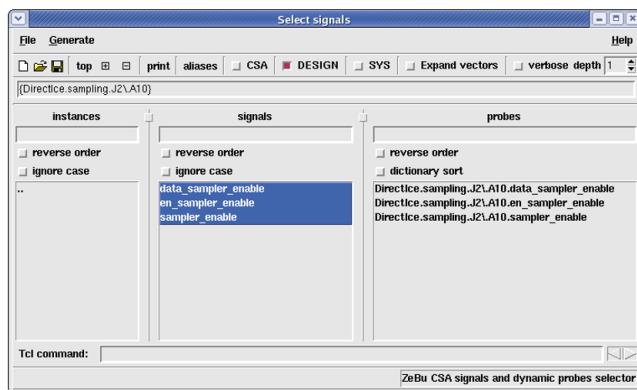
Runtime With Direct-ICE

FIGURE 107. Sampling on Bi-directional Signals

If the sampling feature is enabled at compilation time, then `sampler_enable`, `en_sampler_enable`, and `data_sampler_enable` are:

- set to 1 to activate sampling at runtime (default).
- set to 0 to disable sampling at runtime.

16.8 Mechanical Data

The following figure displays the position of the connectors on the Direct-ICE interface:

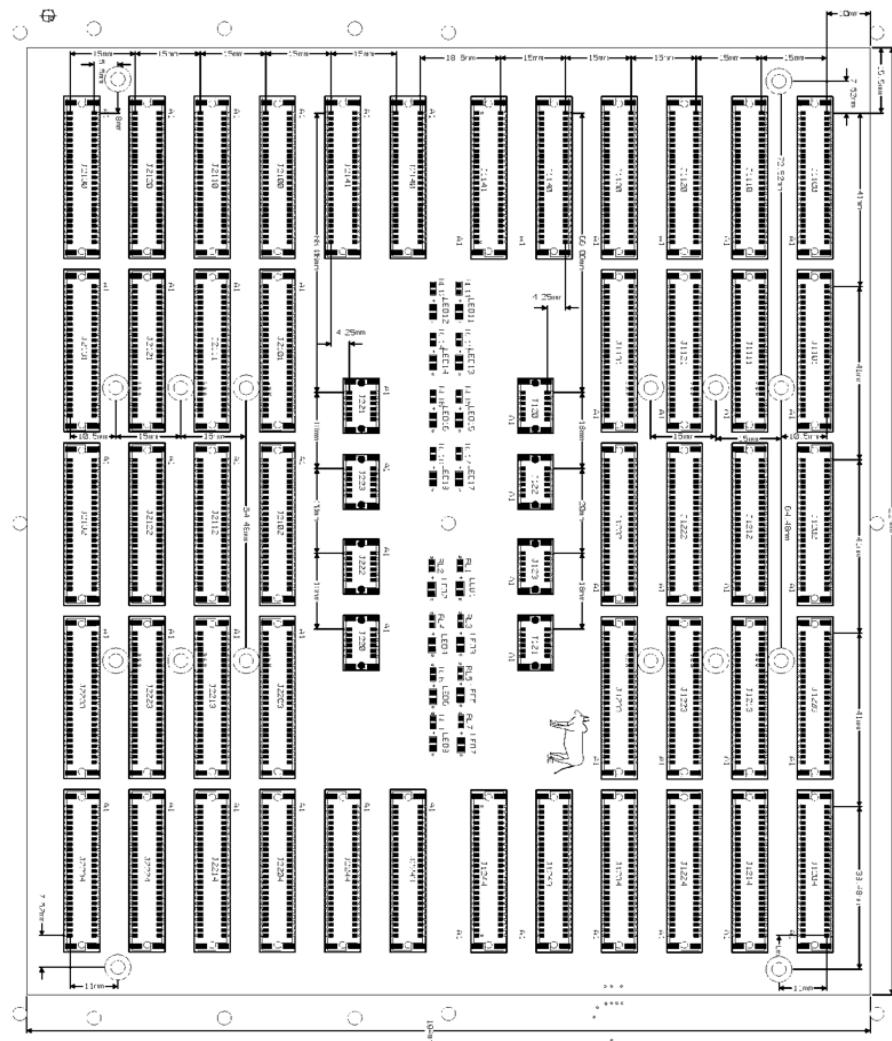


FIGURE 108. Mechanical Position for Direct-ICE Connectors

Mechanical Data

The minimum distance between the Direct-ICE interface and the top of the ZeBu Server chassis is four mm. This additional height should be considered when deciding for the installation facilities of your ZeBu Server system.

The holes in the Direct-ICE interface are designed for spacers, if you want to fix a board to it. The diameter of the holes is 3.2 mm. The spacers are not mounted by default.

Note

The installation of Direct-ICE connectors must be performed by Synopsys personnel only.

17 Smart Z-ICE Interface for ZeBu Server 3

The ZeBu Server 3 Smart Z-ICE interface connects a DUT with a target system. It is intended for use with a fast serial interface for software debug such as JTAG interface. The Smart Z-ICE interface consists of:

- 4x 50-pin connectors on the rear panel of two-slot ZeBu Server units
- 5x 50-pin connectors on the rear panel of five-slot ZeBu Server units

Each connector comprises 16 I/O pins, one clock pin, and two power supply pins.

The voltage level on this interface is programmable from 1.5 V to 3.3 V.

To make integration easier with a software debugger, an adapter is available to connect to the Smart Z-ICE interface using standard HE10 connectors. One HE10 adapter can be plugged on each Smart Z-ICE port in the rear of the ZeBu Server unit.

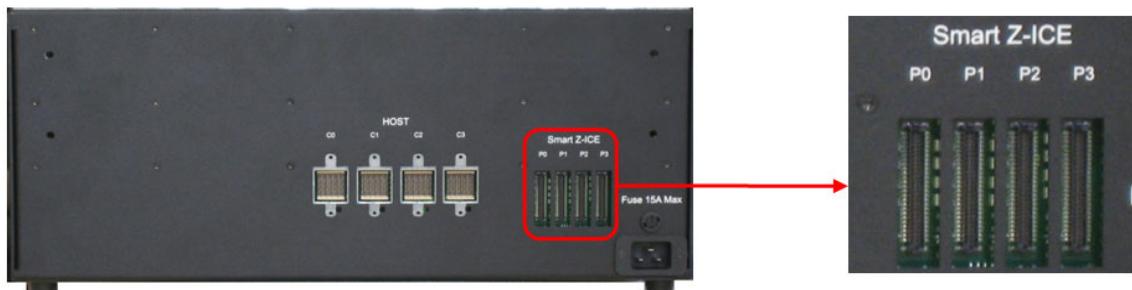


FIGURE 109. Smart Z-ICE Connectors on the Rear Panel of a Two-slot Unit

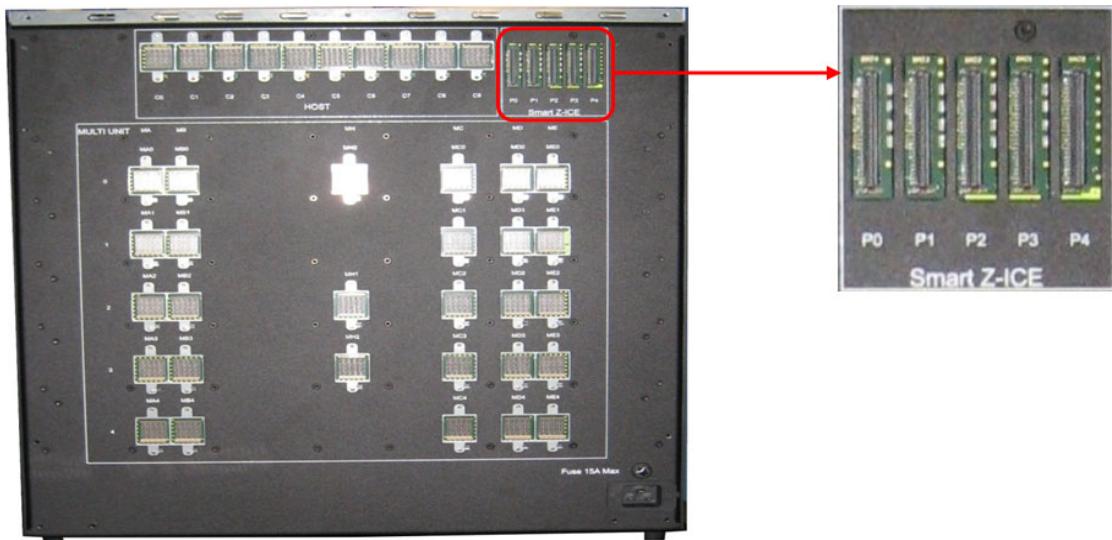


FIGURE 110. Smart Z-ICE Connectors on the Rear Panel of a Five-Slot Unit

This section discusses the following topics.

- *Connection Principle*
- *Physical Description of Smart Z-ICE Interface*
- *Smart Z-ICE HE10 Adapter*
- *Electrical Characteristics*
- *Connecting a Pod to the Smart Z-ICE Connector*
- *Driver Declaration*
- *Remapping for Runtime*
- *Examples*

Connection Principle

17.1 Connection Principle

The data and clock signals of the Smart Z-ICE interface are routed to the DUT using a dedicated FICE FPGA (on the backplane) and the IF (one on each module).

17.1.1 Single-Unit Configuration

In the two-slot units, the Smart Z-ICE interface has four Smart Z-ICE connectors (P0 to P3) and the four clock pins can be used at the same time. The following figure displays the connection for a two-slot unit.

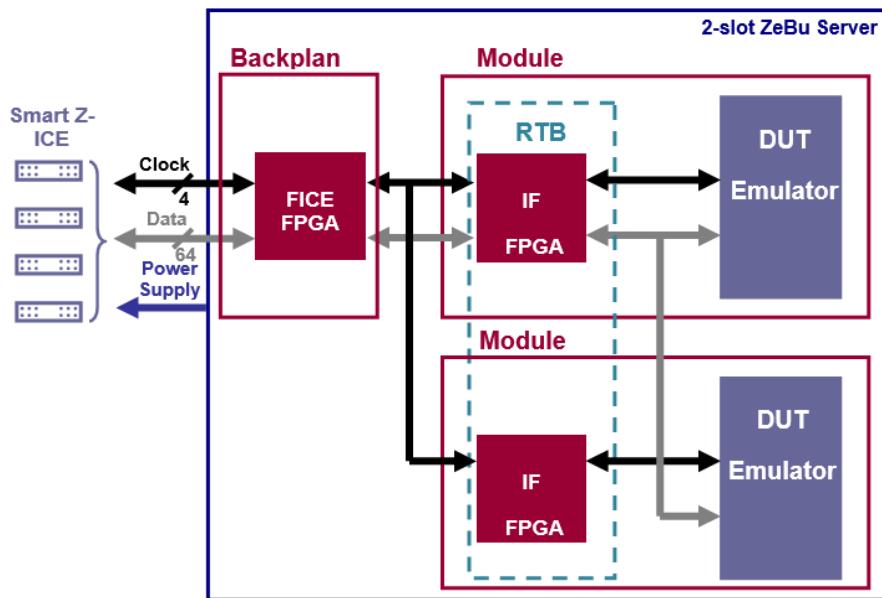


FIGURE 111. Connection Principle for a Two-Slot Unit

In the five-slot units, the Smart Z-ICE interface has five Smart Z-ICE connectors (P0 to P4); but, only four clock pins can be used at the same time. The following figure displays the connection for a five-slot unit.

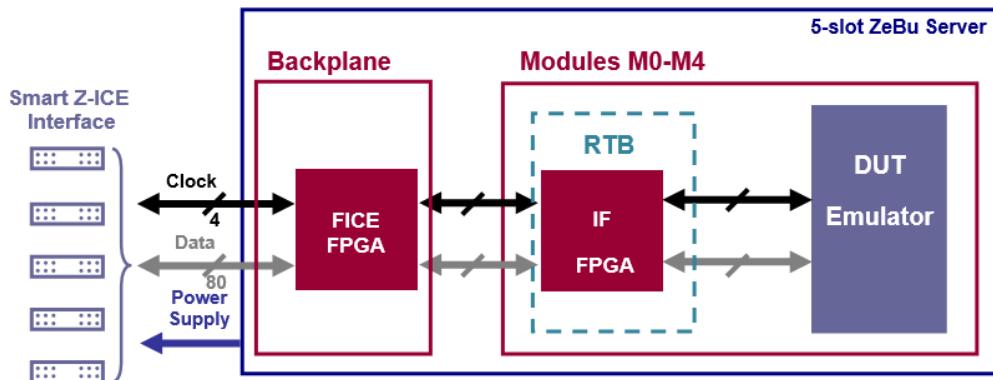


FIGURE 112. Connection Principle for a Five-Slot Unit

17.1.2 Multi-Unit Configuration

In a multi-unit configuration, the Smart Z-ICE feature is only available on the connectors of unit U0. Therefore, the Smart Z-ICE cables must be plugged into the unit U0. The following figure displays the connection for a multi-unit configuration.

