ZeBu® User Guide

Version V-2024.03-1, July 2024

SYNOPSYS®

Copyright and Proprietary Information Notice

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 https://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 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

Feedback

Contents

About This Book 12
Contents of This Book 12
Related Documentation 13
Synopsys Statement on Inclusivity and Diversity 14

1. Emulator System 15
Overview Of Emulators 15
ZeBu Server 5 16
ZeBu EP1 17
ZeBu Server 4 18
ZeBu Emulation Flow 19
Introduction to zCui 19

2. Assembling a Design for Emulation 21
Mapping Your Design and Test Environment 21
Importing a Design From a Simulation Environment 21
Preparing a DUT-Only Environment 22
Clock Modeling 22
Reset Modeling 23
Memory Modeling 23
Inference From a Design 24
Guidelines for Performance 24
Memory Model IP 26
Designating Signals for Access During Runtime 27
Reading Signals 27
Forcing and Injecting Values 27
Verilog System Tasks 28
$display System Task 32
`$memset()` System Task 33
`$plusargs` System Task 33
Example 34

Limitations of `$plusargs` . . . 34
`$fopen` and `$fclose` System Tasks . . . 35
Example . . . 35
Limitations of `$fopen` and `$fclose` tasks . . . 35
`$finish` System Task . . . 36
Enabling $finish Task in zRci . . . 36
zRci Example . . . 36
$sformat and $sformatf System Tasks . . . 37
Verilog Procedural Interface . . . 37

3. Compilation . . . 38
Overall Unified Compile Flow . . . 39
Introduction to UTF . . . 39
Basic UTF Commands . . . 40
Synthesis . . . 41
Synthesis Log Files . . . 41
Compilation Script for VCS (UUM) . . . 42
Setting Up VCS . . . 42
Setting Up the `synopsys_sim.setup` File . . . 42
Setting Up Compilation Commands Using UTF . . . 43
Compilation Settings for DesignWare Blocks . . . 44
Running a Design in Lint Mode . . . 44
Validate ZeBu Synthesis Correctness . . . 45
Specifying the Hardware Architecture for Compilation . . . 47
Flow Selection During Compilation . . . 47
Grid/LSF Handy Commands . . . 47
Gate-Level Netlist Compilation . . . 49
Handling Design Clocks Using Fetch Mode . . . 50
Limitations . . . 50
Removing Strongly Connected Components (SCC) . . . 50
Use Model - Compilation . . . 51
Use Model - Runtime Debug . . . 52
Debug Procedure . . . 52
Runtime Relaxation for Full Loop Breaking . . . 53
Limitations of the Removing SCC Feature . . . 53
Achieving Compilation Objectives . . . 54

Enable Compilation Objective 54
Compiling a Project Using zCui 54
Compiling With zCui in the Batch Mode 55
Compiling With zCui in the GUI Mode 55
Creating Runtime Archives After Compilation 58
GUI Mode 58
Batch Mode 59
Usage 59
Verdi Database Tasks in zCui 60
Important ZeBu Log Files 61
Analyzing Compilation Results Using zBatchExplorer 61
Viewing Timing Information in the zTime Reports 62
Accessing the zTime Reports 63
Using zTime.html for Analysis 63
Viewing Timing Summary 65
Native Compilation Profiler 66
Prerequisites 66
Multi-Process Flow 69
Grid Queue Configuration 70
Reporting XMR Information in a Design 71
Example 72
Viewing Consolidated Information Using zAudit 73
Prerequisites 74
Launching the zAudit Tool 74
zAudit Command Line 74
zAudit Examples 75
Visualizing Compile-Time Data 87
Enabling zCTgram 88
Handling Misfiring SVAs with Latches and Memories in Fanin 90
Use Model 91
Replicating Multiport Memories in zMem 91
Usage 91
Reporting High Fanout Nets 92
Limitations 93
Verilog force/release 94
Reporting Verilog force/release Information 94
Example 94
Limitations 95
Enhanced Multi-Gigabit Transceiver Support 96

Multi-Bin Based FPGA Complexity 96
Enabling Multi-Binning 97
Performance Driven Multi-die Multiplexing Support 97
Enabling PDM Support 98
zCui Flow 98
Memory Optimization 99
Reshaping of User Memories in Backend 99
Reporting 99
Limitations 100
Decomposing Large ZRM Memories to Fit the Hardware 100
Reporting 101
Limitations 101
Memory Coalescing 101
Reporting 103

4. Clock Modeling in ZeBu 104
Types of Clock Supporting Emulation 105
Cycle-Based Clock Control 105
Support of Delays With zceiClockPort 106
Time-Based Clock Control 106
ClockDelayPort 107
Using the clockDelayPort Macro 107
Optimizing clockDelayPorts 109
Configuring clockDelayPort Through designFeatures 110
Using ZeBu Clock Delay Primitive 110
Tolerance Support 111
Limitations With clockDelayPort 113
RTL Delays 114
Enabling Clock Delay Feature 114
Supported Verilog Language Subset 115
Partially Supported Constructs 116
Unsupported Constructs 116
Waveform Dumping for the Clock Delay Feature 117
Reducing Disk Usage 117
Timescale and Precision 117
Compilation Results 118
Timestamp Clock 118
Global Time 118

Limitations of the Clock Delay Feature 119
C_COSIM and Vertical Solution Transactors 119
zDPI and ZEMI-3 120
Example: zDPI or ZEMI-3 import function 120

5. Runtime 121
Controlling Runtime Parameters 121
Emulation Speed 122
Clock Definition 122
Clock Definition . . . . .
Memory Initialization 123
zRci for Controlling Clocks and Managing Memory 123
Controlling Clocks 123
Obtaining a List of Clock Groups 124
Enabling Clocks for N Cycles 124
Disabling Clocks 124
Getting the Number Clock Cycles Executed 125
Managing Memory 125
Saving Memory Contents 125
Loading Memory Contents 126
Cycle-Based C/C++ Co-Simulation 126
Reading Signal Values During Runtime 127
C++ Example 128
zRci Example 128
Changing the Signal Values During Runtime 128
Depositing a Value on a Signal 129
C++ Example 129
`zRci` Example 129
Injecting a Value on a Signal 129
C++ Example 129
`zRci` Example 129
Forcing a Value on a Signal 130
C++ Example 130
`zRci` Example 130
Releasing a Forced Signal 130
C++ Example 130
`zRci` Example 130

6. RunManager 131
Enabling RunManager 131
RunManager APIs 132

7. Using SystemVerilog Assertions 134
SVA Compilation 134
Runtime Usage 135
Controlling Assertions From the C++ Testbench 135
Starting Assertion Processing 136
Stopping SVA Processing 137
Resetting SVA Fail Counters 138
Dealing with Assertion Failures 138
Changing the Reported Severity Level 138
Disabling/Enabling Assertions and Signaling SVA Failures 139
Controlling Assertions Using zRci 140
Starting Assertion Processing 140
Stopping SVA Processing 140
Getting SVA Failure Counters 140
Resetting SVA Fail Counters 141
Runtime Options for Assertion Control Tasks 141
zsva_trigger 142
Post Processing SVA 143
Supported SVA Subset 144

8. ZeBu DPI Technologies 148
Supported DPI Function Calls 149
Unified DPI 149
Unified DPI Versus zDPI 150
Compilation 151
zDPI (Default Technology) 151
Unified DPI 153
Analysis of Unified DPI Log Files 154
Compiling ZeBu DPI files 157
Writing DPI Import Functions 157
C/C++ Language Compatibility 157
Header Files 157

ZeBu API to Control DPI Functions at Runtime 158
C++ API to Control DPI Functions 158
Clocking Mode 159
Enabling Synchronization of DPI Calls 160
Selecting Function Calls Only on Input Changes 160
Loading Dynamic Libraries 161
Starting DPI Call Processing 161
Stopping DPI Call Processing 162
zRci Tcl Commands to Control the DPI Functions 164
Runtime for Unified DPI 164
Processing DPI Function Calls Offline 165
Identifying the DPI Functions 166
Enabling the DPI Offline Feature During Emulation 166
Using `ZEBU_OFFLINE_DPI` and its Offline DPI Specification File 166
Using the DPI Offline Feature With Methods Within the Testbench 168
Creating a Shared Library 171
Creating an Input File for the `zdpiReport` Tool 171
Using the `zdpiReport` Tool 171
Speed-up Decoding 173
Diagnostics 174
Profiling 174
Optimization Guidelines 174
Enhancing Data Transfer 175
Defining Sampling Clock Frequency 176
Investigating Emulation Slowdowns 176
Checking with zDPI Disabled 177
Checking With zDPI Enabled 177
Limitations of the ZeBu DPI Technologies 178
zDPI 178
Multiple Clock Groups 178
Performance 178
Synthesis 178
Unified DPI 178
Example of DPI Function Calls to Print Counter Value 179
Design 179
System Verilog Code 179
Synthesis 180
Implementation of DPI Functions 181

Controlling DPI Calls from a C++ Testbench 181
Implementation of the Testbench 181
Proceeding With Runtime 183
Controlling the Calls from zRci 183

9. Saving the Design State and Restarting from a Saved State 184
Save and Restore - Recommendations 184
C++ Method for Save and Restore 185
C++ Example 187
zRci Method for Save and Restore 187
zRci Example 188

10. Clock Cone Visualization in zBrowser 189
Enabling Clock Cone Visualization 191
Finding Clocks with Maximum Depth 191
Finding Clocks with Maximum TFI 192
Tracing a Known Clock Signal 193
Example 194
Tracing the Clock Path Contributing to Maximum Logic Levels 196
Tracing the Clock Path Contributing to Maximum IO cut 198
Tracing an Intermediate Clock Signal to its Final Output 198
Analyzing Clock Localization 199
Displaying CORE Level Mapping of Clock Instances 199
Use Model 200

11. Appendix 201
designFeatures File 201
Syntax Rules 202
Location of the Calibration Files 203
Declaring the Process Name 203
Single-process Environment 203
Multi-process Environment 203
Unnamed Control Processes 204
Initialization Phases Not Executed on Memory 204
Parameters for Design Clocks 205

Parameters for Primary Clocks 205
Parameters for Transactors 211
Global Transactor Settings 211
Specific Transactor Settings 211
Initializing Memories 212
List of Memory-Content Files 212
Programming the driverClk Reset Signal 213
Programming the DUT Reset 213
Declaration for Smart Z-ICE 213
Symbolic Parameters for Timing Settings 214
Reducing Disk Usage With Clock Delay Feature 215
Runtime Clock file 216
Clock File Content 216
Clock File Syntax 216
Memory Content File 217
ZeBu-Proprietary Binary Format 218
Memory Binary Format for Shorter Load Time 218
Memory Binary Format for Shorter Load Time . . . . .
Memory Content File Text Format 218
Example 219

Preface

This chapter has the following sections:

- About This Book
- Contents of This Book
- Related Documentation
- Synopsys Statement on Inclusivity and Diversity

About This Book

The *ZeBu® 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® User Guide* has the following chapters:

Chapter	Describes...
Emulator System	Introduces ZeBu Server and its features.
Assembling a Design for Emulation	Describes how to prepare your design for emulation before compilation.
CompilationCompilation	Describes how to compile your design in ZeBu.
Clock Modeling in 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.

Chapter	Describes...
ZeBu DPI TechnologiesZeBu DPI Technologies	Describes DPI imported function calls supported in ZeBu.
Saving the Design State and Restarting from a Saved State	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.
Clock Cone Visualization in zBrowser	Describes the Clock panel in zBrowser.
Appendix	Describes the input files for compilation and runtime operations specific to the ZeBu environment.

Related Documentation

Document Name	Description
ZeBu User Guide	Provides detailed information on using ZeBu.
ZeBu Debug Guide	Provides information on tools you can use for debugging.
ZeBu Debug Methodology Guide	Provides debug methodologies that you can use for debugging.
ZeBu Unified Command-Line User Guide	Provides the usage of Unified Command-Line Interface (UCLI) for debugging your design.
ZeBu UTF Reference Guide	Describes Unified Tcl Format (UTF) commands used with ZeBu.
ZeBu Power Aware Verification User Guide	Describes how to use Power Aware verification in ZeBu environment, from the source files to runtime.
ZeBu Functional Coverage User Guide	Describes collecting functional coverage in emulation.
Simulation Acceleration User Guide	Provides information on how to use Simulation Acceleration to enable cosimulating SystemVerilog testbenches with the DUT
ZeBu Verdi Integration Guide	Provides Verdi features that you can use with ZeBu. This document is available in the Verdi documentation set.
ZeBu Runtime Performance Analysis With zTune User Guide	Provides information about runtime emulation performance analysis with zTune.
ZeBu Custom DPI Based Transactors User Guide	Describes ZEMI-3 that enables writing transactors for functional testing of a design.

Document Name	Description
ZeBu LCA Features Guide	Provides a list of Limited Customer Availability (LCA) features available with ZeBu.
ZeBu Synthesis Verification User Guide	Provides a description of zFmCheck.
ZeBu Transactors Compilation Application Note	Provides detailed steps to instantiate and compile a ZeBu transactor.
ZeBu zManualPartitioner Application Note	Describes the zManualPartitioner feature for ZeBu. It is a graphical interface to manually partition a design.
ZeBu Hybrid Emulation Application Note	Provides an overview of the hybrid emulation solution and its components.

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

1

Emulator System

This section presents an overview and the features of the Synopsys emulator, ZeBu.

This section consists of the following topics:

- Overview Of Emulators
- ZeBu Emulation Flow
- Introduction to zCui

Overview Of Emulators

ZeBu Emulator is a very high-capacity emulator system with easy setup and debugging. ZeBu emulator 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. The following emulators are available with ZeBu.

- ZeBu Server 5
- ZeBu EP1
- ZeBu Server 4

To choose the appropriate emulator that meets your requirements, contact your Synopsys Personnel.

Feedback

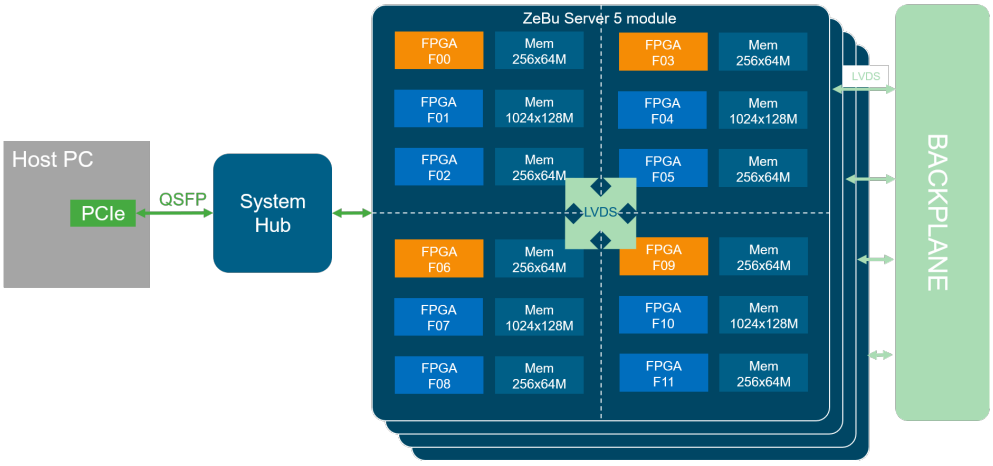
ZeBu Server 5

The following figure shows ZeBu Server 5:

Figure 1 ZeBu Server 5 System

The following figure displays the architecture of ZeBu Server 5.

Figure 2 ZeBu Server 5 Architecture



For details on design capacity and FPGA modules supported on ZeBu Server 5, see *ZeBu Server 5 Site Planning Guide*.

For detailed information about ZeBu Server 5 and software prerequisites for installing ZeBu, see *ZeBu Server 5 Site Administration Guide*.

Feedback

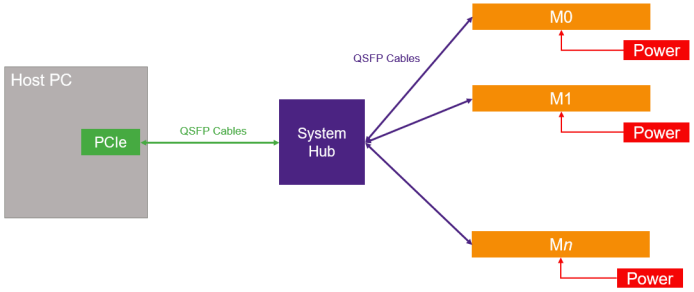
ZeBu EP1

The following figure shows ZeBu EP1:

Figure 3 ZeBu EP1 System

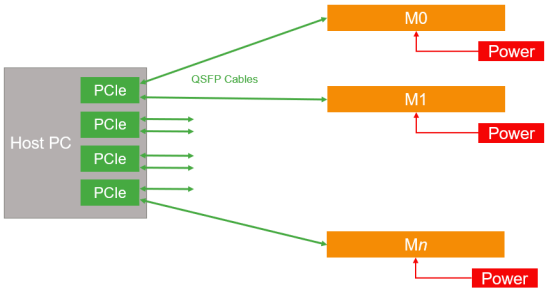
The following figure displays the architecture of ZeBu EP1 with up to 16 modules:

Figure 4 ZeBu EP1 Architecture With Up to 16 Modules



The following figure displays the architecture of ZeBu EP1 with up to 8 modules:

Figure 5 ZeBu EP1 Architecture With Up to 8 Modules



For details on design capacity and FPGA modules supported on EP1, see *ZeBu EP1 Site Planning Guide*.

Feedback

For detailed information about ZeBu EP1 and software prerequisites for installing ZeBu, see *ZeBu EP1 Site Administration Guide*.

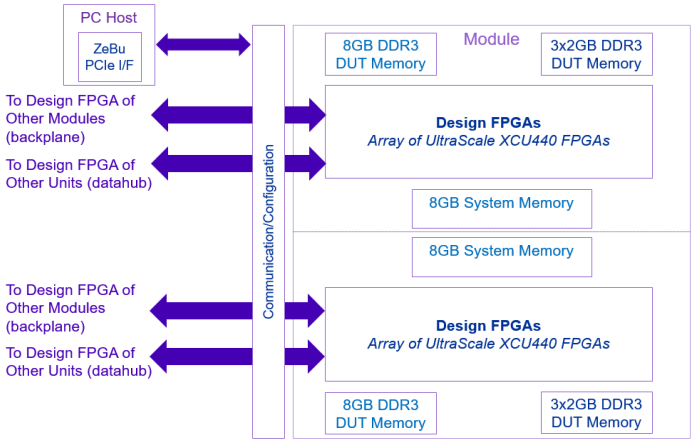
ZeBu Server 4

The following figure shows ZeBu Server 4:

Figure 6 ZeBu Server 4 System

The following figure displays the architecture of ZeBu Server 4.