Connection Principle

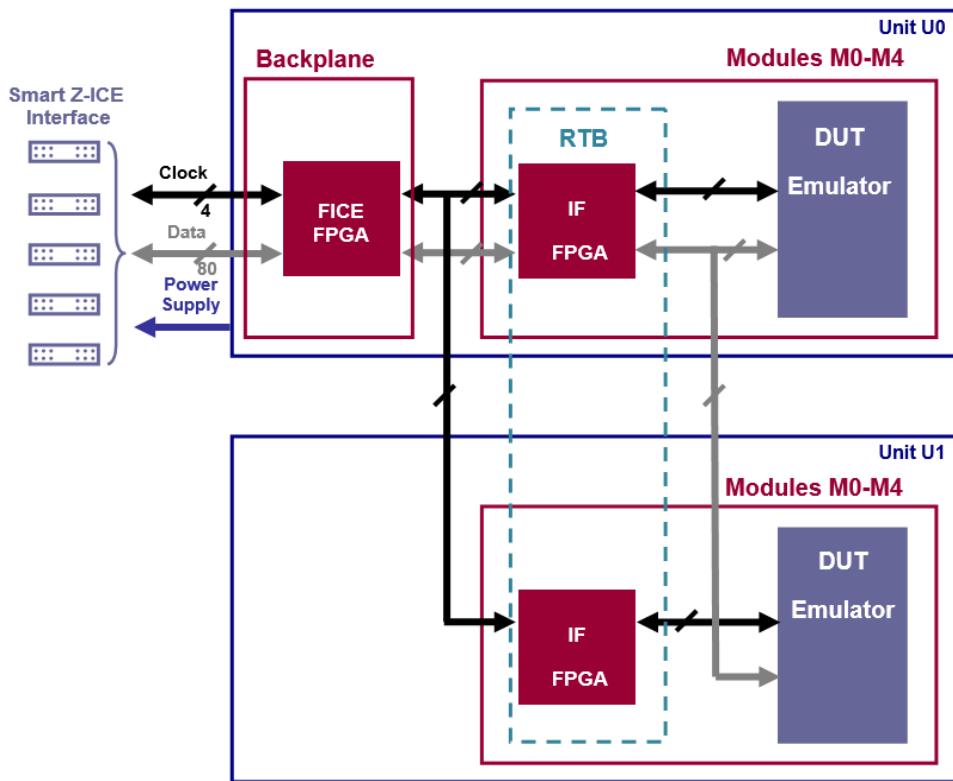


FIGURE 113. Connection Principle in a Multi-Unit Configuration

17.2 Physical Description of Smart Z-ICE Interface

This section discusses the physical description of Smart Z-ICE interface. See the following subsections:

- *Smart Z-ICE Connectors*
- *Pinout*

Smart Z-ICE Connectors

Each Smart Z-ICE connector on the ZeBu Server system rear panel is a 50-pin male connector, distributed by ERNI (Ref. 064319).

For flat cable connection, you must use an ERNI IDC female type B connector with latching locks (Ref. 124262) and the appropriate flat cable (Ref. 034575). For the datasheets of these connectors, see <http://www.erni.com>. The following figures display the Smart Z-ICE Connectors on the ZeBu Server system.

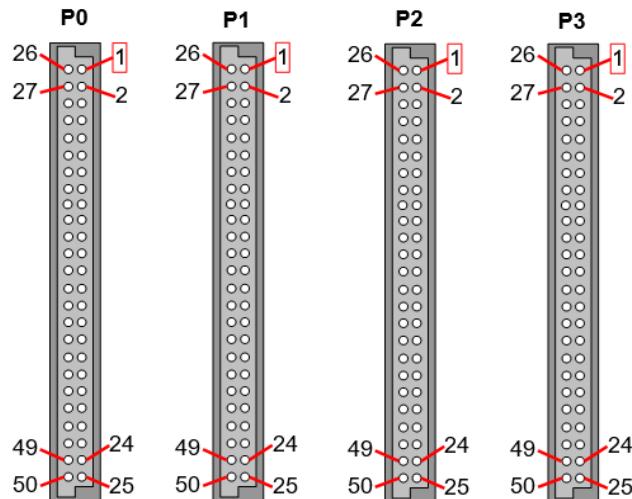


FIGURE 114. Smart Z-ICE Connectors on the Two-Slot ZeBu Server Unit

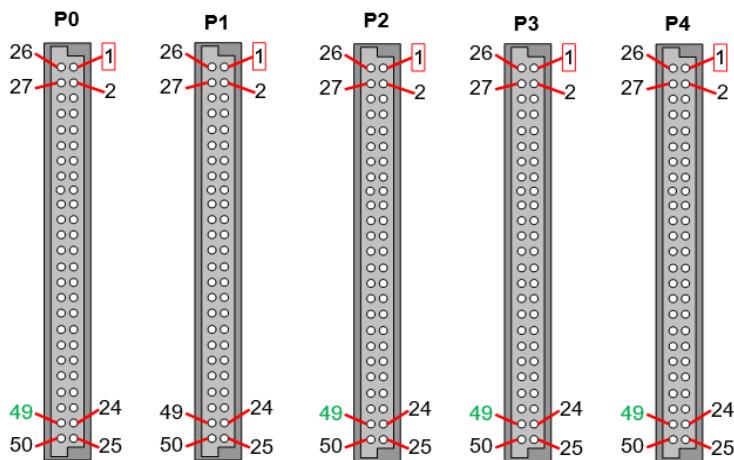
Physical Description of Smart Z-ICE Interface


FIGURE 115. Smart Z-ICE Connectors on the Five-Slot ZeBu Server Unit

Pinout

When a data or clock pin is declared as an input, the signal is transmitted from the target to the DUT. When a pin is declared as an output, the signal is transmitted from the DUT to the target. The following table lists the pin allocation on a Smart Z-ICE connector.

TABLE 54 Pin Allocation on a Smart Z-ICE Connector

Smart Z-ICE Pin #	Signal Name
1 to 24	Ground
25-26	Power Supply (VCC)
27 to 42	Data
43 to 48	Ground
49	Clock
50	Ground

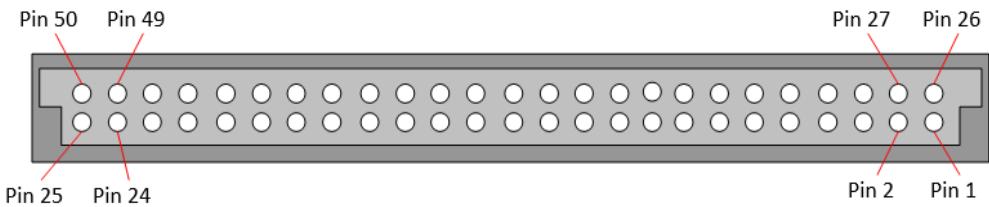


FIGURE 116. Pin Numbering on a Smart Z-ICE Connector

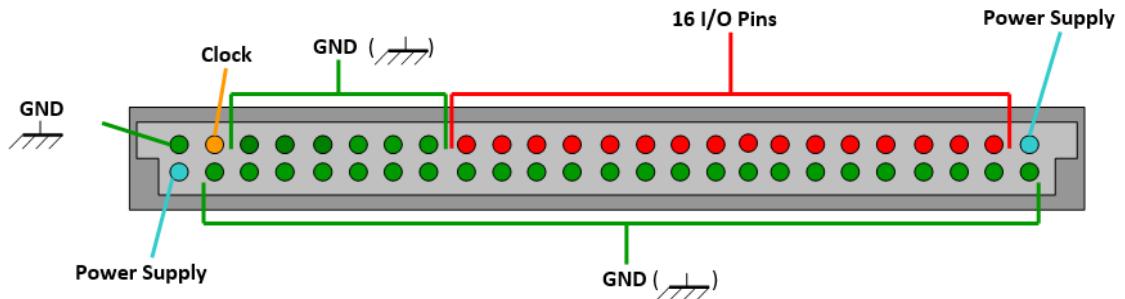


FIGURE 117. Pin Allocation on a Smart Z-ICE Connector

17.3 Smart Z-ICE HE10 Adapter

There is no functional difference while connecting to the Smart Z-ICE with the HE10 adapter. Note that all the signals of the interface are available and the same operating modes are supported.

The ZeBu Server Smart Z-ICE HE10 adapter provides direct connections between the ERNI connector and the HE10 connector as displayed in the following figure.

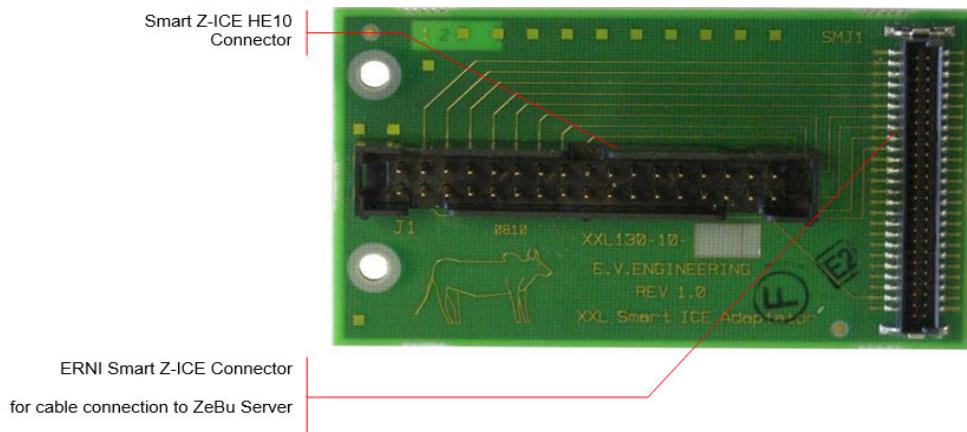


FIGURE 118. ZeBu Server Smart Z-ICE HE10 Adapter

A female ERNI connector is available on the bottom side of the adapter for direct connection (without cable) to a Smart Z-ICE connector on the back panel of a ZeBu Server unit.

The Smart Z-ICE HE10 adapter has the following interface:

- Two power supply pins with programmable voltage level
- One clock pin
- 16 data pins
- 15 pins connected to ground

The electrical characteristics of the Smart Z-ICE interface are not modified by the Smart Z-ICE HE10 adapter. For more details, see [Pinout](#).

The connectors are standard 34-pin, male HE10 connectors. The mating plug is intended for use with a 1.27 mm (.050") ribbon cable.

Pinout

The pin numbering on the Smart Z-ICE HE10 adapter is standard HE10 pin numbering. The following table lists pin allocation on Smart Z-ICE HE10 adapter connector.

TABLE 55 Pin Allocation on Smart Z-ICE HE10 Adapter Connectors

HE-10 Connector Pin#	Signal Name	HE-10 Connector Pin#	Signal Name
1	Data	2	VCC
3	Data	4	GND
5	Data	6	GND
7	Data	8	GND
9	Data	10	GND
11	Data	12	GND
13	Data	14	GND
15	Data	16	GND
17	Data	18	GND
19	Data	20	GND
21	Data	22	GND
23	Data	24	GND
25	Data	26	GND
27	Data	28	GND
29	Data	30	GND
31	Data	32	GND
33	Clock	34	VCC

Smart Z-ICE HE10 Adapter

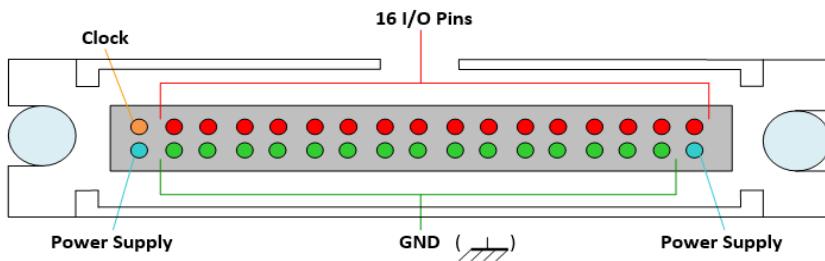


FIGURE 119. Pin Allocation on the Smart Z-ICE HE10 Adapter

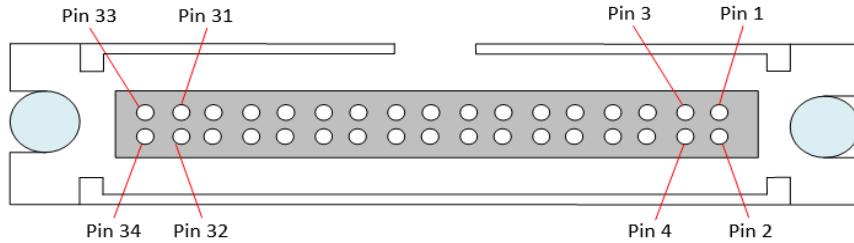


FIGURE 120. Pin Numbering on the Smart Z-ICE HE10 Adapter

When a data or clock pin is declared as an input, the signal is transmitted from the target to the DUT. When a pin is declared as an output, the signal is transmitted from the DUT to the target.