Figure 7 ZeBu Server 4 Architecture



For details on design capacity and FPGA modules supported on ZeBu Server 4, see *ZeBu Server 4 Site Planning Guide*.

Feedback

For detailed information about ZeBu Server 4 and software prerequisites for installing ZeBu, see *ZeBu Site Administration Guide*.

ZeBu Emulation Flow

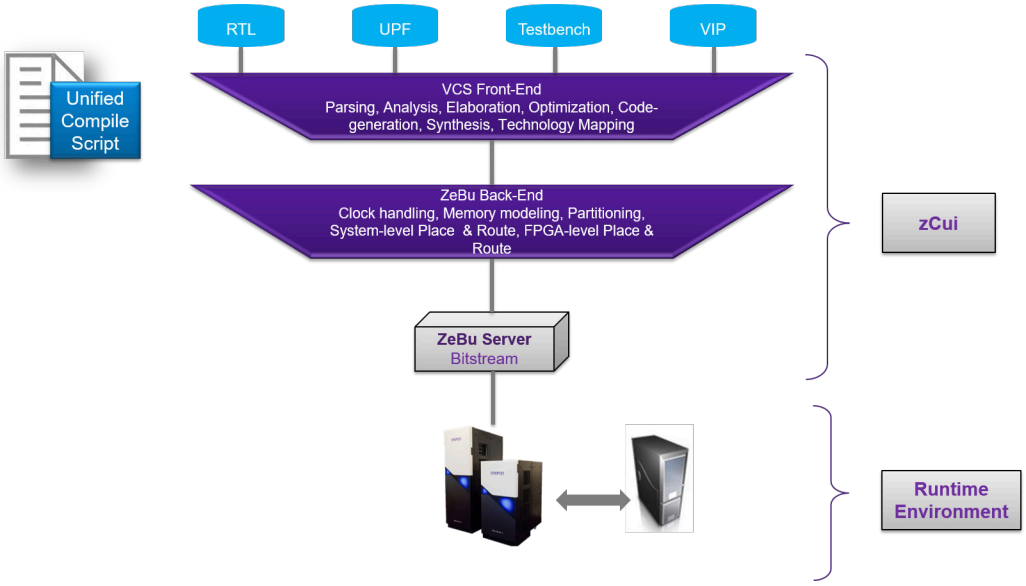
In ZeBu emulation, design files and project files are compiled using VCS.

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 8 ZeBu Emulation Flow



The ZeBu emulation flow consists of the following steps:

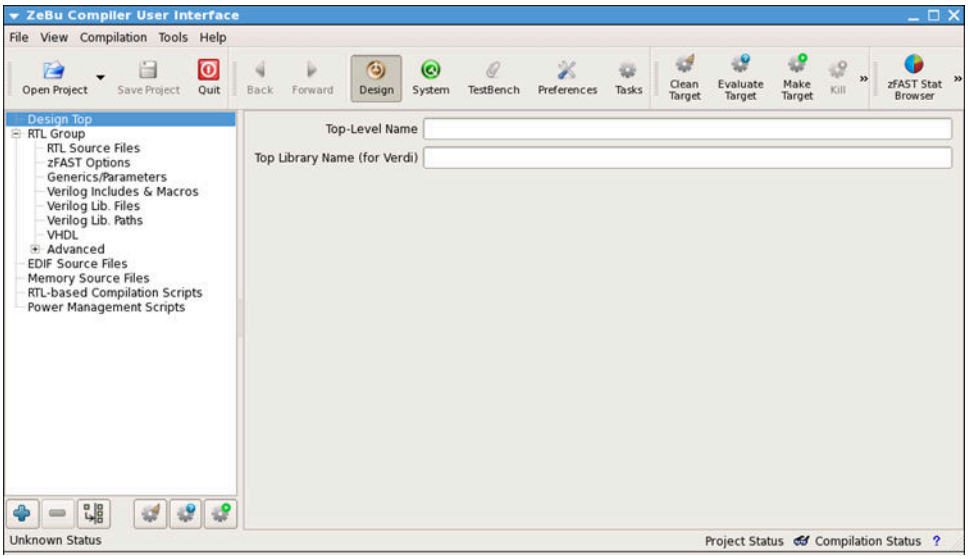
- Assembling a Design for Emulation
- Compilation
- Runtime

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**.

Feedback

Figure 9 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 Using zCui.

2

Assembling a Design for Emulation

This section describes how to assemble your design for emulation before compilation.

It consists of 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

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.

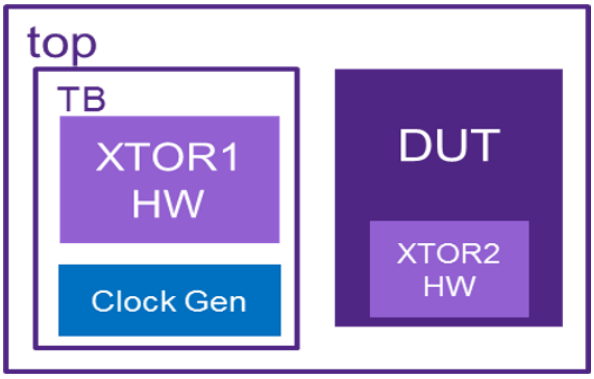
Importing a Design From a Simulation Environment

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.

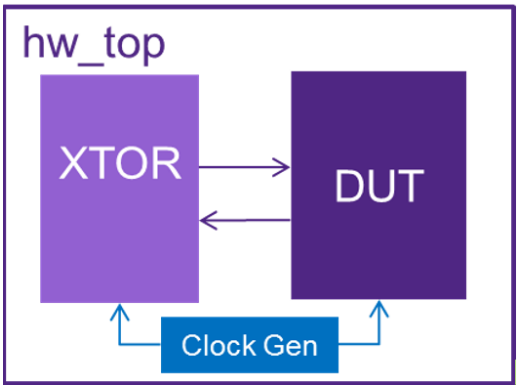
Figure 10 Importing a DUT and Test Environment From a Simulation Environment



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 11 Preparing a DUT-Only Environment



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.

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 **zRci** to control your testbench, **zRci** 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`.

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 topics:

- Inference From a Design
- Memory Model IP

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.

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
```

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).
- 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 UTF command to specify the list of transactors and their respective paths.

Designating Signals for Access During Runtime

This section describes how signals can be accessed during runtime.

It consists of the following topics:

- Reading Signals
- Forcing and Injecting Values

Reading Signals

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

The following UTF 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.

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 UTF 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 UTF 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.

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 *ZeBu Custom DPI Based Transactors User Guide*.

The synthesis of the Verilog system-task is not always enabled by default in DUT. Use the following UTF commands to enable Verilog system-task:

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

In ZEMI3 context, add only `system_tasks` command in UTF.

Some of the Verilog system-tasks are not supported by default for the ZeBu flow, and may require enabling additional features.

The following table lists the Verilog system-tasks support status in ZeBu and specifies the features to enable wherever required.

Table 1 Supported Verilog System-Tasks

Task	Support Status	Default	Required Feature
Simulation Tasks			
`$finish (always block)`	Yes	No	-
`$finish (initial block)`	Yes*	No	*Unified DPI in DUT context
Simulation Time Functions			
`$realtime`	Yes**	No	**RNM
`$time`	Yes***	No	***Clock delay
Timescale Tasks			
`$timeformat`	No	-	-
Conversion Functions			
`$bitstoreal,$realtobits`	Yes**	No	**RNM
`$signed,$unsigned`	Yes	Yes	-
`$cast (statically determined)`	Partial	Yes	-
Array Query Function			

Feedback

Table 1 Supported Verilog System-Tasks (Continued)

Task	Support Status	Default	Required Feature
`$unpacked_dimensions, $dimensions,$left, $right,$low, $high, $size, $increment`	Yes	Yes	-
Math Functions			
`$clog2,$pow,$sqrt,$floor,$ceil,$sin,$cos`	Yes**	Yes	**RNM
`$asin, $ln, $acos, $log10, $atan, $atan2, $exp,$hypot, $sinh, $cosh, $asinh, $acosh, $tan, $atanh`	No	No	-
Bit Vector System Functions			
`$countbits`	No	-	-
`$countones,$onehot, $onehot0`	Yes	Yes	-
`$isunkown`	Partial	-	In SVA
SeverityTasks			
`$fatal$error,$warning, $info`	Yes	No	-
Elaboration Tasks			
`$fatal,$error, $warning, $info`	Yes	Yes	-
Assertion Control Tasks			
`$asserton,$assertoff, $assertkill`	Yes	Yes	-
Sampled Value System Functions			
`$rose,$fell, $stable, $changed, $past`	Yes	Yes	-
Coverage Control Functions			
`$coverage_control,$coverage_get_ max, $coverage_get,$coverage_merge, $coverage_save,$get_coverage, $set_coverage_db_name, $load_coverage_db`	No	-	-

Feedback

Table 1 Supported Verilog System-Tasks (Continued)

Task	Support Status	Default	Required Feature
Probabilistic Distribution Functions			
`$random($memset)`	Partial	-	-
`$dist_chi_square,$dist_erlang, $dist_t,$dist_exponential, $dist_normal,$dist_poisson, $dist_uniform`	No	-	
Stochastic Analysis Tasks and Functions			
`$q_initialize,$q_add`	No	-	-
`$q_full,$q_exam, $q_remove`			
PLA Modeling Tasks			
`$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$plane`	No	-	-
Miscellaneous Tasks and Functions			
`$system`	No	-	-
Display Tasks			
`$display, $displayb, $displayh, $displayo, $write, $writeb, $writeh, $writeo`	Yes	No	-
`$strobe,$monitorh`	No	-	-
`$monitor(initial block)`	Yes	Yes	-
FileI/O Tasks and Functions			

Feedback

Table 1 Supported Verilog System-Tasks (Continued)

Task	Support Status	Default	Required Feature
`$fopen, $fclose, fdisplay, $fdisplayb, $fdisplayh, $fdisplayo, $fwrite, $fwriteb, $fwriteh, $fwriteo`	Yes	No	-
`$swrite, $swriteb, $swriteh, $swriteo`	Yes	No	-
`$sformat,$sformatf`	Yes*	No	*Unified DPI in DUT context
`$fflush`	Yes	No	*Unified DPI in DUT context
`$fstrobe,$fstrobeb,$fstrobeh, $fstrobeo,$fmonitor, $fmonitorb,$fmonitorh, $fmonitoro,$fscanf, $fread, $fseek,$feof, $fgetc, $ungetc, $fgets, $sscanf, $rewind, $ftell, $ferror`	No	No	-
Memory Load Tasks			
`$readmemb,$readmemh`	Yes	Yes	-
Memory Dump Tasks			
`$writememb,$writememh`	No	-	-
`$memset`	Yes	Yes	-
Command Line Input			
`$value$plusargs(always block), $test$plusargs(alwaysblock)`	Yes	Yes	-
`$value$plusargs(initial block), $test$plusargs (initialblock)`	Yes*	Yes	*Unified DPI in DUT context
VCD Tasks			
`$dumpvars,$dumpports` See Note below.	Partial	-	-

Feedback

Table 1 Supported Verilog System-Tasks (Continued)

Task	Support Status	Default	Required Feature
`$dumpfile,$dumpoff, $dumpon, $dumpall,$dumplimit, $dumpflash, $dumpportsoff, $dumpportson,$dumpportsall, $dumpportslimit,$dumpportsflush`	No	-	-

Note:

When specifying `$dumpvars`, add the design object name within double quotes. For example:

```
(* fwc *) $dumpvars(1, "dut_tb.inst_ifx_pc_monitor0_if.pc_o");
```

If the design object name is specified within double quotes, then a specialized algorithm is used to match the design objects in `$dumpvars`. This improves the matching. In addition, the performance and peak memory of `$dumpvars` processing step is improved.

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);
```

$display System Task

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 ZeBu DPI Technologies.

$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.

$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 (file name can be a full path to the file or a relative path).

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.

Limitations of $plusargs

- 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.

$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
```

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
```

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

`int fd;`
`fd = $fopen("my.log", "w")`
	`wire[8*80-1:0] file_name;`
	`assign file_name = "my.log";`
	`fd = $fopen(file_name, "r") ;`

$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 with **zRci**.
- `$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.

Enabling $finish Task in zRci

To enable the `$finish` task, specify following before calling `start_zebu`:

```
dollar_finish -enable <proc_name>|-disable
```

Where,

- `[-enable <proc_name>]`: Tcl callback procedure name.

  Note:

  The procedure must be defined to not receive any arguments.

- `[-disable]`: Disables `$finish` callbacks on this session.

zRci Example

```
proc do_finish {} {
  global zfinished
  puts "#### Finish called ####"
  set zfinished 1
  ccall -flush
  quit
}
###  dollar_finish for $finish, must put before start_zebu
dollar_finish -enable do_finish
```

Feedback

```
start_zebu rundir
...
```

$sformat and $sformatf System Tasks

To enable the support of Verilog system tasks `$sformat` and `$sformatf` inside DUT in the context of Unified DPI, add `-zebu_string` to the VCS command in addition to the UTF settings.

Similarly, to enable the support of Verilog system tasks `$sformat` and `$sformatf` in ZEMI-3, enable ZEMI-3 at runtime.

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 section describes how to compile your design in ZeBu.

It discusses the following topics:

- Overall Unified Compile Flow
- Synthesis
- Compilation Script for VCS (UUM)
- Running a Design in Lint Mode
- Validate ZeBu Synthesis Correctness
- Specifying the Hardware Architecture for Compilation
- Flow Selection During Compilation
- Grid/LSF Handy Commands
- Gate-Level Netlist Compilation
- Handling Design Clocks Using Fetch Mode
- Removing Strongly Connected Components (SCC)
- Achieving Compilation Objectives
- Compiling a Project Using zCui
- Important ZeBu Log Files
- Verilog force/release
- Enhanced Multi-Gigabit Transceiver Support
- Multi-Bin Based FPGA Complexity
- Performance Driven Multi-die Multiplexing Support
- Memory Optimization

Feedback

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) .

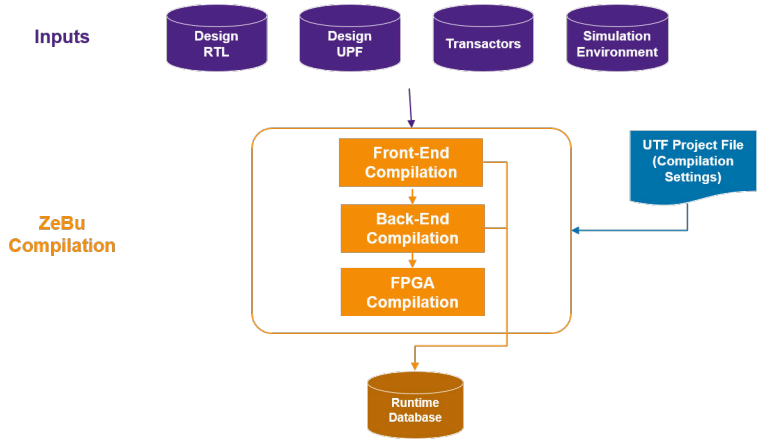
The UTF project file (for example, `project.utf`) is the main input file to the ZeBu compiler. It contains all the information required for successful compilation, including the VCS compilation command and the ZeBu back-end commands.

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 ZeBu back-end compilation, and Xilinx FPGA Place and Route.

The following figure illustrates the ZeBu compilation flow.

Figure 12 ZeBu Unified Compile Flow



This section consists of the following sub-sections:

- Introduction to UTF
- Basic UTF Commands

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.

Feedback

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 2 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.

Basic UTF Commands

The following table lists some of the basic UTF commands.

Table 3 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.
`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.

Feedback

Synthesis

ZeBu synthesis tool is implemented by VCS™ compiler. It supports the following features:

- Formal model construction
  - Fully-synthesized hierarchical design view of the HDL
  - Infers Memories, Registers, and Latches
- 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
- Provides support for VCS congruency and full design optimization at word-level data model.

Synthesis is followed by Bundle synthesis for technology mapping step.

Synthesis Log Files

The following table lists the various logs created by synthesis.

Table 4 Synthesis Logs

Synthesis Stages	Logs
VCS simulation & synthesis	See "`vcs_splitter_VCS_Task_Builder.log`" File can be found in: `zcui.work/zCui/log/vcs_splitter_VCS_Task_Builder.log`
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

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

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
```

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`)

Setting Up Compilation Commands Using UTF

A compilation script (or command line) must be provided to the UTF 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).

In addition, review the RTL code to check if there are any VCS predefined macros. If the content between `` `ifdef VCS … `endif `` is not supported in emulation, then add `-undef_vcs_macro` option for the `vlogan` commands.

In addition, you need to review the RTL code between the `translate_on/translate_off` pragmas. It is supported with simulation but not supported with emulation. In this scenario, while synthesizing for emulation, disable the code using the `-skip_translate_body` option for `vlogan`/`vhdlan` commands.

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 SolvNetPlus by performing the following steps:

1. Log in to the SolvNetPlus online support site using your SolvNetPlus account (*https://solvnetplus.synopsys.com/*).
2. Click the **Documentation** tab and select **VCS®** or **VCS®MX**.

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:

Create the work directories for each library, as shown in the following example:

```
mkdir dware
```

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

```
vhdlan -full64 -work dware
 $SYNOPSYS/dw/fpga_ip/fv/dw_foundation/dware_comp.vhd.e
vhdlan -full64 -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.

Running a Design in Lint Mode

To detect issues in the RTL code, run your design in lint mode by using the following **zCui** command in batch mode:

```
zCui -u <project-name>.utf --lint | --lint elab | --lint synth
```

The following lint modes are available:

- **Elaboration lint mode**: In this mode, only VCS elaboration is run. To enable this mode, specify the following command:

  ```
  zCui --lint elab
  ```

  - If there are no elaboration errors, VCS exits with normal termination.
  - If there are elaboration errors, VCS exits with abnormal termination. Check the VCS log file.

- **Synthesis lint mode**: In this mode, elaboration and the lint version of synthesizer are run and no synthesis files are generated. To enable this mode, specify either of the following commands:

  ```
  zCui -- lint
  ```

  or

  ```
  zCui --lint synthesis
  ```

- If there are no synthesis check issues, VCS exits with normal termination.
- If there are synthesis check errors, VCS exits with abnormal termination. Check the VCS log file, where NOTE and ERROR messages are stated and a summary is provided at the end of the file. In the beginning of the log file, the RTL source files and line numbers are provided for the non-synthesizable constructs. The summary lists the non-synthesizable modules. In some cases, there are some non-synthesizable constructs that are not stated in the summary. However, they are all stated as errors in sections before the summary section of the report.