17.4 Electrical Characteristics

The Smart Z-ICE interface is controlled by FICE, a dedicated Xilinx Virtex FPGA on the ZeBu Server backplane. For details about the electrical characteristics, see the XC2C512 datasheet available on the Xilinx website (<http://www.xilinx.com>).

The voltage on the Smart Z-ICE interface is programmable with the following voltages:

- 1.5 V,
- 1.8 V,
- 2.5 V, or
- 3.3 V.

The voltage regulator on Smart Z-ICE power supply pins is a YNM12S05 connector. The output tolerance of the voltage regulator is $\pm 2.5\%$ in the complete operating range. The maximum authorized current on the Smart Z-ICE interface is 250 mA (overall current for all the ports in use).

Connection to the FICE FPGA is protected by resistors as displayed in the following figure.

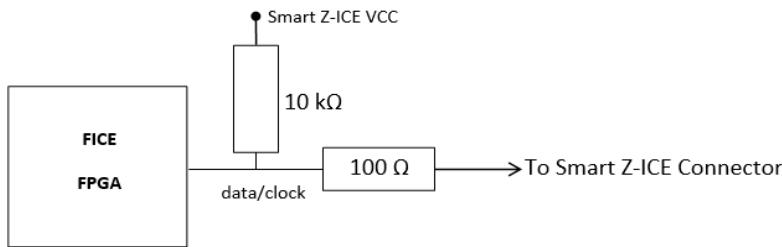


FIGURE 121. Connection to FICE FPGA

Note

Mishandling on Smart Z-ICE connector pins can nevertheless damage the device.

17.5 Connecting a Pod to the Smart Z-ICE Connector

To connect a pod to a Z-ICE port on a ZeBu Server unit (with a Linux PC already connected to ZeBu Server), perform the following steps:

1. Switch OFF the ZeBu Server unit.
2. Switch OFF the pod.
3. Connect the pod to a Smart Z-ICE port on the ZeBu Server unit.
4. Switch ON the pod.
5. Switch ON the ZeBu Server unit.

Note

Do not switch OFF the Linux PC.

17.6 Driver Declaration

The Smart Z-ICE interface requires a specific driver for correct RTB configuration to drive the target. The appropriate SMART_ZICE_ZSE driver is provided in the ZeBu software package. It must be instantiated in the hardware top with declaration of I/O and clock pins with their direction.

At compilation, this driver produces the necessary files to configure the ZeBu system and to connect the DUT to the physical Smart Z-ICE interface. This section provides details for use of the Smart Z-ICE interface, and displays complete examples in the “[Examples](#)” chapter.

17.6.1 Smart Z-ICE Driver Instantiation

To use SMART_ZICE_ZSE in Unified Compile (UC) flow, instantiate a pre-defined module SMART_ZICE_ZSE in Verilog code.

A wrapper module “SMART_ZICE_ZSE” is provided for you to instantiate it and specify the connection for each pin. Each pin ID can only have one direction port used for connection.

Following connection is not allowed.

```
.input_port0_27 (signal1)
.output_port0_27 (signal2)
```

The following is the definition of the wrapper module

```
module SMART_ZICE_ZSE #(
    parameter vcc = "",           // "1v5", "1v8", "2v5", "3v3"
    parameter connector = "", // could be "HE10"
    parameter sampling_clock = "",
    parameter clock_in_port0_49 = "", // the parameter ID for
clock will also parameter clock_out_port0_49 = "", // follow the
rule in legacy flow,           // the clock ID will be "33" if "HE10"
specified
    parameter clock_in_port4_49 = "",
    parameter clock_out_port4_49 = "")
(input_port0_42, input_port0_41, ... input_port0_28,
input_port0_27, input_port1_42, input_port1_41, ... input_port0_28,
input_port1_27, input_port4_42, input_port4_41, ... input_port4_28,
input_port4_27, output_port0_42, output_port0_41, ...
output_port0_28, output_port0_27,... output_port4_42,
output_port4_41, ... output_port4_28, output_port4_27,
inout_port0_42, inout_port0_41, ... inout_port0_28, inout_port0_27, ...
inout_port4_42, inout_port4_41, ... inout_port4_28, inout_port4_27)
```

```
// these Port ID will be odd number from "1" to "31" if "HE10"
specified.

);
```

You must adhere to the following guidelines for using the Smart-ICE.

- Only one SMART_ZICE_ZSE instance is allowed for whole design.
- input_port0_42, output_port0_42, inout_port0_42 represent the data pin number 42 of Port0. As the same pin cannot be declared with different types of connection, check the type of port connections and ensure that they must comply with this rule.

This is NOT allowed:

```
( .input_port0_42(signalA[0]), .output_port0_42(signalB[5]));
```

- If you want to use Port0 No.42 data pins as "input", leave other connections empty or do not instantiate them.

Example: (. input_port0_42(signalA[0]), .output_port0_42());

- If you want to configure the "clock" port for Port0, use port connection similar to data pins. In the legacy flow, clock_in and clock_out are exclusive and cannot be specified simultaneously. The following example shows an incorrect definition of the clock port configuration.

Example: (.clock_in_port0_49 (clock) ,
.clock_out_port1_49(iceClock)) ;

- All these rules of No.42 Data pin of Port0 are also applied on other Ports in the same way.

- vcc can be specified with the following values in UTF command:

"1v5", "1v8", "2v5", or "3v3" (no case sensitivity).

The default value for the Smart Z-ICE voltage is 1.5 V. If you need another voltage, use UTF command as follows:

```
smart_zice_config -vcc "3v3"
```

- connector is used to specify if you want to use adapter HE10. However, the interface data pin mapping is different.

```
smart_zice_config -connector "HE10"
```

17.6.2 Declaring the Type of Physical Interface

By default, the ZeBu Server Smart Z-ICE driver declaration uses the Smart Z-ICE interface described in [Table 54](#).

To use the Smart Z-ICE HE10 adapter, declare it in the UTF file:

```
smart_zice_config -connector HE10
```

When the connector parameter is set to HE10, the Smart Z-ICE driver declaration uses the Smart Z-ICE HE10 adapter interface described in [Table 55](#). Therefore, you do not need to manually convert pin numbers in the UTF file.

17.6.3 Voltage Declaration

The Smart Z-ICE interface voltage level is programmable at 1.5 V, 1.8 V, 2.5 V, or 3.3 V. The programmed voltage is applicable for the complete Smart Z-ICE interface. See the following syntax:

```
smart_zice_config [-connector <CONNECTOR>] [-vcc <VCC>]
```

where,

- CONNECTOR is a string in {HE10|default};
- VCC is a string in {1v5|1v8|2v5|3v3|1V5|1V8|2V5|3V3}

The setting of the Smart Z-ICE voltage must be declared for the setup phase of the ZeBu Server system. The voltage setting is stored in the following section of the `setup_template.zini` file located in the ZeBu system directory (`$(ZEBU_SYSTEM_DIR)`):

```
# Specify default Unit #num Smart Z-ICE Vcc (1.5 V by default)
# See below for available power supply voltages (e.g. Unit 0)
# Select one of the following lines:
#$U0.SmartZice.Vcc = "1.5 V";
#$U0.SmartZice.Vcc = "1.8 V";
#$U0.SmartZice.Vcc = "2.5 V";
#$U0.SmartZice.Vcc = "3.3 V";
```

The default value for the Smart Z-ICE voltage is 1.5 V. If you need another voltage level for the Smart Z-ICE interface, you must modify the `setup_template.zini` file and launch the setup phase again with `zSetupSystem`.

17.6.4 Data Declaration

You must declare the I/O pins in the driver instantiation as follows:

```
.<dir_type>_<connectorID>_<pinID>(<dut_signal>)
```

where,

- `<dir_type>` is the type of connection (input, output or inout).
- `<connectorID>` is the port number: `port0`, `port1`, `port2`, `port3` (or `port4` in 5-slot units).
- `<pinID>` is the pin number on the Smart Z-ICE connector (For details, see [Pinout](#)).

17.6.5 Pin Declarations on Smart Z-ICE Connector

By default, the ZeBu Server Smart Z-ICE interface is used directly. In such a case, use the pins dedicated to Data in when instantiating the Smart Z-ICE driver:

For Example,

```
// pin 27 of P0 declared as an input  
.input_port0_27(signal1)  
// pin 28 of P0 declared as an output  
.output_port0_28(signal2)  
// pin 29 of P0 declared as bi-directional  
.inout_port0_29(signal3)
```

17.6.6 Pin Declarations With Smart Z-ICE HE10 Adapter

To use the Smart Z-ICE HE10 adapter, perform the following steps:

1. Declare the connector type in the UTF file
For details, see [Declaring the Type of Physical Interface](#).
2. Use the pins dedicated to Data in when instantiating the Smart Z-ICE driver
For Example,

```
// HE10 pin 31 of P0 declared as an input  
.input_port0_31(signal1)  
// HE10 pin 29 of P0 declared as an output  
.output_port0_29(signal2)  
// HE10 pin 27 of P0 declared as bi-directional  
.inout_port0_27(signal3)
```

17.6.7 Ambiguous Pin Declarations

Ambiguous declaration with different types of connection for the same pin (for example input and output) causes a compilation error. For example, the following declarations are not allowed:

```
.input_port0_27 (signal)  
.output_port0_27 (signal)  
.inout_port0_27 (signal)
```

17.6.7.1 Declaring the Clock Pin in the designFeatures File

Define the waveform and the frequency of the clock signal explicitly in the **designFeatures** file or in a separate clock file.

```
$B0.<my_clock>.mode = "controlled";
$B0.<my_clock>.Waveform = "-";
```

where, `<my_clock>` is the name of the clock signal to connect to the pin (a controlled clock in this example).

Note *If you are using a separate clock definition file, the designFeatures file contains the path to this clock file.*

17.6.8 Clock Pin as an Input

When the clock pin is an input, it is declared for compilation, but it does not require any definition for runtime in the **designFeatures** file.

17.6.9 Declaration for Synchronous Clock Handling With Smart Z-ICE

When the Smart Z-ICE interface provides a primary clock to the design (for example when connecting a JTAG debugger), the primary clock might have a different frequency than the clocks generated by the ZeBu clock generator.

The frequency of this clock is usually maintained as same as the fastest clock in the DUT considered in the **designFeatures** file.

To avoid this, the ZeBu compiler can use a fast primary controlled clock (output of a `zceiClockPort`) to sample the Smart Z-ICE input clock. For that purpose, the following lines should be present in the Verilog file:

To declare the Smart Z-ICE input clock, see [Clock Pin as an Input](#).

Declaration of the fast primary controlled clock:

```
zceiClockPort <my_ClockPort> ( .cclock(<my_fast_clock>) );
```

Declaration of the `sampling_clock` parameter for the Smart Z-ICE driver:

```
zceiClockPort clk_ClockPort_for_smarHzice (
    .cclock( clk_sampling_smarHzice )
);
```

Note

The Smart Z-ICE input clock is sampled on the rising edge (posedge) of the fast controlled clock.

17.7 Remapping for Runtime

When a design compiled for a given set of Smart Z-ICE connectors needs to be run with a different set of connectors, you can skip the recompilation of the design. In this case, the remapping to the actual connectors must be declared in the `designFeatures` file or using a dedicated environment variable.

17.7.1 Declaration of the Connector in the `designFeatures` File

The following line should be added when the design is compiled for connector i but connector j is intended for runtime:

```
$smarHzICE.connectorRemap_i = j;
```

where,

- i is the compilation index of the Smart Z-ICE connector.
- j is the index of the actual Smart Z-ICE connector for runtime.

For Example,

When the design is compiled for connectors 0 and 1 but the connectors used are connectors 2 and 3, the following lines should be added in the `designFeatures` file:

```
$smartZICE.connectorRemap_0 = 2;
$smartZICE.connectorRemap_1 = 3;
```

17.7.2 Declaration of the Connector Using an Environment Variable

When the same project is used on several ZeBu systems on which the Smart Z-ICE targets are connected differently, modifying the **designFeatures** file is not appropriate. In such a case, you must declare the following environment variable when the design is compiled for connector i but connector j is intended for runtime:

```
ZEBU_SMARTZICE_Pi_PHYSICAL_LOCATION=Pj
```

For Example,

When the design is compiled for connectors 0 and 1 but the connectors used are connectors 2 and 3, the following environment variable should be set (example is for bash shell):

```
$ export ZEBU_SMARTZICE_P0_PHYSICAL_LOCATION=P2
$ export ZEBU_SMARTZICE_P1_PHYSICAL_LOCATION=P3
```

17.7.3 Disabling Smart Z-ICE ports at Runtime

To disable at runtime all the Smart Z-ICE ports that have been enabled during compilation, set the environment variable **ZEBU_SMARTZICE_DISABLE** as TRUE.

A warning message appears as follows:

WARNING: You've compiled for using SmartZice but you've decided not to use it by setting the variable, **ZEBU_SMARTZICE_DISABLE** as OK|YES|TRUE|ON.

17.8 Examples

In the following examples, the SMART_ZICE_ZSE driver is instantiated as my_smart_zice and the voltage level is programmed at 1.8 V.

17.8.1 Mono-Directional Interface: Example 1

32 data signals are connected to ports P0 and P1.

- 16 output signals (smart_out[1] to smart_out[16]) connected to port0
- 16 input signals (smart_in[18] to smart_in[33]) connected to port1

The P0 clock is an output to the target and is mapped to smart_out[0].

The P1 clock is an input from the target and is mapped to smart_in[17].

Both the sampling clocks are generated through a 'zceiClockPort' module:

- zceiClockPort clk_ClockPort_for_smartzice (.cclock(smart_clkout));
- zceiClockPort clk_ClockPort_for_smartzice (.cclock(smart_clkint));

Both clocks are mapped to smart_out[0] and smart_in[17] in the Verilog code respectively.

17.8.2 Mono-Directional Interface: Example 2

84 data signals are connected to ports P0 through P4.

- 42 output signals and 2 output clocks:
 - 8 signals (smart_out[1] to smart_out[8]) connected to port2
 - 16 signals (smart_out[10] to smart_out[25]) connected to port3
 - 16 signals (smart_out[26] to smart_out[41]) connected to port4
- 42 input signals and 2 input clocks:
 - 8 signals (smart_in[1] to smart_in [8]) connected to port2
 - 16 signals (smart_in[10] to smart_in [27]) connected to port0

Examples

- 16 signals (`smart_in[26]` to `smart_in [41]`) connected to port1

The P0 clock is an output to the target and is mapped to `smart_out [0]`.

The P1 clock is an output to the target and is mapped to `smart_out [9]`.

The P2 clock is an input from the target and is mapped to `smart_in [0]`.

The P3 clock is an input from the target and is mapped to `smart_in [9]`.

These sampling clocks are generated through a 'zceiClockPort' module:

- `zceiClockPort clk_ClockPort_for_smartzice (.cclock(smart_clkout_0));`
- `zceiClockPort clk_ClockPort_for_smartzice (.cclock(smart_clkout_1));`
- `zceiClockPort clk_ClockPort_for_smartzice (.cclock(smart_clkin_0));`
- `zceiClockPort clk_ClockPort_for_smartzice (.cclock(smart_clkin_1));`

All clocks are mapped to `smart_out [0]`, `smart_in[9]`, `smart_in[0]`, and `smart_in[9]` in the Verilog code respectively.

17.8.3 Example of Bi-Directional Interface

Six data signals are connected to ports P0, P1, and P2.

- Two input signals (`smart_in[1]` and `smart_in[2]`) connected to port0
- Two output signals (`smart_out[1]` and `smart_out[2]`) connected to port1
- Two inout signals (`smart_io[0]` and `smart_io[1]`) connected to port2

The P0 clock is an input from the target and is mapped to `smart_in[0]`.

The P1 clock is an output to the target and is mapped to `smart_out[0]`.

Both clocks are defined through defparam statements:

- `smart_out[0]` is a controlled clock.
- `smart_in[0]` is a primary clock.

17.8.4 Example of SMART_ZICE_ZSE in UC Flow

The following is an example for SMART_ZICE_ZSE in the UTF flow.

Verilog source code:

```
Module hw_top ;
  SMART_ZICE_ZSE my_smart_zice(
    .clock_out_port0_49(smart_clkout), //clock ports
    .clock_in_port1_49(smart_clkin),
    .input_port0_42( smart_out_load[3] ), //data ports
    .input_port0_41( smart_out_load[2] ),
    .input_port0_40( smart_out_load[1] ),
    .input_port0_39( smart_out_load[0] ),
    .output_port1_42( smart_in_driver[3] ),
    .output_port1_41( smart_in_driver[2] ),
    .output_port1_40( smart_in_driver[1] ),
    .output_port1_39( smart_in_driver[0] ),
  );
  zceiClockPort smart_clkout_port ( .cclock( smart_clkout ) );
  zIceClockPort smart_clkin_port ( .clock( smart_clkin ) );

endmodule
```

UTF File:

```
smart_zice_config -vcc "1v8"
```

18 Smart Z-ICE Support on ZeBu Server 4

This section describes the following sub-topics.

- *Physical Description of Smart Z-ICE Interface*
- *Instantiating Smart Z-ICE*
- *Using the Smart Z-ICE During Emulation*

18.1 Physical Description of Smart Z-ICE Interface

The Smart Z-ICE interface supported by ZeBu Server 4 is based on standard HE10 connectors. The interface consists of 6x34-pin connectors available on the front panel of the ZeBu Server 4 Control Interface. The following figure displays the port connectors on the ZeBu Server 4 control interface.



FIGURE 122. Smart Z-ICE Port Connectors on ZeBu Server 4 Control Interface

In ZeBu Server 4, Smart Z-ICE voltage level cannot be configured and is fixed at 1.8V.

The 34pins of each Smart Z-ICE connector are allocated as follows:

- 16 pins are I/O pins for data
- 15 pins are connected to GND
- 2 pins are connected to power supply
- 1 pin for clock

Physical Description of Smart Z-ICE Interface

The following figure displays the HE10 connector and the pin allocation.

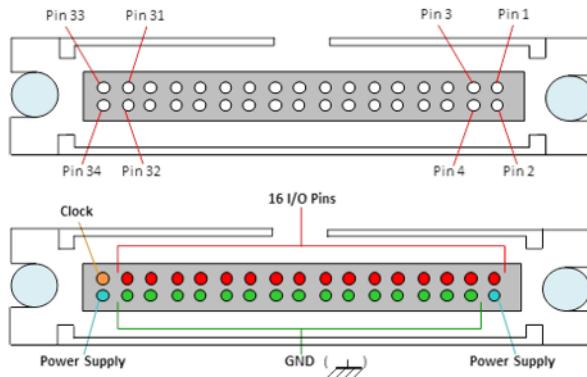


FIGURE 123. Pin Allocation in the ZeBu Server 4 Smart Z-ICE HE10 Connector

The following table lists the pin allocation on ZS4 Smart Z-ICE HE10 connector.

TABLE 56 ZS4 Smart Z-ICE HE10 Connector Pins for Data

HE10 Connector Pin#	Signal Name
1	Data
3	Data
5	Data
7	Data
9	Data
11	Data
13	Data
15	Data
17	Data
19	Data
21	Data
23	Data

TABLE 56 ZS4 Smart Z-ICE HE10 Connector Pins for Data

HE10 Connector Pin#	Signal Name
25	Data
27	Data
29	Data
31	Data
33	Clock

18.2 Instantiating Smart Z-ICE

TABLE 57 ZS4 Smart Z-ICE HE10 Connector Pins for VCC and GND

HE10 Connector Pin#	Signal Name
2, 34	VCC
4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32	GND

In ZeBu Server 4, the Smart Z-ICE connection is by instantiating a dedicated module named SMART_ZICE_ZSE. This module can be instantiated anywhere in the design hierarchy (DUT or transactor); it is supported only in System Verilog and Verilog languages.

The following is the syntax of the module instantiated to use Smart Z-ICE.

```
SMART_ZICE_ZSE (
    .sampling_clock (<sampling clock name>),
    .<port direction>_port<port number>_<pin number> (<wire name>),
    .<port direction>_port<port number>_<pin number> (<wire name>),
    .<port direction2>_port<port number2>_<pin number2> (<wire name2>),
```

Instantiating Smart Z-ICE

```

.clock_out_port<port number>_33 (clock wire name)
);

```

The following table describes each port of the SMART_ZICE_ZSE module.

TABLE 58 Description of ports of SMART_ZICE_ZSE Module

Port	Description
.<port direction>_port<port number>_<pin number>(<wire name>)	<p>Specifies the connection between a specific Smart Z-ICE port, which is defined with a direction (<port direction>), a port number (<port number>), a pin number (<pin number>), and a wire (<wire name>). Note that the code must be written in Verilog or System Verilog.</p> <ul style="list-style-type: none"> • <port direction> accepts the following values: <ul style="list-style-type: none"> - input: Data is transferred from Smart Z-ICE. - output: Data is transferred to Smart Z-ICE. wire <wire name> drives value on Smart Z-ICE. - inout: It means that the wire <wire name> is a bi-directional wire. • <port number> is an integer between 0 and 5. This is linked to the Smart Z-ICE port number expected to be used (see Figure 122) • <pin number>: Represents the specific pin to be used on the Smart Z-ICE port. Table 56 summarizes the allocation for each pin available on each Smart Z-ICE connector.

TABLE 58 Description of ports of SMART_ZICE_ZSE Module

Port	Description
.clock_out_port<port number>_33 (clock wire name)	Specifies the Smart Z-ICE port dedicated to the clock. In contrast to ZeBu Server 3, only clock going from design to Smart Z-ICE is supported.
.sampling_clock (<sampling clock name>)	Specifies <sampling clock name> as the sampling clock for data coming from Smart Z-ICE. The main purpose is to synchronize the input data of Smart Z-ICE. <sampling clock name> can be one of the following: <ul style="list-style-type: none"> • a primary (derived) clock signal (output of a zceiClockPort instance) • a design signal

18.3 Using the Smart Z-ICE During Emulation

During emulation, you can remap an entire Smart Z-ICE port to another one.

There are two possible ways to map Smart Z-ICE.

- Using an environment variable, see [Mapping Smart Z-ICE Using the Environment Variable](#).
- Using the designFeatures command, see [Mapping Smart Z-ICE Using the designFeatures Command](#).

18.3.1 Mapping Smart Z-ICE Using the Environment Variable

To remap a Smart Z-ICE port to another one for emulation runtime, you can use the following environment variable:

`ZEBU_SMARTZICE_<PORT>_PHYSICAL_LOCATION`

where, <PORT> is the identifier of the Smart Z-ICE port instantiated in the design (P0 to P5).

This variable must be set to the value of the port targeted for emulation (P0 to P5).

Using the Smart Z-ICE During Emulation

Example

```
setenv ZEBU_SMARTZICE_P0_PHYSICAL_LOCATION P1
```

In this example, the Smart Z-ICE port P0 is allocated to the physical port P1 for emulation runtime.

18.3.2 Mapping Smart Z-ICE Using the `designFeatures` Command

To remap a Smart Z-ICE port to another one for emulation runtime, you can use the following `designFeatures` command:

```
$smartZICE.connectorRemap_<PORT_NB>
```

where, <PORT_NB> is the number of the instantiated Smart Z-ICE port (0 to 5).

This command sets the number of the port targeted for emulation (0 to 5).

Example

```
$smartZICE.connectorRemap_0 = 2;
```

In this example, the Smart Z-ICE port P0 is allocated to physical port P2 for emulation runtime.

18.3.3 Disabling Smart Z-ICE ports at Runtime

To disable at runtime all the Smart Z-ICE ports that have been enabled during compilation, set the environment variable `ZEBU_SMARTZICE_DISABLE` as TRUE.

A warning message appears as follows:

WARNING: You've compiled for using SmartZice but you've decided not to use it by setting the variable, `ZEBU_SMARTZICE_DISABLE` as OK|YES|TRUE|ON.

19 Appendix

This chapter describes the input files specific to ZeBu for compilation and runtime. Each file format is described by their general syntax and detailed information about their elements.

This section discusses the following topics.

- *designFeatures File*
- *Runtime Clock file*
- *Memory Content File*

19.1 designFeatures File

The designFeatures file is an optional file for the ZeBu runtime software. This file contains user settings that override the default settings for emulation runtime.

The ZeBu runtime uses the designFeatures file if it is present in either the directory where the runtime software is launched.

The path to the designFeatures file can also be provided as an argument to the open method, which is called in the testbench.

If a designFeatures file is not found, the default values are retained for emulation runtime.

The designFeatures file contains the following elements:

- *Location of the Calibration Files*
- *Declaring the Process Name*
- *Parameters for Design Clocks*
- *Parameters for Transactors*
- *Initializing Memories*
- *Programming the driverClk Reset Signal*
- *Programming the DUT Reset*
- *Declaration for Smart Z-ICE*

19.1.1 Syntax Rules

The syntax of the designFeatures file is a set of lines, each containing a parameter setting of the form: <parameter> = <value> ;

The syntax rules for the designFeatures file are as follows:

- Parameter names are case insensitive.
- File names and hierarchical paths are case sensitive.
- Comment lines start with a # character (only at the start of the line).
- <value> can be a decimal number or a string enclosed in double quotes ("").

The emulation runtime software generates a template file, designFeatures.<hostname>.help, in the working directory. This file contains all the settings: user-defined settings declared in the designFeatures file (if present) and

designFeatures File

default settings of non-overridden parameters.

The designFeatures.<hostname>.help file can be modified and renamed as designFeatures to be used in a subsequent emulation runtime session.