For more details about running **zCui** in batch mode, see the Compiling With zCui in the Batch Mode section.

Validate ZeBu Synthesis Correctness

zFmCheck is module-based verification tool to validate that the ZeBu synthesis engine has correctly synthesized Verilog modules, VHDL entity, or architecture pairs. **zFmCheck** verification is based on the comparison between the reference and implementation models using Formality Equivalence Checking and VCS simulation technologies.

In the **zCui** compile directory, **zFmCheck** uses either the synthesizer outputs or the final synthesis netlist output as the implementation model and the decompiled Verilog or VHDL as the reference model.

To enable **zFmCheck**,add the following UTF option before compiling a design in ZeBu:

```
synthesis -generate_db_for_fmcheck true
```

After the design is successfully synthesized, **zFmCheck** can be invoked from the compile directory using the following command:

```
zFmCheck -d "zcui.work directory" -r -v -j NUMBER -c "remote_cmd" -o
 fm.out
```

The log is available at `fm.out/formality.log`.

The following is a sample summary log:

```
###################Formality summary#####################
EQUIVALENT      : 574 (98%)
DIFFERENT       : 3 (1%)
UNKNOWN                                 : 9 (2%)
  |--REF SETUP FAIL        : 3
  |--MULTIPLY DRIVEN       : 1
  |--VERIFY INCONCLUSIVE   : 1
  |--OTHERS     : 4
 BLACKBOX
  UNVERIFIED              : 0 (0%)
  SKIPPED                 : 0 (0%)

            588
#########################################################
#################Simulation summary######################
Modules  : 12
PASS                : 8
FAIL                : 1
ERROR/MISSING       : 3
RUNTIME-ERROR       : 0
#########################################################
```

Basic **zFmCheck** options used in the command above are explained in the following table:

Table 5 zFmCheck Command Basic Options

Command Options	Description
`-r`	Run formality equivalence check
`-v`	Run simulation check
`-o output_directory`	Use a directory to generate files, the directory is relative to the current directory
`-c remote_command`	Remote command to launch Formality
`-d zcui.work directory`	Path to `zcui.work` directory

For more details about **zFmCheck**, see the *ZeBu® Synthesis Verification User Guide*.

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,

```
architecture_file -filename
 {/path/to/ZEBU_SYSTEM/config/zse_configuration.tcl}
```

Flow Selection During Compilation

To provide stable and performant compilation, settings can be enabled during compilation using the following command:

```
set_perf_flow <flow>
```

Where, `<flow>` can be either `base`, `flow1`, or `flow2`.

Note:

If you are using HAPS, you must set the target as follows:

```
set_perf_flow <flow> HAPS
```

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 6 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*

Feedback

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`
- `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
```

Gate-Level Netlist Compilation

When compiling Gate-Level netlist, it implies many instances in the design, one per gate, which has an impact on the capacity.

On the other side, it might lead to long hierarchical level that has an impact on design performance.

The inlining feature is now enabled by default with a limit set to 50. This means, that all modules with cost, the number of its instances and the number of gates/continuous assign/UDPs/regs, which is less or equal to 50 are inlined. This helps to minimize the design capacity and improve the design performance

You can still adjust auto-inline limit by using the following UTF command:

```
Optimization -auto_inline_limit <limit>
```

Where, `<limit>` is an integer

Note that modules can be automatically detected for inlining regardless auto-inline limit if:

- Modules are declared between `celldefine` and `endcelldefine`
- Modules are loaded from a library with `-y` or `-v` options for VCS

The other options are available as part of the `Optimization` UTF command to handle inlining settings 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`
- `-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`.

Handling Design Clocks Using Fetch Mode

Fetch mode replaces the existing clock handling mode (that is, Filter Glitches Synchronous) and computes clock and data in parallel. At any cycle N, clocks for cycle N +1 are precomputed (in parallel) while emulation is computing data paths. Therefore, the driver clock frequency is calculated as:

```
1/driver_clock_period = max(longest clock path, longest data path)
```

The fetch mode is the default clock-processing mode on ZeBu Server 4 and later platforms.

Limitations

The following limitations exist:

- Fetch mode does not support `zrm_transactional_mode`. Use `zrm_performance_mode` instead.
- `zTopBuild` reports an error if above features are enabled during compilation.

Removing Strongly Connected Components (SCC)

Combinational loops are an inherent part of a design. For example, they can be present as a design choice or due to a series of transparent latches completing a loop. In ZeBu, combinational loops can also be created because of a transformation of sequential elements on the clock path. Combinational loops can be state bearing or state-less. For example, an SR latch (with back-to-back NOR) is an example of state bearing combinational loop.

A Strongly Connected Component (SCC) is set of instances in which there exists a combinational path from every instance to every other instance of this set. A SCC can be visualized as a sub-circuit with external inputs and outputs. A loop can be considered as a trivial SCC.

In ZeBu, SCCs can introduce non-determinism at runtime. You can use this enhancement to reduce the number of SCCs present in the design within a set of user-defined constraints.

For more information, see the following:

- Use Model - Compilation
- Use Model - Runtime Debug

- Runtime Relaxation for Full Loop Breaking
- Limitations of the Removing SCC Feature

Use Model - Compilation

The Safe loop breaking option is enabled by default. In this mode, all loops including those deemed oscillatory and unknown that could be accommodated within the hardware resource constraints are broken. The following command is default:

```
ztopbuild -advanced_command {\
loop break -safe_break -consider_oscillatory_sccs=safe_break \
-override_unknown_behavior_scc=yes  -max_lut_overflow 5 \
-max_reg_overflow 5 -rtl=yes}
```

To have access to the original SCC at runtime, use the `-retain_SCC` switch. This feature is useful for debugging emulation mismatch between original and broken SCCs at runtime.

If you need to break an SCC that is not broken using the safe loop breaking option, use full loop breaking:

```
ztopbuild -advanced_command {\
loop break -full_break -max_lut_overflow 5 -max_reg_overflow 5 \
-rtl=yes}
```

When full loop breaking is enabled, loops are broken using a combination of the replication mechanism, which is used in safe loop breaking, and the fallback mechanism.

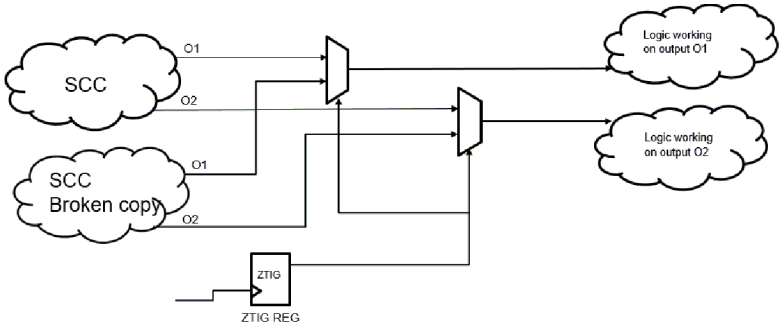
For larger loops, which are not handled using replication, the fallback mechanism is applied. In the fallback mechanism, break registers are only added to select output of an SCC based on heuristics as opposed to adding break registers on all SCC outputs done with replication.

Feedback

Use Model - Runtime Debug

During runtime, you can debug emulation failure caused by specific SCCs. For debugging, compare the original SCC and broken SCC to look for mismatches, as shown in the following figure:

Figure 13 Comparison Between Original and Reduced SCCs



Debug Procedure

There are two steps in debugging the emulation:

- For each output of the broken SCC, compare the corresponding output port in its original equivalent. By default, broken SCC is selected at runtime. To use the original copy, specify the SCC IDs (one per line) in the following file:

  `zcui.work/zebu.work/zebu_dbg_scc_mux_cntrl.txt`

  At runtime, to verify if the original copy was selected, refer to the `zServer.log` file:

  ```
  >> SCCDebugManager: Original value on SCC (<id>) is 1, setting it to
   '0' to disable it
  ```

- Probe the comparator output

  The detailed procedure is as follows:

  1. Use the probe file with `zSelectProbes` to obtain the register coordinates to perform Read-Back at runtime. The probe file is saved at the following location:

     `zcui.work/zebu.work/scc_probe.log`

  2. Use the following commands `zSelectProbes` to get ZRDB coordinates:

     ```
     load_probes zcui.work/zebu.work/scc_probe.log
     save_selection -f scc_selection.zrdb
     ```

3. To generate the ZTDB, use the following zRci script:

```
set fid [dump -file "dump.ztdb" -dynamic_probe]
    dump -fid $fid -load_selection scc_selection.zrdb
    dump -enable -fid $fid
```

4. Convert ZTDB to FSDB using the following command:

```
zConvertToFsdb --ztdb dump.ztdb --fsdb dump_fsdb --zebu-work
 zcui.work/zebu.work --timescale 2ns
nWave dump_fsdb.vf
```

   The captured signals contain the following prefix:

```
N<scc_id>_p<output_id>_<instance name>
```

   A logic-high value indicates a mismatch between the original SCC and the broken SCC.

Runtime Relaxation for Full Loop Breaking

With full loop breaking, if any SCC is broken using the fallback scheme, it might need runtime relaxation feature, which relaxes driver clock cycles based on user input. As a result, sufficient time is allowed for the SCC to settle. Using relaxation at runtime has runtime performance impact and might involve attempting multiple relaxation values; specifically for oscillatory SCC to get the SCC settled to desired value.

Specify the following in the `designFeatures` file to enable relaxation feature:

```
$zRelaxationCount=<n>
```

Runtime relaxation may be a temporary solution to eliminate non-determinism caused due to loops but should not be used permanently because it has significant runtime performance impact. The impact on performance increases as you increase the relaxation value.

Limitations of the Removing SCC Feature

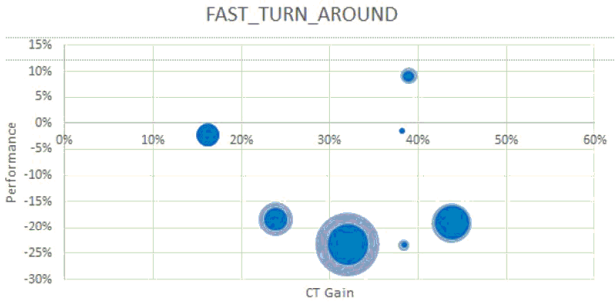
Loops traversing (D->Q) transparent latches are not supported.

Feedback

Achieving Compilation Objectives

ZeBu Server enables you to achieve compilation objectives, such as the following:

- Prioritizing performance at the cost of compilation time and capacity overhead
- Prioritizing the reduction of compilation time at the cost of capacity overhead and performance. The following figure shows the impact of reducing the compilation time for various designs. The light blue surrounding the dark blue represents the increase in the number of boards required after reducing the compilation time.



Enable Compilation Objective

To enable a compilation objective, set the following UTF command:

```
compile -objective [FAST_TURN_AROUND|PERFORMANCE]
```

Where:

- `FAST_TURN_AROUND`: Reduces compilation time at the cost of capacity overhead and increased runtime performance. Use this value for targeting cases where FPGA P&Rs are more than five hours.
- `PERFORMANCE`: Maximizes the throughput and the theoretical `driverClock` frequency (`zTime`) at the cost of capacity overhead and increased compilation time.

Compiling a Project Using zCui

zCui controls the entire compilation flow. It launches the necessary tools for compilation. **zCui** supports the Batch mode and GUI mode.

For more information, see the following topics:

- Compiling With zCui in the Batch Mode
- Compiling With zCui in the GUI Mode

Feedback

- Creating Runtime Archives After Compilation
- Verdi Database Tasks in zCui

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,

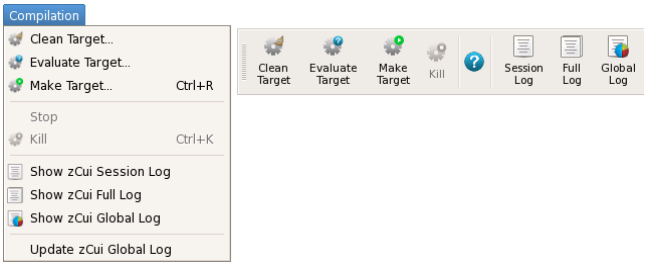
- `-c` launches compilation.
- `-n` runs **zCui** in the batch mode (no GUI).
- `<project_name>`.utf is the existing UTF project file containing information to compile a design for ZeBu using Unified Compile flow.
- `<zcui_uc_work_dir>` is the optional working directory for compilation; by default, this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

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 14 zCui Options



Feedback

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:

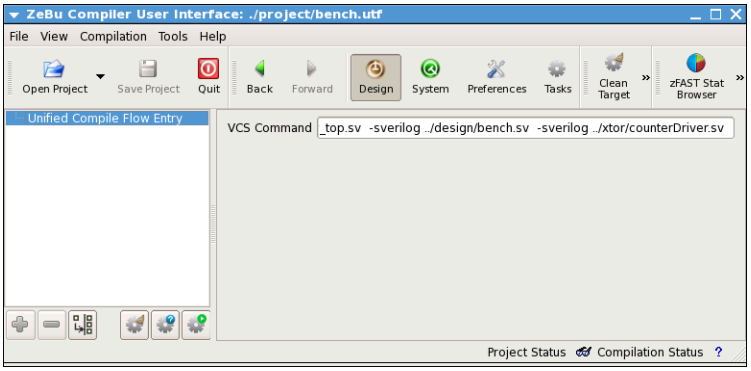
```
zCui -u <project_name>.utf [-w <zcui_uc_work_dir>]
```

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.

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:

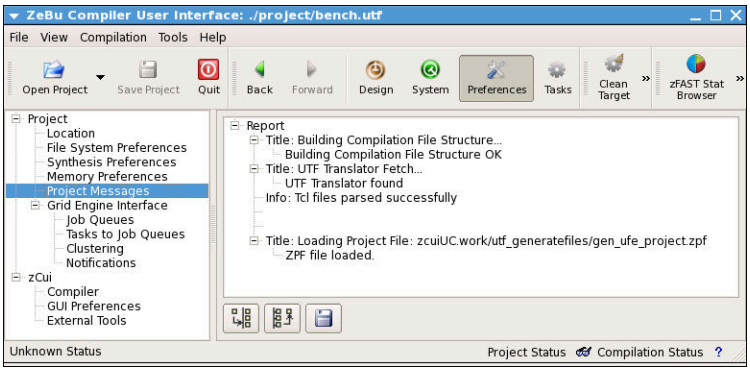
Figure 15 zCui GUI



Feedback

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:

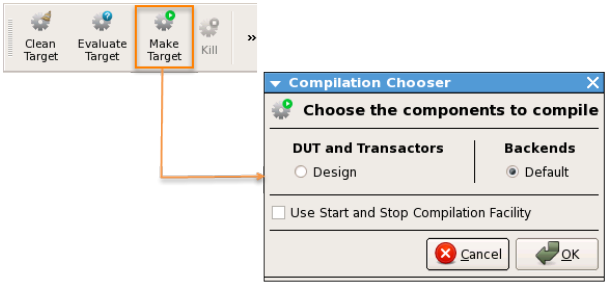
Figure 16 Project Messages Panel



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 17 Launching Compilation in zCui



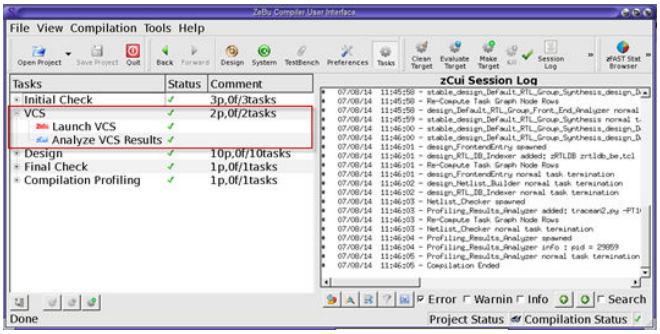
In the **Compilation Chooser** dialog, you must select one of the following:

- 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.

Feedback

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

Figure 18 Viewing the VCS Task in zCui



Creating Runtime Archives After Compilation

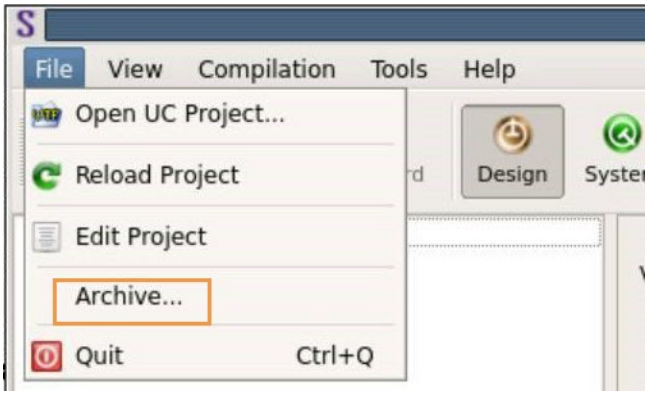
This section consists of the following topics:

- GUI Mode
- Batch Mode
- Usage

GUI Mode

After completing your `zCui` compilation, you can archive the runtime. To archive runtime in `zCui` GUI, click **File** > **Archive** as shown in the following figure:

Figure 19 Archive Menu in zCui



Feedback

You can now export the runtime files using the option as shown in the following figure:

Figure 20 Runtime Files Export

Batch Mode

To archive runtime in batch mode, use one of the following commands:

- `zCui -t <archive_name> -w <zcui.work path>`
- `zCui --usePigz <archive_name> -w <zcui.work path>`

where, `--usePigz` helps compress the files and create the archive at a faster speed.

Additionally, the `buildRuntimeArchive.sh` script is also available to build a runtime archive. Note that you can use the `buildRuntimeArchive.sh` script on read-only `zcui.work` (two other methods trying to log executed actions in the `zcui.work`).

Usage

```
buildRuntimeArchive.sh <archive file name> <zcui.work path> [-tmp]
 [-pigz] [-quiet]
```

Feedback

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}
```