Some of the parameters listed in the designFeatures.<hostname>.help must not be modified. They are set by the compilation process or by the emulation runtime software. The following list includes some of these parameters:

- Frequency settings for ZeBu system clocks (masterClk, traceClk, xClk):
\$<sysclock_name>.Frequency
- For SystemVerilog Assertions: \$svaClk
- For Direct ICE: \$directIce.GCLK, \$directIce.vccIo, \$directIce.clock

19.1.2 Location of the Calibration Files

The default calibration files reside in the default directory (\$ZEBU_SYSTEM_DIR/calibration). The path to the calibration files can also be specified using the \$calibrationPath parameter as follows: \$calibrationPath = ?<path>?;

19.1.3 Declaring the Process Name

This section describes how to declare process names in a designFeatures file.

19.1.3.1 Single-process Environment

When the verification environment has a single Linux process, there is no need to declare the process name in the designFeatures file; the default_process name is used.

This default configuration can be viewed in the designFeatures.help template file:

```
$nbProcess = 1;
$process_0 = "default_process";
```

19.1.3.2 Multi-process Environment

When the verification environment has multiple Linux processes, the process names must be declared.

```
$nbProcess = <nb_process>;  
$process_<ID> = "<process_name>";
```

where,

- `<nb_process>` is an integer between 1 and 16 specifying the number of processes.
- `<ID>` is an integer index between 0 and 15. Processes must be declared with contiguous indexes starting from zero.
- `<process_name>` is an arbitrary string used to identify a process in the open method and log files.

For example

```
$nbProcess = 1;  
$process_0 = "myProcess";
```

In the testbench:

```
// open session  
  
Board *board = Board::open ("myZebuWork", "designFeatures",  
"myProcess");  
  
if (board == NULL)  
    exit (1);
```

```
-- ZeBu : cpp_test_bench : Looking for a connection (pid 28200 at  
Tue 3 8 2010 - 17:46:11)  
-- ZeBu : cpp_test_bench : "my_process" is a full-capability  
process working on          ".../zebu.work".
```

designFeatures File

19.1.3.3 Unnamed Control Processes

A C/C++ testbench that does not control any transactor can execute without declaring its process name; it is known as a control process. The name of a control process may be designated in the open method even though the name is not declared in the designFeatures file.

The control process is used to analyze or control clocks, memories, logic analyzers, and so on. However, the control processes cannot control top-level I/O. The clocks generated by zceiClockPort must be driven by named processes.

For example

```
$nbProcess = 2;
$process_0 = "bench_0";
$process_1 = "bench_1";
```

In the testbench:

```
// open session
Board *board = Board::open ("myZebuWork", "designFeatures",
"bench_2");
if (board == NULL)
    exit (1);
```

```
-- ZeBu : tb_dut : WARNING : A process name has been specified
"bench_2" at open time, but it is not specified in the "./
designFeatures" file.

-- ZeBu : tb_dut : WARNING : The list of specified process names
is:

-- ZeBu : tb_dut : WARNING : "bench_0"
```

```
-- ZeBu : tb_dut : WARNING : "bench_1"
-- ZeBu : tb_dut : Looking for a connection (pid 11307 at Wed 9 3
2011 - 09:20:51) ...
-- ZeBu : tb_dut : "bench_2" is a control-only process working on
"../zcui.work/zebu.work".
```

19.1.3.4 Initialization Phases Not Executed on Memory

In a control-only C/C++ testbench, the memory initialization phase is not executed, which may cause the design might malfunction even if there is no error. To avoid this issue, you must identify the testbench in the designFeatures file.

```
$nbProcess = 1;
$process_0 = "<process_name>";
```

19.1.3.5 zRun Unnamed Process

zRun is a particular case of an unnamed process. You cannot connect with **zRun** in standalone mode using a design with transactor declarations. The connection fails with the following error message:

```
-- ZeBu : zRun : ERROR : LUI2007E : In this configuration, zRun
should be used with an
        embedded test bench.

-- ZeBu : zRun : ERROR : LUI2008E : Ports has been detected, they
were declared at
        compilation time.

-- ZeBu : zRun : ERROR : You can try to force "$nbProcess = 0;" in
the designFeatures file.

-- ZeBu : zRun : ERROR : ZHW0909E : Cannot open Zebu: an error
occurred during connection.
```

designFeatures File

To connect with `zRun`, you must set `nbProcess` to 0 in the `designFeatures` file:

```
"$nbProcess = 0;"
```

19.1.4 Parameters for Design Clocks

This section describes the parameters for design clocks in the `designFeatures` file.

19.1.4.1 Parameters for Primary Clocks

Each design clock is described in the `designFeatures` file by its own parameters, and it can be linked to other clocks in a group.

The syntax for clock parameters is as follows:

```
$U0.M0.<my_clock>.Mode = "controlled | free-running | in-situ";
$U0.M0.<my_clock>.Waveform = "_-";
$U0.M0.<my_clock>.Frequency = <my_realFreq>;
$U0.M0.<my_clock>.VirtualFrequency = <my_virtFreq>;
$U0.M0.<my_clock>.GroupName = "<my_group>";
$U0.M0.<my_clock>.Tolerance = "no | yes";
$U0.M0.<my_group>.TimeStampGroup="yes|no";
$DclkTolerance=<my_tolerance_threshold>

$delayClkEdge_<x> = "NbClk <N>
FROM{ [RISE|FALL|BOTH] (<clk_i>) | [RISE|FALL|BOTH] (<clk_i+1>) ... }
TO{ [RISE|FALL|BOTH] (<clk_j>) | [RISE|FALL|BOTH] (<clk_j+1>) ... }";
```

where,

- Mode is "controlled" (clock defined as `zceiClockPort`) when no Mode is defined. "in-situ" is the default value for clocks coming from Smart Z-ICE.
- Waveform is a sequence of characters ("[_-]+") that define the waveform (shape) of the clock. Character sequences should always start with "_". Default is "_-

Starting clocks with a high level generates a rising clock edge at time zero. This is not recommended as it may cause mismatches with simulation. The following warning message is generated by emulation runtime:

```
-- ZeBu : zServer : WARNING : The controlled clock "U0.M0.AAAA"'s
waveform
is "-_ ", it's not recommended to start clocks with a high level.
```

NOTE: *Clock waveforms starting at a low level are strongly recommended to avoid the effects of the invisible rising edge at the start of emulation.*

- Frequency is mandatory for free-running clocks and is not applicable for other types of clocks. <my_real_freq> is the frequency in kHz (float value).
- VirtualFrequency is a ratio between frequencies of clocks belonging to the same clock group. It is only applicable for controlled clocks. Default is one.
- GroupName identifies a clock group. By default, two clocks always belong to the same group. Use this parameter to split them into separate groups. Default is "MyGroup".
- Tolerance optimizes the runtime performance when several primary clocks provided by the ZeBu clock generator are declared in a single clock group. In this case, the order of clock edges is not relevant; only the median frequency and the frequency ratios are considered. For more details, see [Syntax for Tolerance](#).
- delayClkEdge_<x> optimizes runtime performance by raising the minimum number of driverClk edges between the edges of two design clocks. This feature is particularly useful when the strongest timing constraint applies to slow clocks. For more details, see [Syntax for delayClkEdge_<x>](#).

Example of DUT Clock Description in Controlled Mode

When a clock group contains a single clock, there is no need to declare a clock in controlled mode (default mode).

When a clock group contains multiple clocks, you must declare a frequency ratio using the VirtualFrequency parameter or the Waveform parameter. As the clocks are provided by the ZeBu clock generator, it guarantees that a frequency is set as a relative value (VirtualFrequency) or as a Waveform. The following code examples display how to declare a frequency ratio of two using virtualFrequency and Waveform parameters. In this example `clock2` is two times faster than `clock1`.

designFeatures File

To declare a frequency ratio of two using the `VirtualFrequency` parameter:

```
$U0.M0.clock1.Mode = "controlled";
$U0.M0.clock1.Waveform = "_-";
$U0.M0.clock1.VirtualFrequency = 1;
$U0.M0.clock1.GroupName = "clock_group";
$U0.M0.clock2.Mode = "controlled";
$U0.M0.clock2.Waveform = "_-";
$U0.M0.clock2.VirtualFrequency = 2;
$U0.M0.clock2.GroupName = "clock_group";
```

To declare a frequency ratio of two using the `Waveform` parameter:

```
$U0.M0.clock1.Mode = "controlled";
$U0.M0.clock1.Waveform = "_-";
$U0.M0.clock1.VirtualFrequency = 1;
$U0.M0.clock1.GroupName = "clock_group";
$U0.M0.clock2.Mode = "controlled";
$U0.M0.clock2.Waveform = "_--";
$U0.M0.clock2.VirtualFrequency = 1;
$U0.M0.clock2.GroupName = "clock_group";
```

Example of DUT Clock Description in free-running Mode

These clocks are provided by the ZeBu clock generator, and the frequency is fixed.

```
$U0.M0.my_clock.Mode = "free-running";
$U0.M0.my_clock.Frequency = 5000;
```

Example of DUT Clock Description in `in-situ` Mode

These clocks are provided externally, and the frequency cannot be altered.

```
$U0.M0.my_clock.Mode = "in-situ";
```

Syntax for Tolerance

This parameter can be used to improve runtime performance when multiple primary clocks belong to the same clock group. The performance improvement is more noticeable when:

- The number of primary clocks is higher.
- The declared clock waveforms and their ratios are heterogeneous.
- The clock edges that are very close to each other are processed simultaneously, group-by-group or clock-by-clock.

To activate the `Tolerance` parameter, the following lines must be added for each group/clock, which needs to be modified:

```
$U0.M0.<group>.Tolerance=<value>;  
$U0.M0.<clock>.Tolerance=<value>;
```

where `<value>` is a string, enclosed by double quotes, with the following values:

- "no": Default.
- "yes": Clock edges that are very close to one another are merged.

Note

- *Tolerance can be used only with controlled clocks.*
- *At least two clocks in the same group are required for an effective calculation.*
- *Only yes/no attributes are accepted.*

All clocks in a clock group have the same tolerance. However, the tolerance of a single clock in the group can be overwritten, as displayed in the following example:

```
#Group setting for group1  
$U0.M0.group1.Tolerance = "yes";  
#CLK0 uses the group setting for tolerance  
$U0.M0.CLK0.VirtualFrequency = 8;  
$U0.M0.CLK0.GroupName = "group1";
```

designFeatures File

```
# CLK1 overwrites the group setting to disable tolerance
$U0.M0.CLK1.VirtualFrequency = 5;
$U0.M0.CLK1.GroupName = "group1";
$U0.M0.CLK1.Tolerance = "no";
# CLK2 uses the group setting for tolerance.
$U0.M0.CLK2.VirtualFrequency = 2;
$U0.M0.CLK2.GroupName = "group1";
```

If a clock and a clock group share the same name, the resultant ambiguity in the tolerance parameter cannot be resolved and the runtime exits with the following error message:

```
"LUII1990E" : "Ambiguous designFeatures declaration :
$U0.M0.clk.Tolerance = "yes"
"A Clock Name and a Group Name are the same (clk) : "Tolerance" can
not be attributed"
```

Syntax for `delayClkEdge_<x>`

The `delayClkEdge_<x>` parameter can be specified to improve the runtime performance by increasing the minimum number of `driverClk` edge transitions between subsequent edges of two design clocks. It designates a timing constraint, at least one source clock edge, and one target clock edge. The timing constraint defines the minimum number of `driverClk` edge-transitions between subsequent edges of two design clocks. The syntax for this parameter is:

```
$delayClkEdge_<x> = "NbClk <N>
FROM{ [RISE|FALL|BOTH] (<clk_i>) | [RISE|FALL|BOTH] (<clk_i+1>) ... }
TO{ [RISE|FALL|BOTH] (<clk_j>) | [RISE|FALL|BOTH] (<clk_j+1>) ... }";
```

where,

- <x>: A number that identifies the constraint, ranging from 0 through 15.
- <N>: The minimum number of driverClk edge transitions occurring between the subsequent edges of the two design clocks, ranging from 0 through 31.
- RISE (<clk_i/j>): Timing constraint is applied to the rising edge of <clk_i/j>.
- FALL (<clk_i/j>): Timing constraint is applied to the falling edge of <clk_i/j>.
- BOTH (<clk_i/j>): Timing constraint is applied to both edges of <clk_i/j>.

Note

- delayClkEdge_<x> can be used only with controlled clocks.
- A vertical bar (|) is used to designate a list of clock edges.
- The edges of the source clocks are preceded by a FROM indicator and placed between curly brackets ({ }). Similarly, the edges of the target clocks are preceded by a TO indicator and placed between curly brackets (}).
- You can also use an asterisk (*) to designate all design clocks and all design clocks must belong to the same clock group.