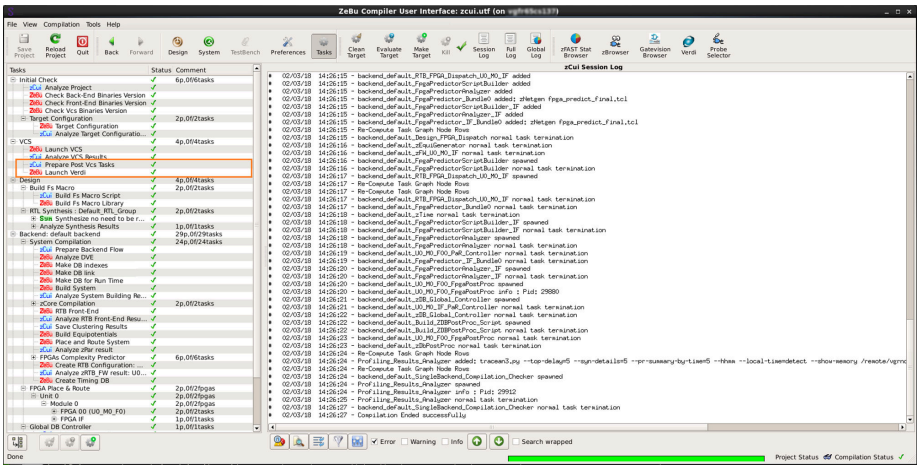
where, `newQueue` is another existing job queue or a new job queue created by `grid_cmd -queue {newQueue} -submit {…} -delete {…} -njobs …`

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 21 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 22 Viewing the Verdi Task in zCui



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

Feedback

Figure 23 Viewing the Verdi Task in 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
```

Important ZeBu Log Files

This section consists of the following topics:

- Analyzing Compilation Results Using zBatchExplorer
- Viewing Timing Information in the zTime Reports
- Native Compilation Profiler
- Multi-Process Flow
- Reporting XMR Information in a Design
- Viewing Consolidated Information Using zAudit
- Visualizing Compile-Time Data
- Handling Misfiring SVAs with Latches and Memories in Fanin
- Replicating Multiport Memories in zMem

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.

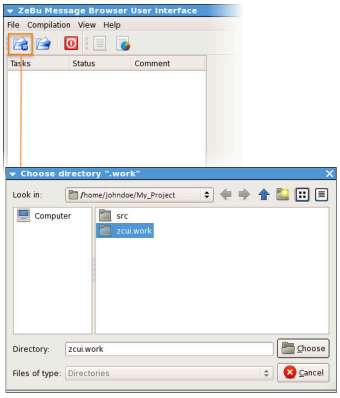
To launch **zBatchExplorer**, use the following command:

```
$ zBatchExplorer
```

Feedback

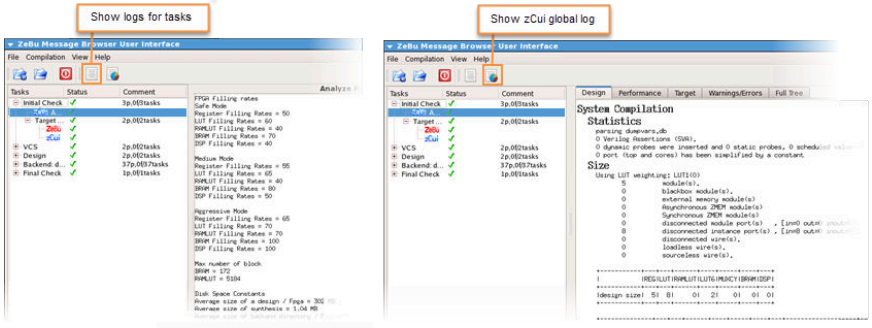
You can choose a working directory (for example, *zcui.work*) to open with the **zBatchExplorer** GUI as displayed in the following figure:

Figure 24 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 25 Selecting Log File and zCui Global File



Viewing Timing Information in the zTime Reports

`zTime` is the ZeBu tool which performs the timing-analysis and determines at which frequency the emulation project can be run. During ZeBu compilation, `zTime` is launched twice:

1. Before FPGA compilation, `zTime` provides a preliminary analysis for early indication whether your design meets your performance goal, so that you can identify areas that are hindering performance.
2. After FPGA compilation, `zTime` provides a final analysis, which reports the achievable emulation runtime frequency based on the actual mapping of the design, on the ZeBu system and FPGA Place & Route.

Feedback

For more information, see the following subsections:

- Accessing the zTime Reports
- Using zTime.html for Analysis
- Viewing Timing Summary
- Tracking Clock Domain ID for Timing Accuracy

Accessing the zTime Reports

Both the `zTime` pre-FPGA report and the `zTime` post-FPGA reports generate similar output. For pre-FPGA timing analysis, the output files are named as `zTime.log` and `zTime.html`. For post-FPGA timing analysis, the output files are named as `zTime_fpga.log` and `zTime_fpga.html`. All these files are stored in the `zebu.work` backend directory. The `.log` files are plain text files. The HTML format provides an interactive exploratory experience with enhanced display for a better user experience.

Apart from directly accessing these files from the backend directory, you can view the `zTime` report using the following utilities:

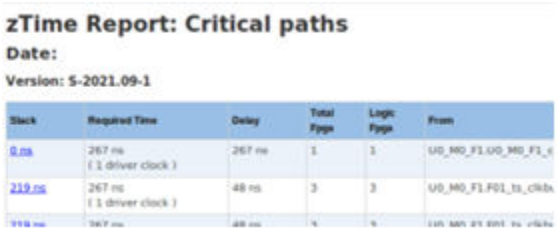
- `zCui`: Click the Performance tab of the Global Log View (see Compiling a Project Using zCui).
- `zBatchExplorer`: Click the Performance tab of the Global Log View (see <>.

Using zTime.html for Analysis

`zTime` HTML reports have similar information displayed in both pre- and post-FPGA reports. This format shows a tabular view of the logic that is crossed FPGA-by- FPGA, in particular critical routing paths, which limit the maximum emulation speed.

The header of the HTML file shows the software release of the ZeBu compiler:

Figure 26 zTime HTML Report: Header



Feedback

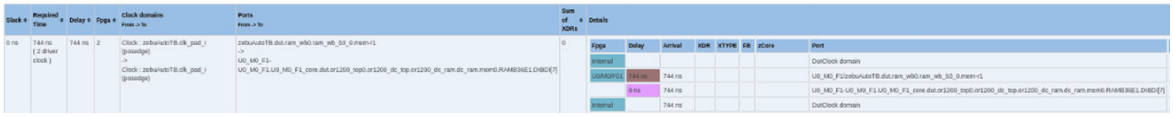
The `zTime` HTML reports provide enhanced display for easier analysis:

- The background color of the tables shows if it is pre-FPGA report (blue background) or post-FPGA report (green background).
- In all the tables of the `zTime` HTML reports, the data can be sorted based on selectors in the top cell of each column.
- The delays displayed in the report are color-coded for easier analysis, with specific colors highlighting the back-annotation resulting from FPGA compilation.

Description	Example HTML Report
Generic colors delays (in both reports)	Inter FPGA delay between 100ns and 150ns Inter FPGA delay between 150ns and 220ns Inter FPGA delay superior to 220ns
Color for paths with memories (in both reports)	Memory
Specific colors for back-annotation in post-FPGA	Missing back annotation : can lead to an optimistic timing Clock domain not found : can lead to a pessimistic timing No back annotation : can be ignored

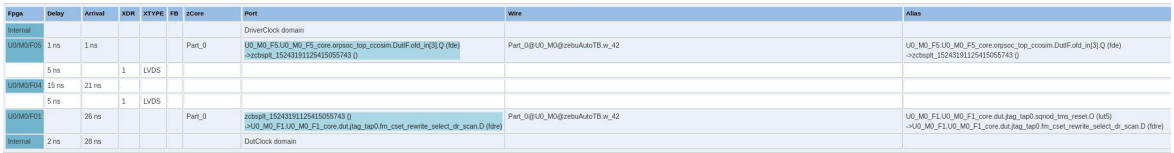
The following figure shows an example of post-FPGA report with sorting selectors in the top row and color-coded delays in the details:

Figure 27 Color Code in zTime Report



The **Alias** column in the report provides an easier identification of the paths in the design compared to the full path name.

Figure 28 Alias Column in zTime Report



Feedback

Viewing Timing Summary

At the end of the `zTime.log` report, the timing summary information, such as critical routing data path delay, longest memory period, and the ZeBu run frequency is provided. An example of a summary section is provided as follows:

```
#-----------------------------------------------------------------#
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
#-----------------------------------------------------------------#
```

The following snippet shows the section of the summary that is impacting the critical path:

```
#   step REPORT :   Critical routing data path delay : 71 ns
#   step REPORT :   . Constant part    : 66 ns
#   step REPORT :   . Multiplexed part : 5 ns
#   step REPORT :   . Memory latency part : 0 ns
```

Where:

- **Critical routing data path delay**: Specifies the total delay of the critical path
- **Constant part**: Specifies the delay related to the traversal of the data path
- **Multiplexed part**: Specifies the delay for the LVDS/MGMT traversal
- **Memory latency part**: Specifies the memory delay added if a memory in the critical path

Tracking Clock-Domain ID for Timing Accuracy

Amongst other details, the `zTime_fpga.html` file reports clock-domain mismatches that occur when `zCoreTiming` and `zFpgaTiming` do not associate the same clock-domain ID (identifier) to a given synchronous element. These mismatches are usually highlighted in pink color in the report.

To reduce such clock-domain mismatches, use the `zFpga_zCore_unify_clockdomain_resolution` variable as follows:

```
set_app_var zFpga_zCore_unify_clockdomain_resolution true
```

In addition, using this option reports correct clock ID and resets name in the “clock domain” columns of the `zTime.html` and `zTime_fpga.html` files.

In summary, using the variable for tracking the clock-domain ID reduces `zTime_fpga` pessimism and `zTime` pre-FPGA to post-FPGA gap.

Feedback

Native Compilation Profiler

The Native Compilation Profiler is introduced in **zCui** to generate a report for global integration and parsing output data for statistics. The Native Compilation Profiler is enabled at the end of the compilation.

For more information, see the following topics:

- Prerequisites

Prerequisites

To generate the reports, activate the profiling flag using the following UTF command:

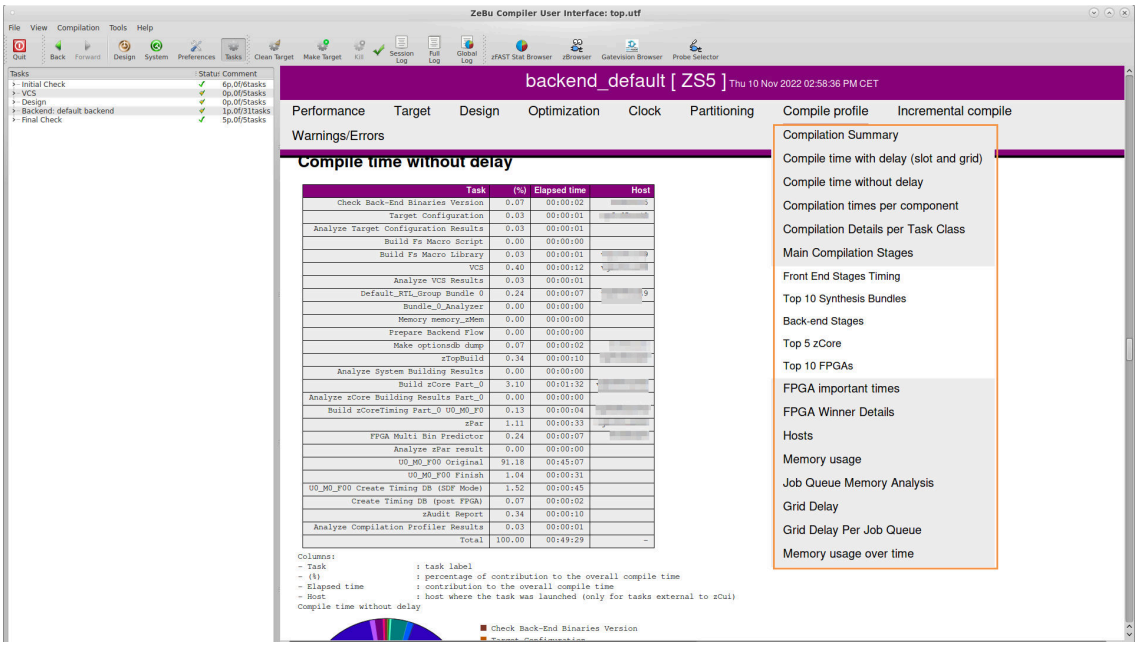
```
profile -compile true
```

The following reports are generated in the `zcui.work/zCui/log` directory:

- `compilationReport.txt`: Provides the textual output of the report.

The following figure shows the options available in the Native Compilation Profiler:

Figure 29 Native Compilation Profiler



This section describes the enhancements in the following:

- Compilation Summary
- Compile Time Table With Delay (Slot and Grid)
- Compile Time Without Delay

Compilation Summary

This section contains a brief summary of the compilation and a table summarizing main compilation stages. The following figure shows an example of the section:

Figure 30 Example of Compilation Summary

```
Compilation Summary

Last session wall clock times:
    FE     : 1h 50min 51s
    BE     : 19h 35min 17s
    Total  : 21h 26min 08s

All sessions cumulative compile times:
    without delay  : 4h 14min 51s
    with delay*    : 21h 25min 51s

*including slot and grid delays, difference with total wall clock time could be NFS wait or internal flow handling delays


           Phase               (%)     Slot delay   Grid delay   Elapsed time   Total # of jobs fired   Max memory consumption
------------------------------ ------- ------------ ------------ -------------- ---------------------- ----------------------
VCS                           13.04     00:00:00     00:06:08       00:33:11                      1                  5.42GB
zTopBuild                      1.66     00:00:00     00:00:13       00:04:14                      1                 17.82GB
zCoreBuild                     8.39     00:00:01     00:40:28       00:21:21                      1                 19.09GB
zCoreTiming                    0.36     00:00:02     00:08:55       00:00:55                      9                 10.73GB
zPar                           0.40     00:00:00     00:00:14       00:01:01                      1                  3.96GB
FPGA Compile                  69.28     00:00:00     14:09:03       02:56:19                 12 Orig                 31.56GB
Create Timing DB (post FPGA)   0.33     00:00:00     00:00:51       00:00:50                      1                  9.89GB
Other tasks                    3.79     00:00:01     02:04:13       00:09:39                     20                      -
Total                        100.00     00:00:04     17:11:18       04:14:29                     58                      -

Columns:
 - Phase                  : compilation phase
 - (%)                    : percentage of contribution to the overall compile time
 - Slot delay             : time waiting for a slot on the grid
 - Grid delay             : time between the task is spawned and the task is launched
 - Elapsed time           : contribution to the overall compile time
 - Total # of jobs fired  : number of jobs with same phase fired
 - Max memory consumption : job in the phase with max memory taken
```

Feedback

Compile Time Table With Delay (Slot and Grid)

This section contains the table listing the tasks that are in the critical path of the compilation including slot and grid delays. The tasks are listed at the beginning of the table. The following figure shows an example of the table:

Figure 31 Example of Compile Time Table With Delay

Compile time with delay (slot and grid)

Task	(%)	Slot delay	Grid delay	Elapsed time	Spawn Time	Finish Time	Host
Target Configuration	0.01	00:00:00	00:00:10	00:00:01	00:00:00	00:00:11	xxxxxxxxxx
Analyze Target Configuration Results	0.01	00:00:00	00:00:00	00:00:02	00:00:13	00:00:15	
Build Fs Macro Script	0.00	00:00:00	00:00:00	00:00:00	00:00:15	00:00:15	
Build Fs Macro Library	0.02	00:00:00	00:00:18	00:00:03	00:00:15	00:00:36	xxxxxxxxxx
VCS	13.04	00:00:00	00:06:08	00:33:11	00:00:36	00:39:55	xxxxxxxxxx
Analyze VCS Results	0.69	00:00:00	00:00:00	00:01:46	00:39:56	00:41:42	
Build Rhino Fsdb	0.26	00:00:01	01:08:28	00:00:40	00:41:43	01:50:51	xxxxxxxxxx
Prepare Backend Flow	0.29	00:00:00	00:00:00	00:00:44	01:50:51	01:51:35	
Make RTL DB indexes	1.00	00:00:00	00:39:25	00:02:33	01:51:35	02:33:33	xxxxxxxxxx
Make RTL DB link	0.02	00:00:00	00:00:12	00:00:03	02:33:33	02:33:48	xxxxxxxxxx
zTopBuild	1.66	00:00:00	00:00:13	00:04:14	02:33:48	02:38:15	xxxxxxxxxx
Analyze System Building Results	0.00	00:00:00	00:00:00	00:00:00	02:38:16	02:38:16	
Build zCore Part_0	8.39	00:00:01	00:40:28	00:21:21	02:38:17	03:40:06	xxxxxxxxxx
Analyze zCore Building Results Part_0	0.01	00:00:00	00:00:00	00:00:01	03:40:07	03:40:08	
Build zCoreTiming Part_0 U0_M0_F4	0.36	00:00:02	00:08:55	00:00:55	03:40:11	03:50:01	xxxxxxxxxx
zPar	0.40	00:00:00	00:00:14	00:01:01	03:50:02	03:51:17	xxxxxxxxxx
Analyze zPar result	0.02	00:00:00	00:00:00	00:00:03	03:51:18	03:51:21	
Classify FPGAs Script Builder	0.00	00:00:00	00:00:00	00:00:00	03:51:21	03:51:21	
Classify FPGAs Bundle 1	1.21	00:00:00	00:15:30	00:03:05	03:51:21	04:09:56	xxxxxxxxxx
Analyze Classify FPGAs Result	0.00	00:00:00	00:00:00	00:00:00	04:09:57	04:09:57	
U0_M0_F07 Original	69.28	00:00:00	14:09:03	02:56:19	04:09:59	21:15:21	xxxxxxxxxx
U0_M0_F07 Finish	0.20	00:00:00	00:00:09	00:00:31	21:15:23	21:16:03	xxxxxxxxxx
U0_M0_F07 Create Timing DB (SDF Mode)	2.74	00:00:00	00:01:13	00:06:59	21:16:05	21:24:17	xxxxxxxxxx
Create Timing DB (post FPGA)	0.33	00:00:00	00:00:51	00:00:50	21:24:18	21:25:59	xxxxxxxxxx
zAudit Report	0.05	00:00:00	00:00:01	00:00:07	21:26:00	21:26:08	xxxxxxxxxx
Total time	100.00	00:00:04	17:11:18	04:14:29	-	-	-

Columns:
- Task : task label
- (%) : percentage of contribution to the overall compile time
- Slot delay : time waiting for a slot on the grid
- Grid delay : time between the task is spawned and the task is launched
- Elapsed time : contribution to the overall compile time
- Spawn time : time spawning on the grid, taking the spawn time of the first task as reference
- Finish time : time finishing, taking the spawn time of the first task as reference
- Host : host where the task was launched (only for tasks external to zCui)

Compile Time Without Delay

The Native Compilation Profiler also contains the compile time without delay table, which corresponds to an infinite farm. Therefore, no waiting for slot time or waiting to be

Feedback

spawned and the criteria is to focus only the execution time, as shown in the following figure:

Figure 32 Example of Compile Time Table Without Delay