For example

For a minimum of four driverClk edge transitions between the rising edge of CLK0 and the rising edge of CLK1:

```
$delayClkEdge_0 = "NbDriverClk 4 FROM{ RISE(CLK0) } TO { RISE(CLK1) }";
```

For a minimum of three driverClk edge transitions between the rising edge of CLK0 or the falling edge of clkfast, and the rising edge of CLK1 or any edge of CLK3):

```
$delayClkEdge_0 = "NbDriverClk 3 FROM{ RISE(CLK0) | FALL(clkfast) } TO{ RISE(CLK1) | BOTH(CLK3) }";
```

For a minimum of three driverClk edge transitions between the rising edge of any clock belonging to the same clock group and the rising edge of any clock belonging to the same clock group:

```
$delayClkEdge_0 = "NbDriverClk 3 FROM{ RISE(*) } TO{ RISE(*) }";
```

designFeatures File

Using a Separate Clock File

For easier reading of the designFeatures file, the clock parameters can be specified in a separate file, as described in [Runtime Clock file](#).

A clock file is specified in the designFeatures file, as follows:

```
$U0.M0.<clock_name>.File = "my_clock_file";
```

Declaring Additional Clocks

You can specify a dummy clock to monitor oversampling. The syntax is:

```
$newClock = "U0.M0.<my_dummy_clock>";
```

Declaration of parameters for <my_dummy_clock> are the same as for primary design clocks (see [Parameters for Primary Clocks](#)).

When a dummy clock is mapped, the following message is displayed:

```
-- ZeBu : zServer : The new clock "U0.M0.d_clk1" will be mapped on
clock 1.
```

These additional clocks also count towards the maximum number of clocks provided by the clock generator.

If the maximum number of primary clocks is exceeded, the declaration of any additional clock displays the following error message:

```
-- ZeBu : zServer : ERROR : LUI1433E : The current ZeBu clock
generator supports 2 primary clocks.

-- ZeBu : zServer : ERROR : LUI1433E : Your design already uses 2
primary clock(s).

-- ZeBu : zServer : ERROR : LUI1433E : You cannot register any new
user clock on this clock generator.

-- ZeBu : zServer : ERROR : LUI1433E : You can increase the number
of clocks with the UTF command clock_config -clock_number <2
|4|8|16>. This requires a recompile of the design.
```

The maximum number of clocks supported by the clock generator can be modified in the UTF project. After making any changes to the UTF project, you must recompile the design.

Propagation Delay

The clock-path synchronizers used by the ZeBu clock generator are programmable; the propagation delay can be programmed as follows:

```
$FKpropagationDelay = <my_FKpropagationDelay>;
```

where `my_FKpropagationDelay` is an integer between 0 and 15 (default is 2), corresponding to the number of system clock periods.

If the `$FKpropagationDelay` is present and the synchronizers are not required, the runtime setting is ignored and the following message is displayed:

```
$FKpropagationDelay specification is ignored (synchronizers are  
not used)
```

Offset Delay for Clock Skews

A delay can be used to offset clock skews. This delay is programmed using the following parameters:

- `zClockFilterTime` (ns) is the delay of the longest path from a clock cone (logic generating the clock signal) entry to a filter. The filters are locked within this delay and all filters are opened at the same time after the specified delay.
- `zClockSkewTime` (ns) = `zClockFilterTime` + Δ , where Δ is the delay of the longest path from a filter to a sequential element.

Both parameters must be specified to control the glitch-filtering algorithm.

Note

`zClockFiltertime` must be smaller than `zClockSkewTime`.

 designFeatures File

The ZeBu compiler estimates both parameters during Static Timing Analysis (`zTime`) and communicates them to the runtime through the `driverClock` frequency.

```
$zClockSkewTime = <my_skew>;
$zClockSkewSel = "driverClk | allUserClk | userClkMask=<mask>";
$zClockSkewMode = "pulse | level";

$zClockFilterTime = <my_filter>;
$zClockFilterSel = "driverClk | allUserClk | userClkMask=<mask>";
$zClockFilterMode = "pulse | level";
```

where,

- `<my_skew>` and `<my_filter>` are integers between 0 and 255 corresponding to the number of system clock periods (default is 40).
- `<mask>` is a 16-bit integer pattern that enables the 16 user clocks. The positioning of the clocks in the clock mask is forced by the clock index in `zebu.work/U0.M0/rtb.xref` file.
- The default for `$zClockSkewSel` and `$zClockFilterSel` is `driverClk`. It is recommended to retain this clock reference.
- `$zClockSkewMode` default value is `pulse`.
- `$zClockFilterMode` default value is `level`.

For more information about clock modeling, see *ZeBu Quick Start Guide*.

Declaring a Timestamp Group

On Zebu Server-4, if users declare several clock groups, it is mandatory to declare one (and only one) of these as the Timestamp group.

19.1.5 Parameters for Transactors

This section describes the transactor parameters declared in the designFeatures file.

19.1.5.1 Global Transactor Settings

By default, the communication infrastructure for transaction-based verification uses a burst mode that is applied to all the ports of a transactor.

The following declaration can be used to modify the settings for the burst mode:

```
$xTor.burstMode = "high | medium | disable";
```

By default, the high burst mode is active, if the host computer supports it.

If the host computer does not support burst mode, the ZeBu runtime (**zServer**) automatically disables it (equivalent to disable value).

In such a case, try the medium value since it may improve the communication performance for large messages. For small-size messages, the disable value is recommended.

19.1.5.2 Specific Transactor Settings

The performance of transactor ports does not depend only on the port sizes. It is because big ports are used rarely while small ports are used more often. For this reason, the size of the buffer that receives port data from the hardware can be modified according the number of messages received in the buffer.

Performance is optimized when a frequently active port has multiple port values (messages) in its buffer. The current `$<driverName>::<portName>.nbMessMax` is given in the `designFeatures.<hostname>.help` file. The `nbMessMax` parameter specifies the maximum number of port-values (messages) to be stored in the buffer.

The `nbMessMax` setting is specific to each transactor port and it must consider the transactor's memory requirements:

```
$<driverName>::<portName>.nbMessMax = <value>;
```

designFeatures File

19.1.6 Initializing Memories

This section describes how to initialize memories in the designFeatures file.

19.1.6.1 List of Memory-Content Files

A file containing a list of design memory instances that must be initialized with a specific content can be specified using the \$memoryInitDB parameter. Each memory is initialized with its corresponding content-file. This initialization is applicable to any memory, irrespective of its physical implementation (LUTs, registers, BRAM, and so on).

This specification is done as follows:

```
$memoryInitDB = "path_to_memory_init_list";
```

The path can be absolute or relative to the current directory. The memory load occurs in the Board::open() function.

For example

Declaration of the file for memory initialization:

```
$memoryInitDB = "~/mydesign/init_mem1.mem";
```

Contents of init_mem1.mem:

```
my_mem1    ~/mydesign/memories/mymem1.txt
my_mem2    ~/mydesign/memories/mymem2.txt
```

Disabling Transactional Communication for zrm Memories

By default, zrm memories are non-transactional. However, if transactional mode is enabled at compile time, it can be disabled at runtime using the following designFeatures parameter:

```
$zrmTransactionalMode = 0;
```

19.1.7 Programming the driverClk Reset Signal

A programmable reset signal driven by `driverClk` is available for transaction-based verification. The duration of the reset state is defined as the number of `driverClock` cycles. It should only be set when a transactor BFM requires multi-cycle assertions of reset.

```
$driverClk.Reset = <value>;
```

where `<value>` is the number of `driverClk` cycles for which reset is active (Default: 0).

Note

It is recommended not to specify the parameter (comment with #). But, for transactors developed with ZEMI-3, you must set it to a non-zero duration: \$driverClk.Reset = 1;

19.1.8 Programming the DUT Reset

You can define a reference clock to program the DUT reset. The active level of the reset is derived from the selected output of the `zceiClockPort` instantiation, that is, `reset` (active high) or `reset` (active low).

The waveform of the reset signal can be programmed using the following parameters:

```
$tgClk.ResetInactive = <nb_inactive_cycles>;
$tgClk.ResetActive = <nb_active_cycles>;
$tgClk.SelectResetClkName = "U0.M0.<my_clock>";
```

19.1.9 Declaration for Smart Z-ICE

For a design using the Smart Z-ICE interface, the following line should be added to relocate a Z-ICE connector at runtime:

```
$smartZICE.connectorRemap_<i> = <j>;
```

where,

- `<i>` is the compilation index of the Smart Z-ICE connector.

Runtime Clock file

- <j> is the index of the actual Smart Z-ICE connector at runtime.

Only the connectors of the unit connected to the host computer can be used with the Smart Z-ICE interface.

For example

When the design is compiled for connectors 0 and 1 but the connectors actually used are connectors two and three, add the following in the designFeatures file:

```
$smartZICE.connectorRemap_0 = 2;
$smartZICE.connectorRemap_1 = 3;
```

The following declarations are present in the designFeatures.<hostname>.help file:

```
$smartZIce.data
$smartZIce.vcc
$smartZIce.clock
```

Comments are automatically set from the compilation and should not be modified in the actual designFeatures file.

19.2 Runtime Clock file

The clock file is an optional input to the Zebu runtime. It is an alternative to specifying the clocks inside the designFeatures file. For information about specifying the clock parameters in a separate file, see [Using a Separate Clock File](#).

19.2.1 Clock File Content

A separate clock file accepts the same parameters as the clock parameters of the designFeatures file (see [Parameters for Design Clocks](#)).

19.2.2 Clock File Syntax

The clock file is a text file whose syntax is described by the following Backus Naur Form (BNF):

```
file_txt :  
    line_txt  
    | file_txt line_txt  
  
line_txt :  
    '\n'  
    | mode '\n'  
    | waveform '\n'  
    | virtual_frequency '\n'  
    | frequency '\n'  
    | group_name '\n'  
    | Tolerance '\n'  
    | delayClkEdge_<x> '\n'  
  
mode :  
    '$'CLOCK_NAME.mode '=' """mode_type""";'  
waveform :  
    '$'CLOCK_NAME.Waveform '=' """WAVEFORMSET""";'  
virtual_frequency :  
    '$'CLOCK_NAME.VirtualFrequency '=' NUMBER ';' ;  
frequency :  
    '$'CLOCK_NAME.Frequency '=' NUMBER ';' ;  
group_name :  
    '$'CLOCK_NAME.GroupName '=' """STRING""";'  
mode_type :  
    controlled  
    | free-running
```

Runtime Clock file

```

    | in-situ

Tolerance_declaratiion :
    '$'GroupName.Tolerance '=' """toleranceSpecifier""';'
    | '$'CLOCK_NAME.Tolerance '=' """toleranceSpecifier""';'

toleranceSpecifier :
    yes
    | no

delayClkEdge_<x> :
    "NbDriverClk" NUMBER "FROM" '{' list_of_edges '}'
                    "TO" '{' list_of_edges '}';'

list_of_edges :
    edgeDeclaration
    | list_of_edges, edgeDeclaration

edgeDeclaration :
    '''edgeSpecifier''' '(' '''clockIdentifier''' ')'
    | '''edgeSpecifier''' '(' '*' ')'

edgeSpecifier :
    RISE
    | FALL
    | BOTH

```

where,

- Terminal token **NUMBER** has the form: [0-9] +
- Terminal token **WAVEFORMSET** has the form: [-] +
- Terminal token **CLOCK_NAME** is the full path to the clock derived from zceiClockPort

- delayClkEdge_<x>:<x> = [0...15]

19.3 Memory Content File

Memory content files are used to initialize, upload, or download memories. Memory content files work with any memory, irrespective of its physical implementation (LUTs, registers, BRAM, and so on).

19.3.1 ZeBu-Proprietary Binary Format

The ZeBu-proprietary binary format for memory content files is not readable. It can be created through a dedicated interface, **hex2bin**.

It is possible to have a 24-character header at the beginning of the binary memory content file to describe the initialization address range. This file is commonly named with the **.bin** extension (you must not use **.raw** extension, which is reserved for the binary file with no header).

When a binary file has no header (binary file generated out of emulation, or an executable file to be downloaded into a ROM), the file name must be named with the **.raw** extension for automatic detection at runtime.

19.3.2 Memory Binary Format for Shorter Load Time

A new binary sparse memory file format enables you to reduce the time taken for loading the memories.

The new compact files can be generated with the **hex2bin** tool and can be loaded using **zRun** Tcl, C++ API or designFeatures.

To generate the compact files, the following three new options are included in **hex2bin**:

- **-s**: Generates a sparse memory file.
- **-f N**: If it is used with **-s**, fills address gaps smaller than N-words with 0's.
- **-a N**: If it is used with **-s**, aligns writes to N-byte boundary.

Example

```
hex2bin -i hex.0.0.0 -o bin0.0.0 -m top.dut.mem00 -p ../zcui.work/
zebu.work -s -f 1000 -a 64
```

In addition, **hex2bin** proposes a new option to accept when the input file content goes beyond the last address of the memory.

- -t: Do not generate an error if the content of the input file goes beyond the last address of the memory

19.3.3 Memory Content File Text Format

The text format of a memory content file is a description of the memory content for each address or address range. Each line starts with the address or the address range and lists the corresponding data as follows:

```
@<address> : <data>
```

Or

```
@<start_addr>,<stop_addr> : <data>
```

where,

- The colon that separates address and <data> is optional.
- The comma between <start_addr> and <stop_addr> is mandatory.
- The values for <address> and <data> are specified in hexadecimal radix ('h, 'H or 0x before the value), binary radix ('b or 'B before the value) or decimal radix ('d or 'D before the value).
- The default radix for address and data values is hexadecimal.
- The default radix for the subsequent <data> values can be specified by adding only the radix specifier on a line, as in the following example (switch to decimal radix):

```
'd
```

- The default radix for subsequent <address> values can be changed by adding @ with the radix specifier on a line, as in the following example (switch to decimal mode):

```
@'d
```

- The maximum addressable size is 32 bits.
- The maximum data size is 4,096 hexadecimal digits. Data longer than 4,096 digits is truncated and only the 4,096 most significant digits are loaded.
- The maximum limit for initializing data in decimal is 32-bit.
- When no address is specified, the value corresponds to the address following the previous address. If no explicit address is given at the beginning of the memory content file, data is written starting from address 0.
- In range mode, only one data can be set on each line of the file.
- If an address is given more than once, the last value for this address is used (this can be used, for example, to initialize a memory with a value and specify values for particular addresses).
- x, x, z and z values are accepted for <data> and written as 0 in the memory.
- Multiple data can be listed in the same line, separated by blanks or tabs.
- Underscore characters ("_") can be used in data word to improve legibility (for example 12345678 can be written 1234_5678).
- C-like line comments start with "//" and end with a line break.

19.3.3.1 Example

The following file initializes a 128-byte memory (16-bit words):

```
// Initialize all the memory with value 0x44
@0, FFFF : 44

// Load address 0 with value 0xBA00
@0 : ba00

// Load address 2 with 0xBB02
@2 : BB02

//Load address 3 with 0xCC03 and address 4 with 0xDD04
CC03
DD04

// Switch to binary mode by default for data
'b

// Load address 5 with 0x1234 in binary with legibility separator
@5: 16'b0001_0010_0011_0100

// Load address 6 with a decimal value (will be 0x10 in the memory)
16

//Switch to hexadecimal by default
'h

//Load address 7 with some undetermined bits (z)
@7 0x01z7

// Load range from address FF00 to FF03 with value 0xA
@FF00,FF03: A
```

The actual content of a memory initialized with this example file is described in the following table:

Address	Value
0000	BA00
0001	0044
0002	BB02
0003	CC03
0004	DD04
0005	1234
0006	0010
0007	0107
.....
FEFF	0044
FF00	000A
FF01	000A
FF02	000A
FF03	000A
FF04	0044

20 zFmCheck

zFmCheck is a tool to validate that the ZeBu synthesis engine has correctly synthesized Verilog modules, Very Hardware Description Language (VHDL) entity, or architecture pairs. **zFmCheck** identifies any problems in the synthesized design between the reference and implementation models using Formality Equivalence Checking and VCS simulation technologies.

In the **zCui** compile directory, **zFmCheck** uses the EDIF output as the implementation model and the decompiled Verilog or VHDL as the reference model.

In this chapter, the term “model” is used to refer a Verilog module, a VHDL entity or an architecture pair.

This section describes the following topics.

- *Preparation*
- *Formal Equivalence with Formality*
- *Formality Results*
- *Constrained Random Testbench Simulation with VCS*
- *Simulation Results*
- *Advanced Simulation Options*
- *Using zFmCheck for Large Design*
- *Best Practices and Further Helpful Uses*
- *zFmCheck Use Model*

20.1 Preparation

To use **zFmCheck**, add the following UTF option before compiling a design in ZeBu:

```
synthesis -generate_db_for_fmcheck true
```

After the design is synthesized, **zFmCheck** can be invoked from the compile directory.

Note

- *If you only want to prove equivalence and not to build the entire design, add the -d option with the **zcui** invocation command to run just through the front-end.*
 - ***zFmCheck** can be invoked when you see that the front-end is complete. After synthesis executes, it means that the front-end has completed.*
-

20.2 Formal Equivalence with Formality

To use Formality for equivalence checking, use the **-r** argument to the **zFmCheck** command.

```
zFmCheck -r -d zcui.work
```

This is the simplest form of the command and is only meaningful for small examples that do not require significant compute intensive resources. Additional options should be used for designs with many models.

This option instructs **zFmCheck** to prepare the reference and implementation models for use with the Formality tool. In addition, match files are created for use by Formality.

20.3 Formality Results

The Formality results are saved within the `fm_edf` subdirectory and are summarized at the completion of the `zFmCheck` command. Each model has a log of the Formality run captured in the `fm_edf/job_<model>.log` file. This file is accessed to determine the state of models as follows:

- **Equivalent:** It indicates that the reference and implementation models match according to Formality. Synthesis is correct for this set. You can find the list of models within `fm_edf/equivalent.log`.
- **Different:** It indicates that the reference and implementation models do not match according to Formality. Synthesis is not correct for this set. You can find the list of models within `fm_edf/different.log`.
- **Unknown:** It indicates that Formality is unable to conclude a result for the given reference and implementation models. These results are further categorized into other sets. You can find the complete set of models within `fm_edf/unknown.log`.
 - **Timeout:** It indicates that Formality is unable to conclude in the given period. Often, this period is too short for a typical Formality proof and so simply increasing the time limit substantially on a subsequent iteration proves equivalent or different. You can find the models that fall into this category in `fm_edf/unknown_timeout.log`.
 - **Ref Setup Fail:** It indicates that Formality tool itself was unable to handle the reference model. Often, this occurs because of a bug in Formality or simply because there are language constructs that the Formality tool is unable to handle. A recent example of this is the SystemVerilog if in an event control. Models that fall in to this category can be found in `fm_edf/unknown_ref_setup_fail.log`. The only way to show results for the unknowns of this type is through constrained random testbench simulation.
 - **Multiply Driven:** It indicates that Formality tool was unable to handle the comparison due to multiply driven signals. Sometimes, you know multiple drivers present in a part of the code (for example, transactor) for which they are tolerating this. Otherwise, it is not acceptable. You can find the models that fall into this category in `fm_edf/unknown_multiply_driven.log`. The multiply driven signals are reported in `fm_edf/multi_driven_<model>.log`.
 - **Verify Inconclusive:** It indicates that Formality was unable to converge and stopped in the given period. You can find the models that fall into this category in `fm_edf/unknown_verify_inconclusive.log`.

- **Unmatched Point:** It indicates that **zFmCheck** was unable to provide an appropriate match point between the reference and implementation models. The models that fall into this category might be correctly handled in constrained random testbench simulation. You can find the models that fall into this category at fm_edf/unknown_unmatched_point.log.
- **Others:** It indicates that **zFmCheck** is unable to deduce why Formality is unable to prove the equivalence. You can find models that fall in to this category at fm_edf/unknown_others.log.
- **Blackbox:** It indicates that Formality sees the model as empty. Models of this type can be found in fm_edf/blackbox.log.
- **Unverified:** It indicates that Formality did not try to verify the model although it was requested to. Models of this type can be found in fm_edf/unverified.log. When the model is encrypted and the encryption method cannot be supported, **zFmCheck** moves the model into this category.
- **Skipped:** It indicates that **zFmCheck** has detected that the model or entity or architecture pair cannot be proved by Formality and so it is skipped. This can occur for certain memory types. Instead of Formality, constrained random simulation should be used on these models. Models of this type can be found in fm_edf/skipped.log.

20.4 Constrained Random Testbench Simulation with VCS

To use constrained random testbench simulation, use the `-v` argument to the `zFmCheck` command.

```
zFmCheck -v -d zcui.work
```

This is the simplest form of the command and is only meaningful for small examples that do not require significant compute intensive resources. Additional options should be used for designs with many models.

Using this option instructs **zFmCheck** to generate a constrained random testbench with an instance of the reference and implementation models for validation. Input stimulus from randomly generated data and output data is asserted to match between the reference and implementation models. Simulation is not exhaustive and concludes after a default set of cycles.

20.5 Simulation Results

Simulation results are saved in the simulation subdirectory and a summary is displayed on the screen at the completion of **zFmCheck**. Each model has its simulation compile and runtime log captured within simulation/logs/<model>_*.log. These files are often accessed to determine what occurred specific to that model.

- **Pass:** This result indicates that simulation has successfully simulated the set of random data and found no differences between the reference and implementation models. You can find the models that fall into this category within the simulation/sim_pass.log file.

NOTE: *A pass in simulation does not mean that the model is correctly synthesized. It simply means that applying the set of random stimuli for the defined set of cycles detected no difference between the reference and implementation models.*

- **Fail:** This set indicates those models that failed during simulation by comparing the reference and implementation results. You can find the models that fall into this category at simulation/sim_fail.log. Failures in this category are concerning and could mean there is a problem in synthesis.

- **SYN_SIM_MISMATCH:** The SYN_SIM_MISMATCH means there are some synthesis and simulation mismatch in the model. The mismatch reasons are as follows:

- ◆ SAME_RESET_SIGNAL
- ◆ TREAT_PROCESS_COMPLETELY_SENSITIVE
- ◆ LATCH_RACE_CONDITION
- ◆ CONVERT_NONBLOCKING_TO_BLOCKING_IN_FUNCTION
- ◆ VDHL_EXPLICIT_ENUM_ENCODING
- ◆ INDEX_RELAXATION_ENABLED
- ◆ NON_BLOCKING_ASSIGNMENT_INSIDE_TASK_OR_FUNCTION

- **Error/Missing:** This set indicates that there was an error in trying to produce the simulation executable. The issue could be in testbench creation, reference model compile, implementation model compile, and design elaboration. To see the models from this category, see the simulation/sim_error.log file.

- **Runtime error:** This set indicates that there was an error in running the simulation. The errors in this category typically indicate a problem with the

constraints. Often, the models are not able to handle the random stimulus on the input to the model whereas in the real design, the stimulus is further constrained. Errors in this category need investigation to determine if it is a real issue with the model or simply the provided stimulus is too vast. To see the models from this category, see the simulation/sim_runtime_error.log file.

- **Timeout:** This set indicates that the simulation is unable to conclude within the given timeout. You can typically increase the timeout to conclude on these, but sometimes, the longer timeout does not help. You can find the list of models that fall into this category within the simulation/sim_timeout.log file.

20.6 Advanced Simulation Options

When constructing a simulation testbench, simulation aspects such as clock frequency, cycles to run, and register initialization can be set through **zFmCheck**. Following is a sampling of some of those options and what they control:

- `-initcycle N`: Specifies the number of initial cycles for simulation. Use this to determine how long you want the simulation to run in the initial state.
- `-gl`: Gate level memory simulation. The state machine in the gate level ensures the last written part take effect.
- `-xprop <1/0>`: Enables or disables `-xprop` in simulation.
- `-freq`: Specifies the user clock frequency in KHz for random simulation.
- `-cycle N`: Specifies the number of cycles to run the random stimulus.

20.7 Using zFmCheck for Large Design

For a large design, run **zFmCheck** in multiple iterations to find the set of models that need attention to a minimal set in the following sections:

- *Initial Iteration (With or Without -v)*
- *Subsequent Iterations*
- *Timeout Iteration*

Initial Iteration (With or Without -v)

In the first iteration, minimize the concerning models and give any details into identifiable issues. In the first iteration, you should use the `-r`, `-j`, `N`, and `-c`

Using zFmCheck for Large Design

“cmd” options with **zFmCheck**. You might also want to add **-v** to automatically perform simulation on the set that is not equivalent in Formality.

To distribute the jobs in parallel, use **-j**. For example, use **-j 100** with a design that has 200 models and runs 100 processes where each job contains 2 models to process.

To distribute the processing onto the grid, use **-c “cmd”**. For example, use **-j 100** with a design that has 200 models and runs 100 invocations of the **-c** option. In addition, it distributes all 100 processes onto 100 machines where each machine processes 2 jobs per machine.

Formality runs on the entire design and then a summary is provided for all blocks that Formality (**-r**) was able to prove. For items that Formality is not able to prove equivalent, a constrained random testbench is created to stimulate with random simulation (**-v**). A simulation summary is also provided for the set of simulated models.

```

prompt> zFmCheck -r -v -j 10 -c "grid_cmd" -d zcui.work
#####
#Formality summary#####
EQUIVALENT:      574 (98%)
DIFFERENT:       3 (1%)
UNKNOWN:        9 (2%)
|--REF SETUP FAIL:      3
|--MULTIPLY DRIVEN:     1
|--VERIFY INCONCLUSIVE:  1
|--OTHERS:            4
BLACKBOX:        2 (0%)
UNVERIFIED:      0 (0%)
SKIPPED:         0 (0%)
-----
588
#####
#Simulation summary#####
Modules: 12
PASS: 8
FAIL: 1
ERROR/MISSING: 3
RUNTIME-ERROR: 0
#####

```

In this case, 588 models were processed and 574 were proven correct by Formality. The remaining 14 models, 2 are black boxes. It means, there is really nothing in those 2 models. The final 12 models are passed to constrained random simulation using VCS. In the 12 models, consider 8 models are passed, 1 failed, and 3 are not able to prove due to errors in compile or missing results.