```
Compile time without delay


                Task                      (%)     Elapsed time      Host
------------------------------------ -------- -------------- ------------
Target Configuration                      0.01       00:00:01   xxxxxxxxxx
Analyze Target Configuration Results      0.01       00:00:02
Build Fs Macro Script                     0.00       00:00:00
Build Fs Macro Library                    0.02       00:00:03   xxxxxxxxxx
VCS                                      13.02       00:33:11   xxxxxxxxxx
Analyze VCS Results                       0.69       00:01:46
Build Rhino Fsdb                          0.26       00:00:40   xxxxxxxxxx
Prepare Backend Flow                      0.29       00:00:44
Make RTL DB indexes                       1.00       00:02:33   xxxxxxxxxx
Make RTL DB link                          0.02       00:00:03   xxxxxxxxxx
zTopBuild                                 1.66       00:04:14   xxxxxxxxxx
Analyze System Building Results           0.00       00:00:00
Build zCore Part_0                        8.38       00:21:21   xxxxxxxxxx
Analyze zCore Building Results Part_0     0.01       00:00:01
Build zCoreTiming Part_0 U0_M0_F7         0.50       00:01:17   xxxxxxxxxx
zPar                                      0.40       00:01:01   xxxxxxxxxx
Analyze zPar result                       0.02       00:00:03
Classify FPGAs Script Builder             0.00       00:00:00
Classify FPGAs Bundle 1                   1.21       00:03:05   xxxxxxxxxx
Analyze Classify FPGAs Result             0.00       00:00:00
U0_M0_F07 Original                       69.18       02:56:19   xxxxxxxxxx
U0_M0_F07 Finish                          0.20       00:00:31   xxxxxxxxxx
U0_M0_F07 Create Timing DB (SDF Mode)     2.74       00:06:59   xxxxxxxxxx
Create Timing DB (post FPGA)              0.33       00:00:50   xxxxxxxxxx
zAudit Report                             0.05       00:00:07   xxxxxxxxxx
Total                                   100.00       04:14:51   -

Columns:
 - Task                  : task label
 - (%)                   : percentage of contribution to the overall compile time
 - Elapsed time          : contribution to the overall compile time
 - Host                  : host where the task was launched (only for tasks external to zCui)
```

Multi-Process Flow

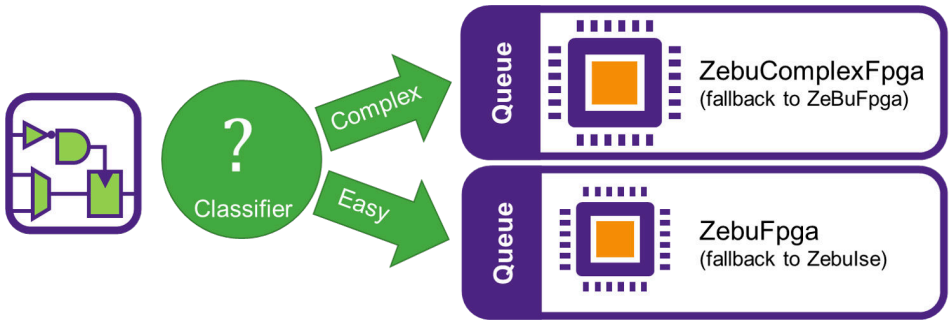
The Multi-Process Flow (MPF) feature parallel processes per SLR level netlist to improve Vivado compile time. For non-MPF, Vivado runs one process for all the SLRs.

Vivado Placer and Router in MPF mode ensures that placement and routing in each SLR happens parallelly as a separate process.

- Single machine MPF: All parallel SLR placement and routing happens on the same machine.
- Multi-machine MPF: Host machine executes a parent SLR job while the child SLR jobs land on different machines in the grid such as LSF, UGE, netbatch, and so on.

Feedback

Figure 33 Smart Job Scheduling for Complex FPGAs



MPF feature requires Vivado 2020.1 or later versions.

To enable MPF, use the following UTF commands:

- **FPGA Complexity Predictor**

```
fpga -complexity_predictor true
```

- **MPF**

```
fpga -multi_process OFF|SINGLE_HOST | MULTI_HOST
 -multi_process_activation_percent_limit<Integer from 0-100>
 -multi_process_params {INCL_PARFF=true}
```

  - `SINGLE_HOST` runs single machine MPF where all the child processes are spawned on the same machine.
  - `MULTI_HOST` runs multi-machine MPF with child jobs spawned on a different machine.
  - `Multi_process_activation_percent_limit` selects the FPGAs to be launched in MPF, which is controlled by the maximum limit specified. There is no requirement on minimum value, which indicates that the ZeBu Compiler can launch 0-N jobs if the `multi_process_activation_limit` is set to N.

    Note:

    Higher value of the `-multi_process_activation_percent_limit` parameter can overload the grid usage.

  - `-multi_process_params {INCL_PARFF=true}` enables MPF for PARFF jobs through **zCui**.

Grid Queue Configuration

For grid queue configuration, use the following command:

```
grid_task_association -task {<zebu_task_name>} -queue
 {<given_queue_name>}
```

Feedback

```
grid_cmd -queue ZebuComplexFpga -submit { <user-defined command> }
 -delete {} -njobs <num>
grid_cmd -queue ZebuFpga -submit { <user-defined command> } -delete {}
 -njobs <num>
```

- `ZebuIse/ZebuFpga`: The default queue for FPGA compile. Every FPGA P&R task is assigned to this queue and submitted to its associated machines. The complexity score of each FPGA decides which queue to be assigned.
- `ZebuComplexFpga`: The `ZebuComplexFpga` queue is used for hard FPGA compile. This queue is used with more powerful machines to accelerate the compilation of hard FPGAs by assigning them to powerful machines to free up the longpole.

With MPF activation, **zCui** forwards Submit and Kills commands specified in the job queue for FPGA P&R task by defining `VIVADO_REMOTE_MPF_SUBMIT` and `VIVADO_REMOTE_MPF_KILL` environment variables in interactive mode by default (qrsh).

For MPF, you must use only non-interactive remote commands. Therefore, to override the **zCui** default setting, you must set the following environment variables to use non-interactive grid commands (`qsub`, `bsub`):

- Multi-machine MPF remote submission

  ```
  VIVADO_REMOTE_MPF_SUBMIT <corresponding string for grid command>
  ```

- Multi-machine MPF remote termination

  ```
  VIVADO_REMOTE_MPF_KILL < corresponding string for grid command>
  ```

For better MPF performance, the following grid configuration is suggested for more efficient grid usage

- `ZebuComplexFpga` queue should be on a FAST queue with a host having fastest host on the grid.
- `ZebuFpga` queue should be on a NORMAL queue so that it can use a slower or a faster host.

Reporting XMR Information in a Design

XMRs in a design can be captured at the beginning of Unified Front-end (UFE) stage during compilation. Therefore, the XMRs have been transformed by elaboration and might have some differences with the original XMRs in the source RTL. The following information of each XMR is collected and captured in a log file (`xmr_debug.log`).

- Owner module of XMR (that is, the module in which the XMR resides)
- XMR string
- Uses of XMR and XMR’s source locations (file name and line number)

- Target module of XMR
- Target signal of XMR and its source location (file name and line number)

The XMR information is captured in the following format:

```
MODULE: {MODULE NAME}
       XMR: {XMR EXPR}
        USE    : {XMR_USE} {FILE_PATH + LINE_NUMBER}
    TARGET MODULE  : {TARGET_MODULE}
    TARGET SIGNAL  : {TARGET_SIGNAL_EXPR} {FILE_PATH + LINE_NUMBER}
```

Since a module usually has multiple XMRs, XMRs are organized based on their owner modules.

To capture XMRs in a design, add the following UTF command to your UTF file:

```
design -report xmr
```

The log file (`xmr_debug.log`) is captured in `zcui.work/vcs_splitter/simon.out`.

Example

```
File: dut.sv
01: module dut();
02: wire x;
03: wire y;
04: test  u1();
05: sub  s1(.in(u1.temp2));
06:
07: endmodule
08:
09: module test (in, out);
10: input in;
11: output out;
12: reg temp, temp2;
13: assign out = dut.x;
14: assign temp = dut.x;
15: initial begin
16: force dut.y = dut.s1.temp;
17: force temp = in;
18: end
19: endmodule
20:
21: module sub(in, out);
22: input in;
23: reg temp;
24: output out;
25: assign out = in;
26: endmodule
```

The XMR information is captured in the log file (`xmr_debug.info`) as follows:

```
MODULE: dut
    XMR: u1.temp2
      USE  : sub s1(.in(u1.temp2)); (dut.sv:5)
      TARGET MODULE  : test
      TARGET SIGNAL     : reg temp; (dut.sv:23)

MODULE: test
   XMR: dut.x
     USE : assign out = dut.x; (dut.sv:13)
     TARGET MODULE  : dut
     TARGET SIGNAL :  wire x; (dut.sv:2)
   XMR: dut.x
     USE : assign temp = dut.x; (dut.sv:14)
     TARGET MODULE  : dut
     TARGET SIGNAL : wire x; (dut.sv:2)
   XMR: dut.y
     USE : force dut.y = dut.s1.temp; (dut.sv:16)
     TARGET MODULE  : dut
     TARGET SIGNAL : wire y; (dut.sv:3)
   XMR: dut.s1.temp
     USE : force dut.y = dut.s1.temp; (dut.sv:16)
    TARGET MODULE  : sub
    TARGET SIGNAL : reg temp; (dut.sv:23)
```

Viewing Consolidated Information Using zAudit

The `zAudit` tool consolidates information from the log files in `zcui.work`. The tool provides a summary of the consolidated information. Therefore, the tool should be used before analyzing with the Verdi tool.

After the compilation has completed, the report generated by the `zAudit` tool is available at the following location:

`zcui.work/zCui/log/backend__zAuditReport.log`.

The `zAudit` tool provides the following information:

- **Reporting**: Provides a summary of consolidated information using `summary`, `fpga`, `technology`, `memory`, `mapping`, `combinatory`, `timing`, and `zpar` commands.
- **Recommendation** to improve compilation and performance using `diagnostic` and `bottleneck` commands. The recommendation concerns all ZeBu tools (`zAudit`, `FPGA`, `zTopBuild`, `vivado` and so on.) and performance.

After the completion of compilation, the zAudit tool generates a report and is available at the following location:

`zcui.work/zCui/log/backend__zAuditReport.log`

For more information, see the following subsections:

- Prerequisites
- Launching the zAudit Tool
- zAudit Command Line
- zAudit Examples

Prerequisites

The following configuration file is required for `zAudit`:

```
$ZEBU_ROOT/etc/zAudit/defaults.ini
```

This file contains default settings, such as the following:

- Weight of the fields for memory scoring
- Default number of rows that appear in the tables

You can edit these settings and pass the file to the `zAudit` tool using the following command: `zAudit -config-file ./my_defaults.ini`

Launching the zAudit Tool

To view the list of commands support by `zAudit`, use the following:

```
zAudit -help
or
zAudit -h
```

To view the detailed help for a given command, use the following:

```
zAudit <zAudit_command> -h
```

zAudit Command Line

To invoke `zAudit`, use the following command:

```
$ZEBU_ROOT/bin/zAudit
```

Common `zAudit` use models are as follows:

- To view the information in `zAudit`, use the following:
  - When the compilation database is needed, use the following:

    ```
    zAudit -z <zcui.work> <zAudit_command>
    ```

  - When a separate database is needed for compilation analysis, use the following:

    ```
    zAudit -zcui_work <zcui.work> --db ~/zAudit.db <zAudit_command>
    ```

The available zAudit commands are listed in the following table:

Table 7 zAudit Commands

Compilation Analysis	**<zAudit_command>**
Global Compilation Summary	*summary*
Activated Technologies	*technology*
FPGA Metrics	*fpga*
Memory Costs	*memory*
Mapping Information	*mapping*
Information on zPar	*zpar*
Level of Combinational Logic	*combinatory*
SCC information	*loop*
Timing Statistics	*timing*
Information on User clocks	*clock*
Warnings & Recommendations	*warning*
Timing Bottlenecks	*bottleneck*
Improvement Suggestions	*diagnostic*
Displays all starred reports	*report*

zAudit Examples

See the following examples:

- Example 1: View General Statistics
- Example 2: View FPGA Metrics
- Example 3: Sort and View Readable Memories
- Example 4: View Memories using More Than 50 BRAM
- Example 5: Core-level Mapping Statistics
- Example 6: Level of Combinatory Logic

Feedback

- Example 7: Timing Statistics and Bottleneck
- Example 8: View Diagnostic Statistics

Example 1: View General Statistics

To view summary of general statistics such as `zTime` frequency, design size, FPGA compile times, use the following command:

```
zAudit -z zcui.work summary
```

Summary

Category	Details
General	Platform: ZS4, ZeBu Release: V-2024.03-1, Vivado Release: 2022.1_AR76726
Memories	Total: 169, By Type: {BRAM: 151, RAMLUT: 15, ZRM: 3}, With # Ports: {1-3: 166, 4-7: 0, 8-15: 0, 16-31: 2, 32+: 1}
Clocks	Fetch Mode: yes, Excalibur: no, RTL Clocks: no, CDP Replication: no, Localization Strategy: {zTopBuild: N/A, zCores: {PARTIAL: Part_0}}
Size	REG: 5,179,006, LUT: 9,155,704, LUT6: 3,379,212 (36% of LUT), URAM: 0, BRAM: 5,262, RAMLUT: 177,976, DSP: 123, MUXCY: 181
Mapping	Units: 1, Modules: 1, Estimated FPGAs: 10 Avg Fill: {REG: 7%, LUT: 24%, BRAM: 14%} Max FPGA Fill: {REG: 14%, LUT: 45%, BRAM: 38%} Avg CUT in zTopBuild: {FPGA: 8,669}, Max CUT: {FPGA: 14,066}
Routing	routerType: EZCR, routing Effort: default, pinAssignmentEffort: zap, SocketMode: component, SAG: disabled
Timing	driverClk: 1428 kHz (700 ns) Pre-Post delta before rounding: {driverClk: 1952 -> 1462 (-25%)} bottleneck: routing data paths : 684ns, Longest Delays: {data: 684 ns, memory: 78 ns}, #MGT in top path: {data: 0, filter: 0}
FPGA	Total Num. of completed FPGAs: 12 Original Winners: 12 Total number of FPGA with PDM enabled: 0 Total Number of PARFF: 0 Num of Failed FPGAs: 0, Failed FPGAs: [] Num FPGA compile ongoing: 0, FPGA compile ongoing: [] Long Pole FPGA Compile Times: {U0_M0_F08: 264, U0_M0_F10: 185, U0_M0_F11: 174, U0_M0_F09: 155, U0_M0_F05: 149} FPGA compile status: PASS
UPF	Yes, Full UPF
XTOR	Yes, XTOR Nbr: 6, Ports IN Nbr: 85, Ports OUT Nbr: 86
DPI	zDPI, Call Nbr.: 135, Bits: 27186, Max size: 793
Compile Time	Front-End: 29m:48s Back-End: 1h:01m:29s FPGA-Compile: 4h:26m:04s Others: 26m:34s Total: 6h:23m:55s

The summary of the general statistics displays the following information:

- **General**: Hardware platform and software version
- **Memories**: Number of memory instances, type and number of ports
- **Clocks**: Clock handling and performance technique setup used during compilation
- **Size**: Design size such as number of LUT, RAM, MUXCY, DSP, and so on used during compilation
- **Mapping**: Number of units/modules/FPGAs with fill rate and cut information
- **Routing**: System routing

- **Timing**: Timing estimation
- **FPGA**: Overview on FPGA metric and compile status
- **Compilation**: Compilation time details for the main steps

An HTML version of general statistics table is generated. It has links to compile profiler and log files from where the data is extracted:

Basic info

Keyword	Description
zAudit	ZeBu Compile Analysis, Diagnostics, and Recommendations
zcui_work	path_zcui.work/zcuiUC.work
zebu_work	/backend_default
zAudit db	/zAudit.db
Global Compile Log	Path to Global Log
Stage	Completed

Summary

Category	Details	Log file Path
General	Platform: ZS4, ZeBu Release: V-2024.03-1, Vivado Release: 2022.1_AR76726	zCui.log
Memories	Total: 169, By Type: {BRAM: 151, RAMLUT: 15, ZRM: 3}, With # Ports: {1-3: 166, 4-7: 0, 8-15: 0, 16-31: 2, 32+: 1}	Path to zMem dir
Clocks	Fetch Mode: yes, Excalibur: no, RTL Clocks: no, CDP Replication: no, Localization Strategy: {zTopBuild: N/A, zCores: {PARTIAL: Part_0}}	zTime.log
Size	REG: 5,179,006, LUT: 9,155,704, LUT6: 3,379,212 (36% of LUT), URAM: 0, BRAM: 5,262, RAMLUT: 177,976, DSP: 123, MUXCY: 181	Bundle_*.log
Mapping	Units: 1, Modules: 1, Estimated FPGAs: 10 Avg Fill: {REG: 7%, LUT: 24%, BRAM: 14%} Max FPGA Fill: {REG: 14%, LUT: 45%, BRAM: 38%} Avg CUT in zTopBuild: {FPGA: 8,669}, Max CUT: {FPGA: 14,066}	zTopBuild.log
Routing	routerType: EZCR, routing Effort: default, pinAssignmentEffort: zap, SocketMode: component, SAG: disabled	zPar.log
Timing	driverClk: 1428 kHz (700 ns) Pre-Post delta before rounding: {driverClk: 1952 -> 1462 (-25%)} bottleneck: routing data paths : 684ns, Longest Delays: {data: 684 ns, memory: 78 ns}, #MGT in top path: {data: 0, filter: 0}	zTime_fpga.log
FPGA	Total Num. of completed FPGAs: 12 Original Winners: 12 Total number of FPGA with PDM enabled: 0 Total Number of PARFF: 0 Num of Failed FPGAs: 0, Failed FPGAs: [] Num FPGA compile ongoing: 0, FPGA compile ongoing: [] Long Pole FPGA Compile Times: {U0_M0_F08: 264, U0_M0_F10: 185, U0_M0_F11: 174, U0_M0_F09: 155, U0_M0_F05: 149} FPGA compile status: PASS	
UPF	Yes, Full UPF	
XTOR	Yes, XTOR Nbr: 6, Ports IN Nbr: 85, Ports OUT Nbr: 86	
DPI	zDPI, Call Nbr.: 135, Bits: 27186, Max size: 793	
Compile Time	Front-End: 29m:48s Back-End: 1h:01m:29s FPGA-Compile: 4h:52m:38s Total: 6h:23m:55s	compilationReport.txt

Example 2: View FPGA Metrics

To view the FPGA metrics, use the following command:

```
zAudit -z zcui.work fpga
```

The FPGA metrics displays the following information for every FPGA:

- FPGA index
- Filling rates for LUTs/FFS/LUT6
- Nb of control sets

Feedback

- Number of QiWC
- FPGA total compile time
- Vivado placer compile time
- Vivado router compile time
- FPGA IO cut
- Winner PARFF strategy
- PARFF_TARGET
- Complexity score
- Binary score for multi-binning

zAudit also provide a detailed information for every FPGA using the following command:

```
zAudit -z <zcui.work> fpga <Un_Mn_Fxy>
```

The zAudit command provides the following information in seven different tables:

- **FPGA compilation process statistics**

LEVEL-2 FPGA HOST INFO

Metric	Value	Optimum_values
FPGA	U0_M0_F04	
Place(LoadFactor)	0.59	<1.0
Route(LoadFactor)	0.48	<1.0
MaxLOADFactor	1.10	<1.0
Place(MemFree)	91	>=64 GB
Route(MemFree)	80	>=64 GB
Place(MemAvail)	98	>=48 GB
Route(MemAvail)	89	>=48 GB
CPU_Accelerate_Factor(placer)	1.81	>1.75
CPU_Accelerate_Factor(Router)	1.87	>1.75

- **FPGA Filling-rates**

LEVEL-2 FPGA UTILIZATION

Metric	post_link	post_place
FPGA	U0_M0_F04	U0_M0_F04
LUTs(%)	52.54	48.52
Registers(%)	19.60	18.78
BRAMS	1392	1383
LUT6(%)	46.22	47.94
Control_Set	26513	25368
LUTRAMs	37591	18273
URAMs	254	254
DSPs	254	254
QIWC	775047	775047
MaxSLRLUTs(%)	51.59	51.59
IOCUT	145	145
Winner	Original	Original

- **FPGA congestion**

Feedback

LEVEL-2 FPGA CONGESTION

Metric	Value	Optimum_values
FPGA	U0_M0_F04	
North	2	<6
South	2	<6
East	2	<6
West	3	<6
North(Short)	3	<6
South(Short)	2	<6
East(Short)	3	<6
West(Short)	2	<6
MaxCong	3	<6
MaxCong(Short)	3	<6

- **FPGA compile time**

LEVEL-2 FPGA COMPILE_TIME(MINS)

Phases(Major)	Value
FPGA	U0_M0_F04
zNetgen	10
Link_Design	10
Netlist_Analysis	2
Read_Const	4
Opt_Design	9
Place_Design	65
Post_Place	1
Route_Design	75
Timing_Analysis	3
Bitstream	11
Location_Building	3
Total_Vivado	181
Total_CT	195.933

- **FPGA timing stats**

LEVEL-2 FPGA TIMING

Metric	Value
FPGA	U0_M0_F04
Source	wc_ip_top/wrapper/qiwc_ip_cluster/cluster_qiwc_ip[2]_qiwc_ip_0/u.xst_wrapper/qiwc_ip_sub_header_fifo/wc_fifo_256_last_aware_0/u.xst_wrapper/din_r_reg[6]/C
Destination	wc_ip_top/wrapper/qiwc_ip_cluster/cluster_qiwc_ip[2]_qiwc_ip_0/u.xst_wrapper/qiwc_ip_sub_header_fifo/wc_fifo_256_last_aware_0/u.xst_wrapper/wc_fifo_256/SRL_bit0[6]/D
Logic_Levels	0
Max_fanout	1
Required_time	9.524
Requirement	0.000
Skew(ns)	0.146
Slack(ns)	0.011
Clock_Root(Src)	X4Y7
Clock_Root(Dest)	X4Y7
DataPathDelay(ns)	0.515
WNS(Post_Place(ps))	562
WHS(Post_Place(ps))	-474
DataPathLogicDelay(ns))	0.123
DataPathNetDelay(ns)	0.392

- **Information about MPF - activated**

LEVEL-2 MPF INFO

Metric	Value
FPGA	U0_M0_F04
Single_machine	0
Multi_machine	0

Feedback

- **FPGA characteristics**

```
    LEVEL-2 FPGA CHARACTERISTICS
+------------------------+------------+
| Metric                 | Value      |
+========================+============+
| FPGA                   | U0_M0_F04  |
| total_trigger_bits     | 0          |
| num_trigger_cells      | 0          |
| max_bits_per_cell      | 0          |
| trigger_cell_instance  | NA         |
+------------------------+------------+
```

zAudit also provide a graphical presentation of FPGA metrics using the following command:

```
zAudit -z <zcui.work> fpga -plot <report_name/html>
```

Example 3: Sort and View Readable Memories

To sort and view the ‘readable’ memories first, use the following command:

```
zAudit -z zcui.work memory
```

Feedback

zAudit memory command displays the following tables:

```
                                    Memory-Threshold-Settings
+-----------------------------------+-------------------------------+---------------------------------------+
| memSizeThreshold(default --> 2048) | memPortThreshold(default --> 128) | memSizeThresholdHard(default --> 50000) |
+===================================+===============================+=======================================+
|              2048                 |              128              |                50000                  |
+-----------------------------------+-------------------------------+---------------------------------------+
```

```
      Memory-User-Settings
+----------------------+--------+
|       settings       | value  |
+======================+========+
|   memSizeThreshold   | 2048   |
|   memPortThreshold   |  128   |
| memSizeThresholdHard | 50000  |
+----------------------+--------+
```

The following information is displayed for every memory:

- Module name
- Module elaboration name
- Variable name
- Width
- Depth
- Inferred type
- Inferred reason
- Reset type
- Init type

Feedback

Memory-Names (From Front end - Unified compile)

zmem_name	module_name	module_elab_name	variable name
top_4_no_SWM_0000_ZMEM_top_collected_signals2_r	top_4_no_SWM	top_4_no_SWM_0000	top_collected_signals2_r
top_4_no_SWM_0001_ZMEM_top_collected_signals2_r	top_4_no_SWM	top_4_no_SWM_0001	top_collected_signals2_r
top_4_no_SWM_ZMEM_top_collected_signals2_r	top_4_no_SWM	top_4_no_SWM	top_collected_signals2_r
top_4_SWM_ZMEM_top_collected_signals2_r	top_4_SWM	top_4_SWM	top_collected_signals2_r
top_4_no_SWM_0000_ZMEM_top_collected_signals_r	top_4_no_SWM	top_4_no_SWM_0000	top_collected_signals_r
top_4_no_SWM_0001_ZMEM_top_collected_signals_r	top_4_no_SWM	top_4_no_SWM_0001	top_collected_signals_r
top_4_no_SWM_ZMEM_top_collected_signals_r	top_4_no_SWM	top_4_no_SWM	top_collected_signals_r
top_4_SWM_ZMEM_top_collected_signals_r	top_4_SWM	top_4_SWM	top_collected_signals_r
ICache_ZMEM_data_arrays_0_0	ICache	ICache	data_arrays_0_0
ICache_ZMEM_data_arrays_0_1	ICache	ICache	data_arrays_0_1

Memory-Names (From Front end - Unified compile)

zmem_name	width	depth	inferred type	inferred_reason	reset type	init type
top_4_no_SWM_0000_ZMEM_top_collected_signals2_r	1	51114	REG	write port could not be extracted for this one dimensional signal		
top_4_no_SWM_0001_ZMEM_top_collected_signals2_r	1	51114	REG	write port could not be extracted for this one dimensional signal		
top_4_no_SWM_ZMEM_top_collected_signals2_r	1	51114	REG	write port could not be extracted for this one dimensional signal		
top_4_SWM_ZMEM_top_collected_signals2_r	1	51114	REG	write port could not be extracted for this one dimensional signal		
top_4_no_SWM_0000_ZMEM_top_collected_signals_r	1	29208	REG	write port could not be extracted for this one dimensional signal		
top_4_no_SWM_0001_ZMEM_top_collected_signals_r	1	29208	REG	write port could not be extracted for this one dimensional signal		
top_4_no_SWM_ZMEM_top_collected_signals_r	1	29208	REG	write port could not be extracted for this one dimensional signal		
top_4_SWM_ZMEM_top_collected_signals_r	1	29208	REG	write port could not be extracted for this one dimensional signal		
ICache_ZMEM_data_arrays_0_0	32	512	ZMEM	Array size is greater than memSizeThreshold	None	All_0
ICache_ZMEM_data_arrays_0_1	32	512	ZMEM	Array size is greater than memSizeThreshold	None	All_0

Note:

zAudit memory command does not always display the list of all memories in the design.

To display the list of all memories in a design, use `--rows` option

```
zAudit -z <zcui.work> memory  --rows 50
```

The value of `--rows` option corresponds to the number of memories in the design.

It is also possible to have a different reporting on the memory using the following options:

- `--rows INTEGER RANGE`: maximum number of rows to display in a table
- `--module_name TEXT`: print memory based on `module_name`. This option is supported with default mode only. This option is not supported with `--stage=synth` or `--stage=backend`.
- `--zmem_name TEXT`: print memory based on `zmem_name`. This option is supported with default mode only. This option is not supported with `--stage=synth` or `--stage=backend`.
- `--depth TEXT`: print memory based on memory depth. This option is supported with default mode only. This option is not supported with `--stage=synth or --stage=backend`.

Example usage: `memory --depth yes`.

Feedback

- `--stage TEXT`: reporting stage. Possible values are `--stage=backend` and `-stage=synth`. If stage is not provided, zAudit reports memory from vcs/unified compile.

Example usage:

```
zAudit --db <.db> memory
zAudit --db <.db> memory --stage  backend
zAudit --db <.db> memory --stage synth
```

- `--sort TEXT`: `--sort` timing table by column name. The valid names are `module_elab_name`, `width`, `depth`, `wr`, `rd`, `rw`, `async`, `type`, `reg`, `lut`, `bram`, `uram`, and `score`.
- `--sort(--stage=backend)`: timing table by column value. The valid names are BRAM, RAMLUT, ZRM
- `--filter TEXT`: filter option is supported with `--stage=synth` only. It reports memory table results based on column values. The parameter is `column_name<op>value`, where valid `<op>` (operators) are `[= != < > <= >=]`. Example usage:`--filter depth>5`. For yes/no fields, the only valid usage is `async=<yes/no>`. The parameters with < or > typically must be quoted to avoid shell redirect.

Example 4: View Memories using More Than 50 BRAM

To view the memories utilizing more than 50 BRAM sorted by the memory width in `zAudit`, use the following command:

```
zAudit -z zcui.work memory --sort width --filter "bram>50"
```

Example 5: Core-level Mapping Statistics

To view the `zCoreBuild` post-split design statistics, use the following command

```
zAudit -z <zcui.work> mapping
```

zAudit mapping command displays the following table:

zTopBuild Post-split design statistics

Core	IO	REG	REG%	LUT	LUT%	LUT6	LUT6%	BRAM	RAMLUT	DSP	MUXCY
Part_0	0	1,784,368	2%	1,663,121	5%	701,102	2%	2,560	0	576	142
Sum	0	1,784,368		1,663,121		701,102		2,560	0	576	142
Mean	0	1,784,368	2%	1,663,121	5%	701,102	2%	2,560	0	576	142
Max	0	1,784,368	2%	1,663,121	5%	701,102	2%	2,560	0	576	142

The previous table displays the following information for every core:

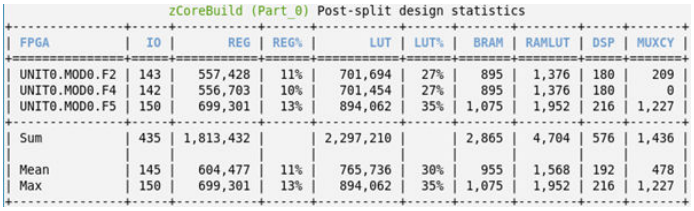
- Core identifier
- Number of IO
- Number and filling rate of the registers

Feedback

- Number and filling rate of the LUT
- Number and filling rate of the LUT6
- Number of BRAM/RAMLUT/DSP/MUXCY
- The total, mean and max per core of every resource type

zAudit also provides a detailed information for every core using the following command:

```
zAudit -z <zcui.work> mapping <core_identifier>
```



zCoreBuild (Part_0) Post-split design statistics

FPGA	IO	REG	REG%	LUT	LUT%	BRAM	RAMLUT	DSP	MUXCY
UNIT0.MOD0.F2	143	557,428	11%	701,694	27%	895	1,376	180	209
UNIT0.MOD0.F4	142	556,703	10%	701,454	27%	895	1,376	180	0
UNIT0.MOD0.F5	150	699,301	13%	894,062	35%	1,075	1,952	216	1,227
Sum	435	1,813,432		2,297,210		2,865	4,704	576	1,436
Mean	145	604,477	11%	765,736	30%	955	1,568	192	478
Max	150	699,301	13%	894,062	35%	1,075	1,952	216	1,227

The previous table provides the following information for every FPGA:

- FPGA index
- Number of IO
- Number and filling rate of the registers
- Number and filling rate of the LUT
- Number and filling rate of the LUT6
- Number of BRAM/RAMLUT/DSP/MUXCY
- The total, mean and max per FPGA of every resource type

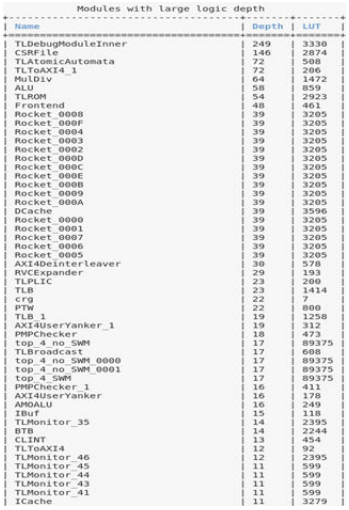
Example 6: Level of Combinatory Logic

To view the combinatory modules with large logic depth and its distribution, use the following command:

```
zAudit -z <zcui.work> combinatory
This command provides two tables:
```

Feedback

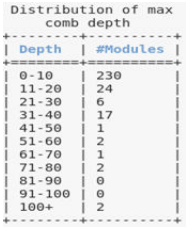
- Modules with large logic depth: It contains the list of combinatory modules, its depth and number of LUTs



Modules with large logic depth

Name	Depth	LUT
TLDebugModuleInner	249	3330
CSRFile	146	2874
TLAtomicAutomata	72	508
TLToAXI4_1	72	206
MulDiv	64	1472
ALU	58	859
TLROM	54	2923
Frontend	48	461
Rocket_0008	39	3205
Rocket_000F	39	3205
Rocket_0004	39	3205
Rocket_0003	39	3205
Rocket_0002	39	3205
Rocket_000D	39	3205
Rocket_000C	39	3205
Rocket_000E	39	3205
Rocket_000B	39	3205
Rocket_0009	39	3205
Rocket_000A	39	3205
DCache	39	3596
Rocket_0000	39	3205
Rocket_0001	39	3205
Rocket_0007	39	3205
Rocket_0006	39	3205
Rocket_0005	39	3205
AXI4DeinterLeaver	30	578
RVCExpander	29	193
TLPLIC	23	200
TLB	23	1414
crg	22	7
PTW	22	800
TLB_1	19	1258
AXI4UserYanker_1	19	312
PMPChecker	18	473
top_4_no_SWM	17	89375
TLBroadcast	17	608
top_4_no_SWM_0000	17	89375
top_4_no_SWM_0001	17	89375
top_4_SWM	17	89375
PMPChecker_1	16	411
AXI4UserYanker	16	178
AMOALU	16	249
IBuf	15	118
TLMonitor_35	14	2395
BTB	14	2244
CLINT	13	454
TLToAXI4	12	92
TLMonitor_46	12	2395
TLMonitor_45	11	599
TLMonitor_44	11	599
TLMonitor_43	11	599
TLMonitor_41	11	599
ICache	11	3279

- Distribution of max combinatory depth



Distribution of max comb depth

Depth	#Modules
0-10	230
11-20	24
21-30	6
31-40	17
41-50	1
51-60	2
61-70	1
71-80	2
81-90	0
91-100	0
100+	2

Example 7: Timing Statistics and Bottleneck

zAudit provide two timing-related commands:

- `zAudit -z <zcui.work> timing`
- `zAudit -z <zcui.work> bottleneck`

These commands require the following mandatory settings in UPF when compiling the design.