Subsequent Iterations

Subsequent iterations are typically used to find the concerned models.

The following are some typical iteration to explain subsequent iteration set: (all are on grid with the large -j option)

3. Run `zFmCheck -r` with large timeout for those models that timed out. This is described in the [Timeout Iteration](#) section.
4. Run `zFmCheck -v` with mid-size timeout that are not equivalent.
 - a. Set the `-j` option to the remaining set, if possible or some evenly distributed fraction of 400 models left
 - b. Run `-j 100` to get 4 models per job.
5. Run `zFmCheck -r -v` with the UC1 results with the remaining models.
 - Compare the differences between UC1 and UC2 (VCS) to quickly identify any differences that need attention.

Timeout Iteration

For those models that Formality fails to prove due to timeout, launch **zFmCheck** that focuses on such models. For this, use the `-l` option to increase the default timeout (that is, `-l 24:00:00` to specify a run for at least 24 hours). You can see the results on single models after running for 14 hours. Some models might never converge in Formality. This iteration should minimize the results even further.

```
zFmCheck -l 12:00:00 -mf zcui.work/design/synth_Default_RTL_Group/
formality/fm_edf/timeout.log -r -d zcui.work -j 30 -c "grid_cmd"
```

Assuming a functional grid with 30 machines and 30 models that timed out, the command runs just those models that timed out in Formality for 12 hours. The results should be available in just over 12 hours.

Note

Use the same number of jobs (-j) as models that timed out (wc -l timeout.log) to get the optimal distribution. If the grid is not large enough for the number of models that timed out, use a fraction of the time based on when you want to see results.

For example, if you want results in 12 hours (next morning) and you have 100 models to validate with a grid of only 25 machines, use -j 25 with a timeout of 3 hours (1/4 of 12 hours). Since the number of machines (25) is only a quarter of the timeouts, it should complete the analysis of the 100 within 12 hours given 4 models per job.

After this set completes, you might have some that still timed out, but you should likewise have many that did not timeout.

20.8 Best Practices and Further Helpful Uses

This section discuss the best practices for using **zFmCheck** in the following subtopics:

- [Organize Your Results](#)
- [Using -norm to Keep Simulation Data](#)
- [Using DVE to Debug a Simulation Failure](#)
- [Using Verdi to Debug a Simulation Failure](#)
- [Using -save du to Launch Formality GUI](#)
- [Why Use -scm Option?](#)
- [Checking Status of a zFmCheck run](#)

20.8.1 Organize Your Results

When working on a large design, **zFmCheck** is invoked multiple times. Each execution of **zFmCheck** is important to keep a track of results and examine the results in the future. Some **zFmCheck** runs encompass a single zcui.work directory and some encompass many zcui.work directories. The results must be organized according to the expected usage.

You must adhere to the following rules when starting to analyze a full design:

1. Delete the zcui.work directory because the space occupied by zcui.work is huge and the space is required for a different compile.
2. Do not store the results directly in zcui.work. Instead, use the -g option to place the results in an area that has ample space and is not within zcui.work.

NOTE: By default the results are stored within zcui.work.

3. Use a fully qualified path name with the -g or -o option. This is stored within the results and helps you to know exactly where the results are stored. The -o option uses the current path as the relative path. For example, -g 'pwd' / fm is same as -o fm.
4. Use a name for your output directory to indicate what you are doing. Following is an example of iterations that occur:

```
fm_r           #Run zFmCheck with just -r(no -v) on entire
design
```

```
fm_r_timeouts   #Run zFmCheck with just -r(no -v) on design
models that timed out.
```

```
fm_v_348         #Run with just -v on a set of 348. This implies
there is 348 models left.
```

```
fm_r_v_120_in_uc1 #Run with both -r and -v on the remaining 120
unknowns with UC1.
```

This naming helps to indicate what has occurred.

20.8.2 Using -norm to Keep Simulation Data

When executing zFmCheck with -v, use -norm to preserve all the data needed to produce the simv (VCS) executable for each module.

Example:

```
zFmCheck -g `pwd`/fm -v -norm -m dut -d zcui.work
```

The fm/simulation/sim_dir/dut/*sh file compiles all the necessary files and runs simv. You can execute this directly to re-create the **zFmCheck** run.

The fm/simulation/sim_dir/dut/simon_dut_tb.* file is the generated testbench for simulating the reference and implementation models.

Note

*If you want to debug a simulation failure, modify the *sh file to comment the testbench generation, modify the existing testbench to (insert an FSDB dump) and then rerun the *sh file.*

20.8.3 Using DVE to Debug a Simulation Failure

To launch a waveform debugger on the simulation run, there are multiple ways to do it.

To launch the waveform debugger from the zFmCheck command line, add `-debug` to the VCS command line and then the `-gui` option to the `simv` execution as follows:

```
zFmCheck -v -opt_simv "-gui" -opt_vcs "-debug" -d zcui.work -m  
model_name -norm
```

Where, `-norm` retains all the simulation data used to produce `simv` for the model named “`model_name`”.

20.8.4 Using Verdi to Debug a Simulation Failure

To debug the waveform of a failed simulation module, use Verdi as follows:

```
zFmCheck -v -verdi_debug -d zcui.work
```

This command generates Verdi command files under simulation/scripts named as `${model_name}_verdi_cmd.sh`.

To perform debug, launch Verdi using the following steps:

1. Navigate to the simulation/sim_dir/model/ directory.
2. Execute `${model_name}_verdi_cmd.sh`.

20.8.5 Using -save du to Launch Formality GUI

To use the formality GUI on a given module, make sure you have `-save du` on the zFmCheck command line. You can now directly launch Formality using the following:

```
# Replace bundle.tcl and mod with the specific one for your case.
```

```
fm_shell -f bundle.tcl -x "set Module mod"
```

Now, from the fm_shell prompt, type start_gui.

20.8.6 Why Use -scm Option?

To determine if the EDIF results of combined VCS (simon) and **zFast** produced a correct model, use zFmCheck -r.

To determine if VCS (simon) specifically (no **zFast**) has an issue in synthesis according to Formality equivalence checking, additionally use zFmCheck -scm.

If a model result is DIFFERENT in Formality from the -r option but is EQUIVALENT with the -scm option, it indicates an issue in the **zFast** optimizer.

20.8.7 Checking Status of a zFmCheck run

To check the status of a **zFmCheck** run, use the -p option. A summary of the results is displayed and *formality.log* is updated for the selected directory.

```
zFmCheck -p -g fm_r
```

20.9 zFmCheck Use Model

To print the use model and help message, use zFmCheck -h.

The following is an example use model of **zFmCheck**:

```
zFmCheck -d synthesis_directory [-g output_directory] [-j  
number_of_jobs] [-c remote_command] [-m module_names | -mf  
module_list_file] [-t top_module_name]
```

```
zFmCheck -d synthesis_directory [-g output_directory] [-j  
number_of_jobs] [-c remote_command] [-m module_names | -mf  
module_list_file] [-t top_module_name] [-k] [-f formality_script] [-s  
suppress_messages] [-l hh:mm:ss]
```

```
zFmCheck -d synthesis_directory -r [-g output_directory] [-j  
number_of_jobs] [-c remote_command] [-m module_names | -mf  
module_list_file] [-t top_module_name] [-k] [-f formality_script] [-s
```

zFmCheck Use Model

```

suppress_messages] [-l hh:mm:ss]

zFmCheck -r -v -scm -d synthesis_directory [-t top_module_name] [-g output_directory]

zFmCheck -v -d synthesis_directory [-t top_module_name] [-g output_directory]

zFmCheck -v -scm -d synthesis_directory [-t top_module_name] [-g output_directory]

zFmCheck -r -v -d synthesis_directory [-t top_module_name] -gl [-g output_directory]

zFmCheck -r -v -scm -d synthesis_directory [-t top_module_name] -gl [-g output_directory]

zFmCheck -v -d synthesis_directory [-t top_module_name] -gl [-g output_directory]

zFmCheck -v -scm -d synthesis_directory [-t top_module_name] -gl [-g output_directory]

```

Where:

- **-c <remote-command>:** Launches Formality
- **-ct <module-name>:** Verifies the module and its submodules as a cluster
- **-ctf <file-name>:** Specifies the user cluster file

The file format is as follows:

```

{ m1_for_cluster1 m2_for_cluster1 ... top_for_cluster1}
{ m1_for_cluster2 m2_for_cluster2 ... top_for_cluster2}
{ ... ... }

```

- **-ct_intf:** Clusters the module with SV interface
- **-cycle <integer>:** Specifies the number of cycles for simulation; the integer should be at least 1
- **-d <synthesis-directory>:** Path to the synthesis directory (for example, zcui.work/design/synth)
- **-f <formality-script>:** Loads the Formality commands
- **-freq:** Specifies the user clock frequency in KHz for random simulation

- **-g <directory>**: Specifies a directory to generate files; the directory is relative to the **-d** directory; Default is **formality**
- **-h**: Prints the help message
- **-initcycle <integer>**: Specifies the number of initial cycles for simulation; the integer should be at least 1
- **-initial_reg**: Enables initial reg in simulation
- **-xprop <1/0>**: Enables or disables **-xprop** in simulation
- **-gl <1/0>**: Enables or disables the gate level simulation for memory modules
- **-j <jobs>**: Specifies the number of Formality jobs to launch in parallel
- **-k**: Skips modules if they are EQUIVALENT by formality check (looking in the log file)
- **-l <hh:mm:ss>**: Sets timeout limit for each job
- **-m <module-names>**: Verifies the specified modules
- **-mf <file-name>**: Verifies the module list file; only the modules listed in this file are verified
- **-mh <module-name>**: Verifies the module and its submodules
- **-n**: Specifies the number of modules to be checked for each bundle (for example, **-n 100:200**), starts from 1
- **-no_fm_lic**: Runs formality verification without formality license
- **-nocluster**: Do not cluster the modules with complex type ports for simulation
- **-norm**: Do not remove the random simulation intermediate directories
- **-o <directory>**: Specifies a directory to generate files; the directory is relative to the current directory
- **-opt_vhdlan <vhdlan_opts>**: Passes options to **vhdlan** invoked in simulation
- **-opt_vlogan <vlogan_opts>**: Passes options to **vlogan** invoked in simulation
- **-opt_vcs <vcs_opts>**: Passes options to VCS invoked in simulation
- **-opt_elab <vcs_opts>**: Passes options to VCS elaborate in simulation
- **-opt_simv <vcs_opts>**: Passes options to simulation runtime

- -p: Prints only result (parse current log files)
- -r: Runs Formality jobs (with previously created jobs)
- -rerun: Reruns formality check on simulation RUNTIME-ERROR modules without time limit
- -rt <integer>: Specifies the number of times to rerun simulation on RUNTIME-ERROR modules
- -s <suppress-messages>: Suppresses messages in Formality
- save <d|u|du>: Saves sessions for DIFFERENT or/and UNKNOWN
- -scm: Runs formality on modules at (SIMON) level
- -u <tcl-cmd-directory>: Specifies the directory where user supplied Tcl files reside
- -v: Runs vcs simulation on modules
- -xtor: Verifies only the modules under xtor directories
- +lic+wait: Specifies or waits for license if none is available

Extraction is the first step of verification. Synthesis files are read from the synthesis directory (provided with -d) and written into the formality directory. By default, the formality directory is 'formality' inside the synthesis directory. If -g is used, a different name can be provided.

If the directory provided with -g is absolute, then this absolute path is used. All files in the destination directory (formality) are deleted.

If -d is not provided, the formality directories provided with -g are relative to '.' (default is ./formality).

Absolute path can still be provided in -g. All job scripts and saved sessions in the destination directory (formality) are deleted.

When running the Formality/Simulation jobs (with -r), a report is automatically printed (summary only). All saved sessions in the destination directory (formality) are deleted. The option -c can be used with -r, -v or -scm.

Printing report (-p) can be run at any time (also while running Formality).

Multiple -mf <file-name> can be given, and the module names in the file are separated. Lines starting with # are treated as comments

The following log files are saved in the directory provided by -g. If -g is not used, the default directory is formality present in the synthesis-directory, which is provided by -d:

- equivalent.log
- different.log
- unknown.log
- blackbox.log
- unverified.log
- sim_pass.log
- sim_fail.log
- sim_error.log

There are three main flow -r, -v, and -scm. The priority is -r > -v > -scm. The lower priority flow only works on the failed modules, which are verified by the higher priority flow.

If only one flow is selected, all the modules are run through the selected flow.