```
timing_analysis -advanced_command {set zcui_ztime_dump_verdi_db true}
timing_analysis -advanced_command {set zcui_ztime_enable_rtl true}
global_setenv ZEBU_SERIALIZED_ZFPGATIMING_BACK_ANNOTATION 1
```

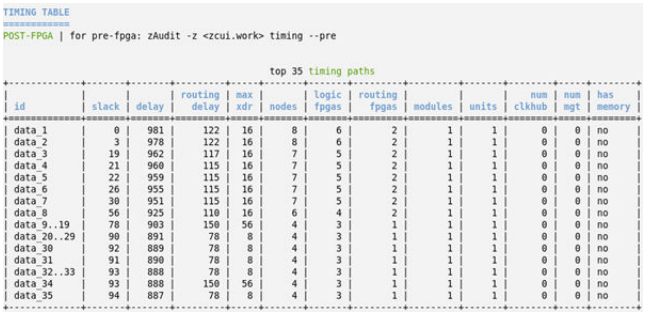
Timing Statistics

To view a key statistics for worst paths based on slack criteria, use the zAudit timing command:

```
zAudit -z <zcui.work> timing
The following table is displayed:
```

Feedback

Figure 34 Timing Statistics



```
TIMING TABLE
============
POST-FPGA | for pre-fpga: zAudit -z <zcui.work> timing --pre

                                                top 35 timing paths
+-------------+-------+-------+---------+-----+-------+-------+---------+---------+-------+-----+-----+--------+
|             |       |       | routing | max |       | logic | routing |         |       | num | num | has    |
| id          | slack | delay |   delay | xdr | nodes | fpgas |   fpgas | modules | units | clkhub | mgt | memory |
+=============+=======+=======+=========+=====+=======+=======+=========+=========+=======+=====+=====+========+
| data_1      |     0 |   981 |     122 |  16 |     8 |     6 |       2 |       1 |     1 |   0 |   0 | no     |
| data_2      |     3 |   978 |     122 |  16 |     8 |     6 |       2 |       1 |     1 |   0 |   0 | no     |
| data_3      |    19 |   962 |     117 |  16 |     7 |     5 |       2 |       1 |     1 |   0 |   0 | no     |
| data_4      |    21 |   960 |     115 |  16 |     7 |     5 |       2 |       1 |     1 |   0 |   0 | no     |
| data_5      |    22 |   959 |     115 |  16 |     7 |     5 |       2 |       1 |     1 |   0 |   0 | no     |
| data_6      |    26 |   955 |     115 |  16 |     7 |     5 |       2 |       1 |     1 |   0 |   0 | no     |
| data_7      |    30 |   951 |     115 |  16 |     7 |     5 |       2 |       1 |     1 |   0 |   0 | no     |
| data_8      |    56 |   925 |     110 |  16 |     6 |     4 |       2 |       1 |     1 |   0 |   0 | no     |
| data_9..19  |    78 |   903 |     150 |  56 |     4 |     3 |       1 |       1 |     1 |   0 |   0 | no     |
| data_20..29 |    90 |   891 |      78 |   8 |     4 |     3 |       1 |       1 |     1 |   0 |   0 | no     |
| data_30     |    92 |   889 |      78 |   8 |     4 |     3 |       1 |       1 |     1 |   0 |   0 | no     |
| data_31     |    91 |   890 |      78 |   8 |     4 |     3 |       1 |       1 |     1 |   0 |   0 | no     |
| data_32..33 |    93 |   888 |      78 |   8 |     4 |     3 |       1 |       1 |     1 |   0 |   0 | no     |
| data_34     |    93 |   888 |     150 |  56 |     4 |     3 |       1 |       1 |     1 |   0 |   0 | no     |
| data_35     |    94 |   887 |      78 |   8 |     4 |     3 |       1 |       1 |     1 |   0 |   0 | no     |
+-------------+-------+-------+---------+-----+-------+-------+---------+---------+-------+-----+-----+--------+
```

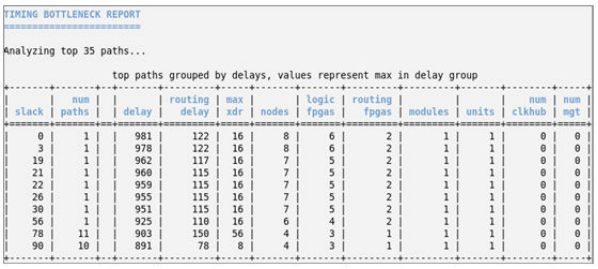
Each **data_*** contains paths with the same slack value.
To display the list of path for every **data_***:
zAudit -z <zcui.work> timing data_*

Bottleneck

To list the most critical paths with details, use bottleneck command:

```
zAudit -z <zcui.work> bottleneck
```

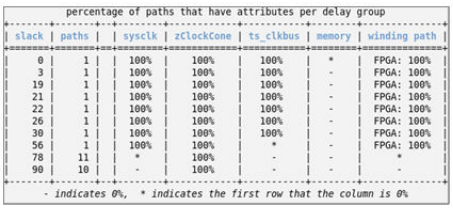
zAudit bottleneck command displays three tables that report top timing paths.



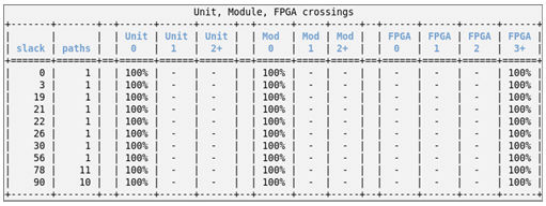
```
TIMING BOTTLENECK REPORT
========================

Analyzing top 35 paths...

                    top paths grouped by delays, values represent max in delay group
+-------+-------+---+-------+---------+-----+-------+-------+---------+---------+-------+-----+-----+
|       |   num |   |       | routing | max |       | logic | routing |         |       | num | num |
| slack | paths |   | delay |   delay | xdr | nodes | fpgas |   fpgas | modules | units | clkhub | mgt |
+=======+=======+===+=======+=========+=====+=======+=======+=========+=========+=======+=====+=====+
|     0 |     1 |   |   981 |     122 |  16 |     8 |     6 |       2 |       1 |     1 |   0 |   0 |
|     3 |     1 |   |   978 |     122 |  16 |     8 |     6 |       2 |       1 |     1 |   0 |   0 |
|    19 |     1 |   |   962 |     117 |  16 |     7 |     5 |       2 |       1 |     1 |   0 |   0 |
|    21 |     1 |   |   960 |     115 |  16 |     7 |     5 |       2 |       1 |     1 |   0 |   0 |
|    22 |     1 |   |   959 |     115 |  16 |     7 |     5 |       2 |       1 |     1 |   0 |   0 |
|    26 |     1 |   |   955 |     115 |  16 |     7 |     5 |       2 |       1 |     1 |   0 |   0 |
|    30 |     1 |   |   951 |     115 |  16 |     7 |     5 |       2 |       1 |     1 |   0 |   0 |
|    56 |     1 |   |   925 |     110 |  16 |     6 |     4 |       2 |       1 |     1 |   0 |   0 |
|    78 |    11 |   |   903 |     150 |  56 |     4 |     3 |       1 |       1 |     1 |   0 |   0 |
|    90 |    10 |   |   891 |      78 |   8 |     4 |     3 |       1 |       1 |     1 |   0 |   0 |
+-------+-------+---+-------+---------+-----+-------+-------+---------+---------+-------+-----+-----+
```



```
                percentage of paths that have attributes per delay group
+-------+-------+---+--------+-------------+-----------+--------+--------------+
| slack | paths |   | sysclk | zClockCone  | ts_clkbus | memory | winding path |
+=======+=======+===+========+=============+===========+========+==============+
|     0 |     1 |   |  100%  |    100%     |   100%    |   *    | FPGA: 100%   |
|     3 |     1 |   |  100%  |    100%     |   100%    |   -    | FPGA: 100%   |
|    19 |     1 |   |  100%  |    100%     |   100%    |   -    | FPGA: 100%   |
|    21 |     1 |   |  100%  |    100%     |   100%    |   -    | FPGA: 100%   |
|    22 |     1 |   |  100%  |    100%     |   100%    |   -    | FPGA: 100%   |
|    26 |     1 |   |  100%  |    100%     |   100%    |   -    | FPGA: 100%   |
|    30 |     1 |   |  100%  |    100%     |   100%    |   -    | FPGA: 100%   |
|    56 |     1 |   |  100%  |    100%     |     *     |   -    | FPGA: 100%   |
|    78 |    11 |   |   *    |    100%     |     -     |   -    |      *       |
|    90 |    10 |   |   -    |    100%     |     -     |   -    |      -       |
+-------+-------+---+--------+-------------+-----------+--------+--------------+
      - indicates 0%,  * indicates the first row that the column is 0%
```



```
                                Unit, Module, FPGA crossings
+-------+-------+---+------+------+------+---+------+------+------+---+------+------+------+------+
|       |       |   | Unit | Unit | Unit |   | Mod  | Mod  | Mod  |   | FPGA | FPGA | FPGA | FPGA |
| slack | paths |   |  0   |  1   |  2+  |   |  0   |  1   |  2+  |   |  0   |  1   |  2   |  3+  |
+=======+=======+===+======+======+======+===+======+======+======+===+======+======+======+======+
|     0 |     1 |   | 100% |  -   |  -   |   | 100% |  -   |  -   |   |  -   |  -   |  -   | 100% |
|     3 |     1 |   | 100% |  -   |  -   |   | 100% |  -   |  -   |   |  -   |  -   |  -   | 100% |
|    19 |     1 |   | 100% |  -   |  -   |   | 100% |  -   |  -   |   |  -   |  -   |  -   | 100% |
|    21 |     1 |   | 100% |  -   |  -   |   | 100% |  -   |  -   |   |  -   |  -   |  -   | 100% |
|    22 |     1 |   | 100% |  -   |  -   |   | 100% |  -   |  -   |   |  -   |  -   |  -   | 100% |
|    26 |     1 |   | 100% |  -   |  -   |   | 100% |  -   |  -   |   |  -   |  -   |  -   | 100% |
|    30 |     1 |   | 100% |  -   |  -   |   | 100% |  -   |  -   |   |  -   |  -   |  -   | 100% |
|    56 |     1 |   | 100% |  -   |  -   |   | 100% |  -   |  -   |   |  -   |  -   |  -   | 100% |
|    78 |    11 |   | 100% |  -   |  -   |   | 100% |  -   |  -   |   |  -   |  -   |  -   | 100% |
|    90 |    10 |   | 100% |  -   |  -   |   | 100% |  -   |  -   |   |  -   |  -   |  -   | 100% |
+-------+-------+---+------+------+------+---+------+------+------+---+------+------+------+------+
```

The following details are displayed for the most critical path:

Feedback

```
current post-fpga frequency: 1019 kHz (981ns)
--> pre-post frequency delta is 1848 kHz -> 1019 kHz (-44%)

to increase frequency by 0% to 1022 kHz (978ns), 1 paths must be improved

the first longest path (highest number of nodes) in the 0 slack group:

+--------------+-----------------------------------------------------------------------------------------+
| field        | value                                                                                   |
+==============+=========================================================================================+
| id           | data_1                                                                                  |
| slack        | 0                                                                                       |
| delay        | 981                                                                                     |
| routing_delay | 122                                                                                    |
| clock_from   | kpc_tb_top.clock_ClockPort20.clockdelayport_core.clockdelayport_state_machine.clock (posedge) |
| clock_to     | DRIVERCLOCK (system)                                                                    |
| nodes        | 8                                                                                       |
| modules      | 1                                                                                       |
| units        | 1                                                                                       |
| routing_fpgas | 2                                                                                      |
| logic_fpgas  | 6                                                                                       |
| has_memory   | False                                                                                   |
| num_mgt      | 0                                                                                       |
| num_clkhub   | 0                                                                                       |
| has_ts_clkbus | True                                                                                   |
| max_xdr      | 16                                                                                      |
| crosses_fpga | 7                                                                                       |
| crosses_mod  | 0                                                                                       |
| crosses_unit | 0                                                                                       |
| winding_path | fpga                                                                                    |
+--------------+-----------------------------------------------------------------------------------------+

Note: The full path can be displayed with: zAudit -z <zcui.work> timing data_1
```

The following information is displayed for recommendations to improve timing:

```
Recommendations To Improve Timing:

================================================================================
zTime

ISSUE:
  path has zClockCone in data path
OPTIONS:
  [OK] fetch mode: yes

ISSUE:
  path contains ts_clkbus and routing fpgas which can suggest that:
  - might contain more than just clockbus, typically due to clock stopping logic such as transactors or triggers
  - might be high CUT across these FPGAs
```

Example 8: View Diagnostic Statistics

To view diagnostic statistics of FPGAs, use the following command:

```
zAudit -z <zcui.work> diagnostic
```

```
>> DIAGNOSTIC REPORT <<

================================================================================
zAudit

ISSUE:
  zebu.work/tools/zTime_Verdi_db not found, some timing information cannot be displayed
RECOMMENDATION:
  Add UTF commands:
   timing_analysis -advanced_command {set zcui_ztime_dump_verdi_db true}
   timing_analysis -advanced_command {set zcui_ztime_enable_rtl true}
   global_setenv ZEBU_SERIALIZED_ZFPGATIMING_BACK_ANNOTATION 1


================================================================================
Vivado

ISSUE:
  CONG-LUT6-UTIL : has high LUT6 Utilization which can increase P&R time and chance of P&R failure
  FPGAs: U0_M0_F02 U0_M0_F04 U0_M0_F05
RECOMMENDATION:
  Target 20% LUT6 fill rate at synthesis and Back-End
   synthesis -advanced_command {Compile:MinLut6ToRemap = 10000}
   synthesis -advanced_command {Compile:TargetLUT6Ratio = 0.2}
   ztopbuild -advanced_command {cluster set -max_fill_lut6 20}
  QIWC: FPGAs having high LUT6 fill rate which can lead to chance of FPGA failure or long compile time
   ztopbuild -advanced_command {cluster set -max_fill_lut6 20}

ISSUE:
  Route 35-3
  FPGAs: U0_M0_F00 U0_M0_F01 U0_M0_F02 U0_M0_F03 U0_M0_F04 U0_M0_F05
RECOMMENDATION:
  ROUTER ==> Router is having difficulty in routing, although there is no single match for the complexity predictor. It is possible that timing congestion could be high

ISSUE:
  SSI : FPGA is having high SLL demand
  FPGAs: U0_M0_F05
RECOMMENDATION:
  QIWC: FPGAs having high qiwc fill which can lead to chance of FPGA failure or long compile time
   ztopbuild -advanced_command {cluster set -max_fill_qiwc 40}
```

Visualizing Compile-Time Data

zCTGram integration with zAudit allows you to visualize Vivado compile-time data. Analyzing key statistics such as, filling rates, backend solution quality, partitioning,

routing, and synthesis quality in a text file is highly complex and time consuming process. zCTgram provides an effective approach to visualize such key parameters.

Enabling zCTgram

To enable zCTgram, use the following command:

```
zAudit -z zcui.work fpga -plot abc.html
```

`abc.html` is generated in the current working directory. You can use any Internet browser to open `abc.html file`. For example, *firefox abc.html*. The HTML displays the following information:

Compilation Details

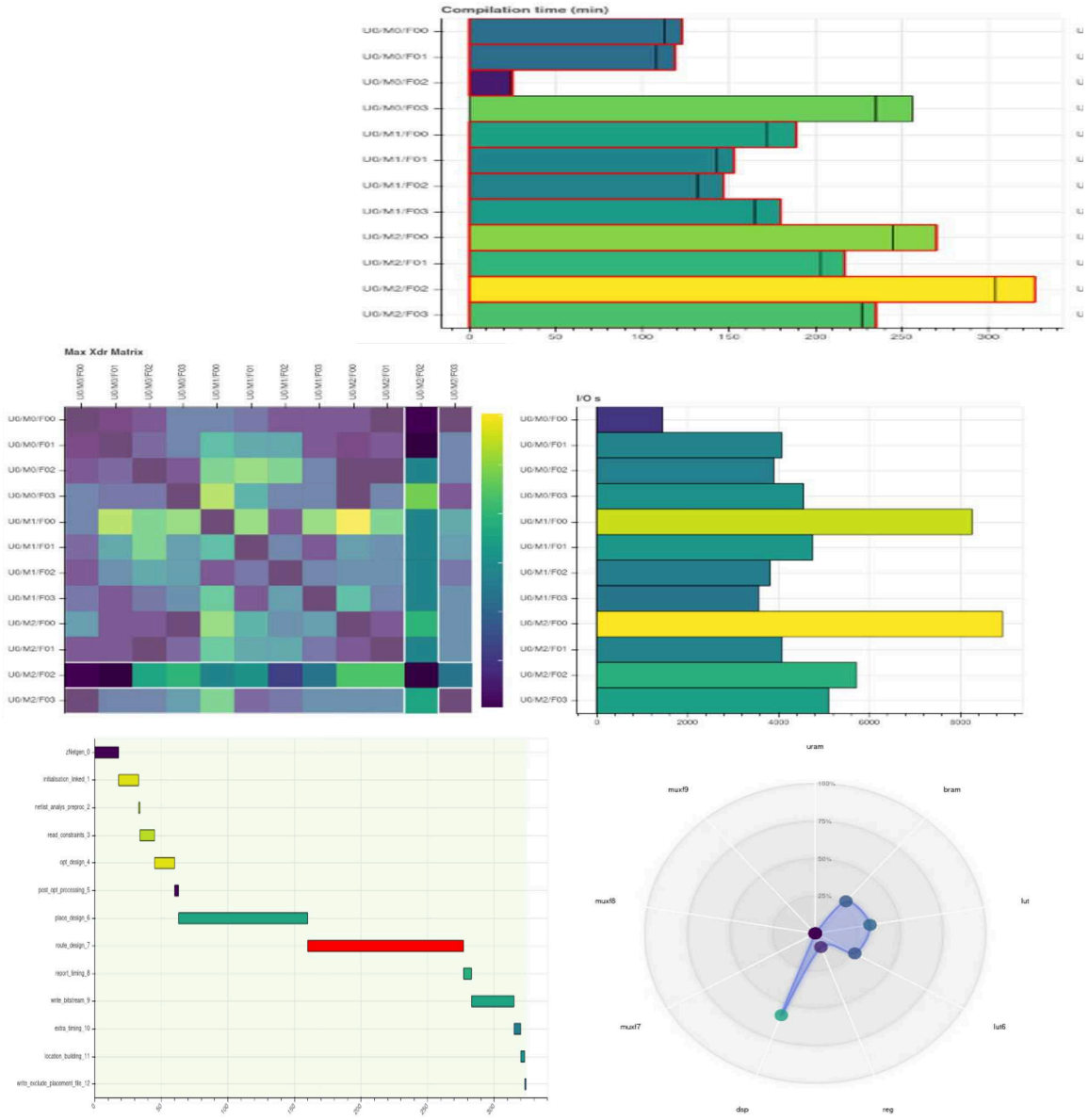
- Overall Compilation time for each FPGA is shown in the Gantt chart.
  - Vivado execution time
  - Total time taken
  - CPU load is displayed based on the grid status
  - PARFF triggered
- Compilation time of each Vivado phase is shown on clicking each FPGA.
  - The host name where the job is launched is used to identify the health of the machine.
  - The host load information is useful to further analyze the compile time.
- Max XDR Matrix for the design between the FPGAs in the module.
  - Color gradient shows the increasing severity from Violet to Yellow
- IO distribution for each FPGA

Limitation: HOPS are not supported in the zAudit tool.

- Resource utilization:
  - The web diagram shows the percentage of resource used by the FPGA, such as MUX*, URAM, BRAM, LUT, DSP, and so on.
  - This provides details about complications in the design compilation which leads to increase in the compile time.

Feedback

Figure 35 Compilation Details Tab



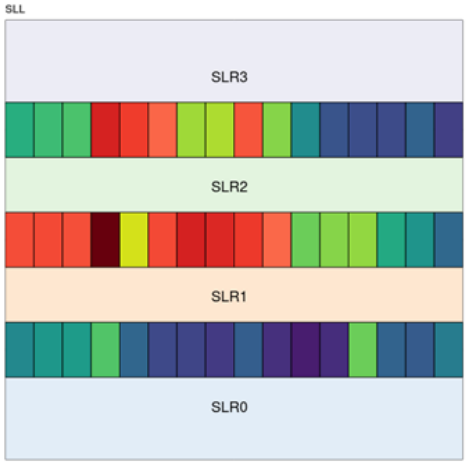
SLL

The SLL tab shows the percentage of congestion between the SLRs.

Feedback

- Red bars towards the left indicate high congestion due to URAM/BRAM.
- Red bars towards the right indicate the routing congestion.

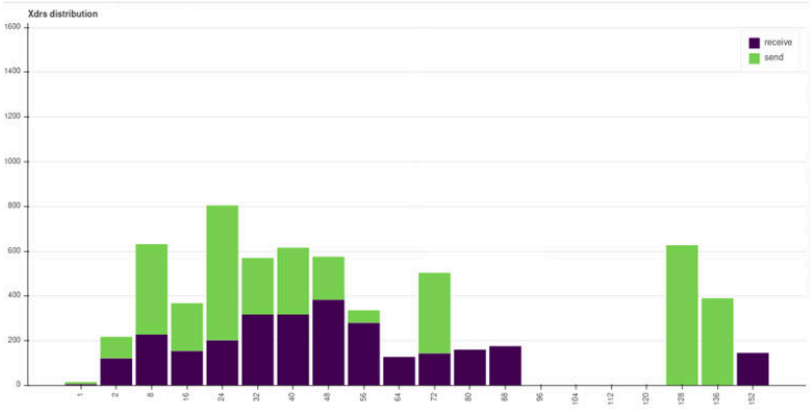
Figure 36 SLL Tab



Xdrs Distribution

Xdrs distribution is used to visualize the XDRs used in the design and also the distribution of the nets.

Figure 37 Xdrs Distribution Tab



Handling Misfiring SVAs with Latches and Memories in Fanin

If a latch is a support register of an SVA, it is possible that a clock cone glitch propagates to the SVA causing it to misfire. This is not true when using Fetch-Mode clock processing, but happens with FGS. You can avoid this behavior in FGS with this feature to resolve the issue due to an extra cycle delay at the output of memory.

Feedback

Use Model

To prevent SVA misfiring due to latch transparency and glitches from clock cone propagation, use the following switches in FGS mode:

- VCS switch:

```
-Xashwani=0x20
```

- UTF command:

```
ztopbuild -advanced_command { clock_handling
 filter_glitches_synchronous -FGS_fetch_mode=no
 -report_unsafe_latch_array_to_sva=warning
 -FGS_filter_latch_transparency_to_sva=hybridMode}

ztopbuild -advanced_command "enable replicate_clock_tree"
```

This feature provides a diagnostic tool to flag SVA with memories in *fanin*. These can also trigger incorrect SVA misfiring, which are printed in the `zTopBuild.log` file.

Note:

This flow can lead to increase in design size.

Replicating Multiport Memories in zMem

The default sysClk frequency used in ZeBu Server 4 is 50 MHz. Higher sysClk frequency can improve memory performance, but may make it harder for Vivado to achieve timing closure. To help Vivado converge faster, high-fanout nets in a multiport zMem can be replicated to reduce their fanout count.

Note:

Only FSM control signals with source and destination nets within the same FSM wrapper modules are supported.

Usage

To control high fanout replication settings in zMem, specify the following UTF command:

```
memory_preferences -advanced_command { config_zmem
 -high_fanout_replication false|true|clk_50|clk_100|clk_200 }
```

To change the fanout limit, specify the following UTF command:

```
memory_preferences -advanced_command { config_zmem
 -fanout_replication_limit <n> }
```

Default is 64.

For more details, run the following help command:

```
zMem -help config_zmem
```

Reporting High Fanout Nets

zMem writes the fanout count of nets in the following format:

```
##########################################################################
#######
# Fanout Report for memory: dut_ZMEM_memA
##########################################################################
#######


#  GENERAL STATISTICS:
 ########################################################
# Number of Wires driving pure clock fanout:
      3
# Number of Wires driving pure data  fanout:
   7403
# Number of Wires driving both clock and data fanout:
      0
# Number of Wires without any load:
      1

############################################################  FANOUT
 HISTOGRAM:
# Number of wires whose fanout is in [     1,       1] :
 6333
# Number of wires whose fanout is in [     2,       3] :
   9
# Number of wires whose fanout is in [     4,       7] :
 1054
# Number of wires whose fanout is in [     8,      15] :
   0
# Number of wires whose fanout is in [    16,      31] :
   0
# Number of wires whose fanout is in [    32,      63] :
   0
# Number of wires whose fanout is in [    64,     127] :
   0
# Number of wires whose fanout is in [   128,     255] :
   0
# Number of wires whose fanout is in [   256,     511] :
   0
# Number of wires whose fanout is in [   512,    1023] :
   0
# Number of wires whose fanout is in [  1024,    2047] :
   4
# Number of wires whose fanout is in [  2048,    4095] :
   0
```

```
# Number of wires whose fanout is in [  4096,    8191] :
   5
# Number of wires whose fanout is in [  8192,   16383] :
   0
# Number of wires whose fanout is in [ 16384,   32767] :
   1
# Number of wires whose fanout is in [ 32768,   65535] :
   0
# Number of wires whose fanout is in [ 65536,  131071] :
   0
# Number of wires whose fanout is in [131072,  262143] :
   0
# Number of wires whose fanout is in [262144,  524287] :
   0
# Number of wires whose fanout is in [524288, 1048575] :
   0

##########################################################  FANOUT TOP
 SCORES:
# Displaying all the wires whose fanout is greater than 128.
#   21000  #  dut_ZMEM_memA.memory_base.dut_ZMEM_memA_base_add_gnd
    #
#    5250  #  dut_ZMEM_memA.we0Multiplex_out
    #
#    5250  #  dut_ZMEM_memA.memory_base.wire_sysFreq
    #
#    4200  #  dut_ZMEM_memA.addr0Multiplex_out0
    #
#    4200  #  dut_ZMEM_memA.addr0Multiplex_out1
    #
#    4200  #  dut_ZMEM_memA.addr0Multiplex_out2
    #
#    1063  #  dut_ZMEM_memA.wire_sysFreq
    #
#    1055  #  dut_ZMEM_memA.w0clk_reg_in
    #
#    1053  #  dut_ZMEM_memA.And_w0_FSM0_opt
    #
#    1050  #
 dut_ZMEM_memA.zMem_multiport_TO_clk_100_And_FSM1_FD_FIFOr1_FD  #
```

The report log is available at the following location:

```
zcui.work/design/synth_Default_RTL_Group/zmem/Memory_<module>_zMem/
<module>_ZMEM_<array>_report.log
```

Limitations

Replication of multiport memories with cross hierarchies are not supported.

Feedback

Verilog force/release

Verilog force release is supported by default on ZeBu. The instrumentations are handled at synthesis level to be more accurate according to LRM.

To disable Verilog force release processing, use the following UTF command:

```
verilog_force_release -enable false
```

Reporting Verilog force/release Information

Verilog force release information is captured in different log files.

The default information is available in `vcs_splitter_VCS_Task_Builder.log` at `zcui.work/zCui/log`. It contains the basic information on the following:

- The number of Verilog force/release statements
- The number of XMRs

To capture the Verilog force/release statements, add the following UTF command in your UTF file:

```
design -report force
```

The report is generated at `zcui.work/vcs_splitter/simon.out/force_debug.log` and it captures the following information module by module:

- All force/release statements in the design
- Source locations of all force/release statements
- The XMR string, its target module, and its target signal (with source location), if the forced/released signal is an XMR.

Example

```
1  module top();
2     wire x;
3     wire y;
4     test u1();
5     sub s1(.in(u1.temp2));
6  endmodule
7
8  module test (in, out);
9     input in;
10    output out;
11    reg temp, temp2;
12    assign out = top.x;
13    assign temp = top.x;
14    initial begin
15        force top.y = top.s1.temp;
16        force temp = in;
17    end
18 endmodule
19
20 module sub(in, out);
21    input in;
22    reg temp;
23    output out;
24    assign out = in;
25 endmodule
26
```

Feedback

The captured information is as follows:

- In `zcui.work/zCui/log/vcs_splitter_VCS_Task_Builder.log`

```
[Native Force]: Native force detects two Verilog force/release in the
 design after analyzing.
[Native Force]: Native force creates two reader/writer connections
 after processing.
```

- In `zcui.work/vcs_splitter/simon.out/force_debug.log`

```
1  ######################################First Part#########################################
2  MODULE: test
3      STATEMENT [ID:1]: force top.y = top.s1.temp; (top.sv: 15)
4          XMR: top.y
5              TARGET SCOPE:   top
6              TARGET SIGNAL:  wire y; (top.sv: 3)
7          XMR: top.s1.temp
8              TARGET SCOPE:   sub
9              TARGET SIGNAL:  reg temp; (top.sv: 22)
10     STATEMENT [ID:2]: force temp = in; (top.sv: 16)
11
```

Limitations

The following limitations are applicable:

- Force/release is not supported on VHDL signals.
- Force/release is not supported on a function/task local signal.
- Force handling might not follow simulation behavior in the following cases:
  - Multi force release scenarios: If multiple forces and releases are applied on a signal from different processes and different modules.
  - Signals that have other drivers as follows:
    - resettable flop
    - latch/memlatch with enable
    - other combinational scenario where sensitivity plays a significant role for triggering always block in RTL
  - Avoid applying force on signals that are read after write in the same process. For example, avoid applying force on **b** and **c** as follows. However, for read-after-write in same process/always-block is supported for simple scenarios.

```
always @(*)
begin
b = a;
c = b;
```

```
d = c;
end
```

- Forced signal in *always@** block: When forcing a signal that is written by an *always@** process, in simulation, the signal maintains its forced value until another write occurs. In the implementation, force/release happens immediately, even if there is no change on signals.
- Force on SV Real datatypes is not supported
- Force on SV nettypes is not supported
- If force statements are declared in process where there is no sensitivity list, simulation ignores those process. However, it is synthesized in emulation.

  Example:

```
always(*)
force a = 1'b1;
```

Enhanced Multi-Gigabit Transceiver Support

The support for fast Multi-Gigabit Transceiver (MGT) links is enabled by default. This feature reduces the latency of the cable links between each ZeBu units, and contribute to increase the overall **zTime** frequencies.

Note:

This feature only applies to multiunit systems of ZeBu Server 4 and ZeBu Server 5.

To restore the previous behavior, see the *ZeBu Release Notes* for your hardware.

Multi-Bin Based FPGA Complexity

Multi-Binning feature is used for predicting an FPGA as complex with higher accuracy. This feature allows you to declare the required threshold. If an FPGA is declared as complex, then it is submitted to the ZebuComplexFpga queue. If the queue is full, then it is submitted to ZebuFpga queue. The P&R task launch in each queue happens in the order of the most recent complex submitted.

Multi-Binning feature is enabled by default with the following settings:

- Predicting the FPGA as complex or non-complex with a user-defined threshold (default is 6 hours for ZeBu Server 5 and 4 hours for ZeBu Server 4).
- MPF is not enabled by default. You can use single host or multi host MPF.

For enabling MPF, see section Multi-Process Flow.

- You can enable single or multi host MPF to submit FPGAs having threshold value higher than the user-defined threshold.
- Job submission to a particular queue for FPGA build happens in the order of complexity (more complex FPGA to run first and less complex to be scheduled later).

Enabling Multi-Binning

The Multi-Binning predictor is enabled by default for ZeBu Server 4 and ZeBu Server 5. You can configure the default threshold value by using the following UTF command:

```
fpga -user_complexity_threshold <0-10>
```

The predictor is run post zPar stage before the creation of the backend compile jobs. A complexity score is written in the file `<fpga_dir>/complexity_score.info`.

To disable the predictor, use the following UTF command:

```
fpga -user_complexity_threshold -1
```

Performance Driven Multi-die Multiplexing Support

Performance Driven Multi-die Multiplexing (PDM) is supported to improve the stability and the routability of FPGAs. PDM support is added in the compilation flow between **zPar** and FPGA compilation.

- *PdmDap* (1 per FPGA): Identifies prior FPGAs with high Super Long Line (SLL) demands
- *PdmTiming*: Identifies the multiplexing rates of the low critical nets between dies of the FPGAs

  Note:

  The use of zTime estimation using Vivado arc timing information is mandatory.

This section describes the following subtopics:

- Enabling PDM Support
- zCui Flow

Feedback

Enabling PDM Support

To enable the PDM support, use the following UTF command:

```
fpga -inter_die_tdm true [-inter_die_tdm_params {strategy=PDM_DIRECT |
 PDM_PARFF}]
```

where,

- `PDM_DIRECT`: Enables PDM for original compilation
- `PDM_PARFF`: Enables PDM only when PARFF starts (with dedicated PARFF (PDM-aware) strategies)

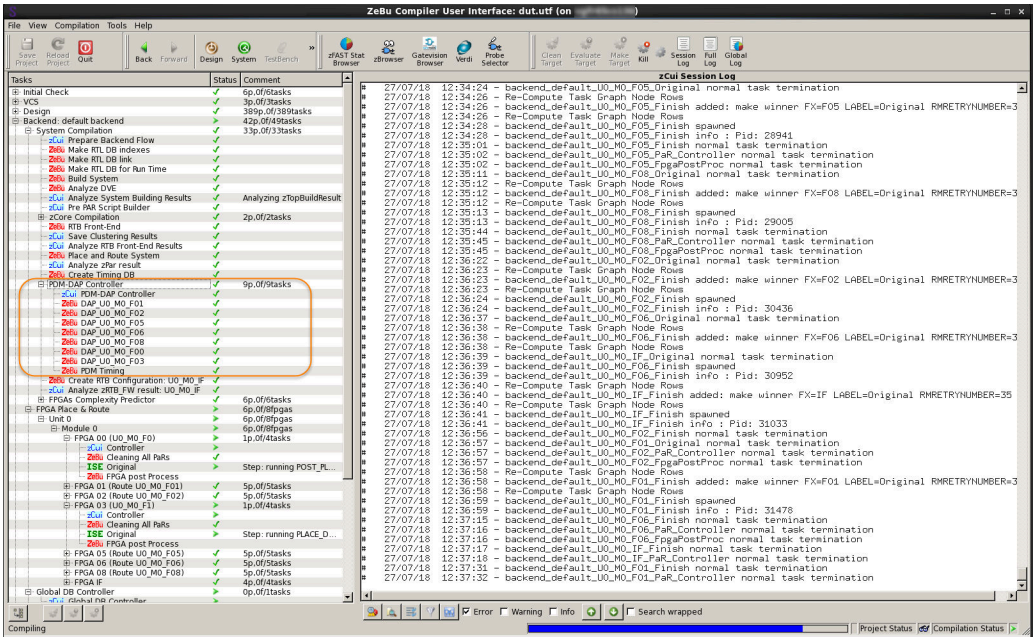
  Note:

  By default, the PDM_DIRECT mode is set when PDM support is enabled.

zCui Flow

The following figure shows when the PDM flow is enabled. Die Aware Partitioning (DAP) and PDM are launched by **zCui** before the FPGAs.

Figure 38 PDM Flow in zCui



Memory Optimization

The following design memory optimizations are available in ZeBu:

- Reshaping of User Memories in Backend
- Decomposing Large ZRM Memories to Fit the Hardware
- Memory Coalescing

Reshaping of User Memories in Backend

The RTL memories are reshaped in the backend by leveraging logic optimizations that identify certain bits of the address as a constant. Any address bit that is tied to a constant can reduce the memory form factor by half. This is called reshaping of user memories.

This could result in substantial saving of capacity by means of memory primitives, which could ease partitioning, and avoid issues such as congestion and overflow further down the line.

Enable Reshaping of User Memories

Use the following `ztopbuild` UTF command to enable reshaping of user memories:

```
ztopbuild -advanced_command {memory_opt port_opt -enable_reshaping
 [-exclude_instances <EXCLUDE_RTL_INSTANCES>] [-exclude_modules
 <EXCLUDE_RTL_MODULES>] [-include_instances <INCLUDE_RTL_INSTANCES>]
 [-include_modules <INCLUDE_RTL_MODULES>]}
```

Note:

- Memory can be reshaped only if the same constant value is present on an address bit for all ports of that memory.
- The following options accept RTL paths, and not EDIF paths.
  - `-exclude_instances`
  - `-include_instances`
  - `-exclude_modules`
  - `-include_modules`

Reporting

The following examples depict the **Memory Optimization** section of the `zTopBuild_report.log`.

Example 1

It contains the details of all memory instances that have undergone reshaping, without filtering, or excluding any modules.

```
Instance name : dve.i_dut.i_gpu.i_geometry.i_tesselation.i_mst_tr_buf.BUF_RDATA_R
Original memory name : mst_tr_buf_SNPS_ID_92124_ZMEM_BUF_RDATA_R
Original memory frequency : 7692 kHz
Optimized memory name : ms_tr_buf_SNPS_ID_92124_ZMEM_BUF_RDATA_R__portopt__4_replace
Optimized memory frequency : 7692 kHz
Memory type : ZMEM
Ports optimized away are as follows :
For all ports, address bit : 4 set to : VCC

Total number of memories considered : 826
Total number of memories affected by unconnected port optimization : 1
Total number of memories affected by address reshaping : 10
Total bit volume of memories processed : 34386886884
Total bit volume reduced by reshaping : Absolute (233472), %age reduction : (0.000678957%)
Number of instances with address space reduced by a factor of 2 is : 10
Unconnected/Inactive Port Optimization: End
MemoryOpt::preProcess Nothing to do
MemoryOpt::postProcess Nothing to do
```

Example 2

The following image contains the details of all memory instances that have undergone reshaping, when excluding two memories with a user-pattern using the command: `(memory_opt port_opt -const_prop -exclude_modules {mst_tr_buf.BUF_RDATA_R})`:

```
Total number of memories considered : 824
Total number of memories affected by unconnected port optimization : 1
Total number of memories affected by address reshaping : 8
Total bit volume of memories processed : 34386878692
Total bit volume reduced by reshaping : Absolute (229376), %age reduction : (0.000667045%)
Number of instances with address space reduced by a factor of 2 is : 8
Unconnected/Inactive Port Optimization: End
MemoryOpt::preProcess Nothing to do
MemoryOpt::postProcess Nothing to do
```

Limitations

Memories created through the sparse memory command are not subjected to port optimization or reshaping.

Decomposing Large ZRM Memories to Fit the Hardware

Along with the on-chip memories (RAMLUT, BRAM and URAM), there are on-board memory resources (called ZRM memory banks) which are meant for large sized memories (exceeding 500 kBytes), using DDR memory banks.

Feedback

The largest ZRM memory bank available, or the total count of the ZRM memories that can be mapped to a bank, is limited by the hardware configuration of ZeBu. For example, the largest available ZRM memory bank on ZS4 has a capacity of 8 GB.

When the design includes user memories having a size greater than the largest ZRM bank available, it results in compilation errors.

This feature splits such large ZRM memories, which exceed the size of the largest available ZRM bank, into legally conforming ZRM memories, based on the ZeBu configuration.

Enable Decomposing Large ZRM Memories

Use the following `ztopbuild` UTF command to enable decomposing large ZRM memories:

```
ztopbuild -advanced_command {memory_opt decompose_large_zrm}
```

Reporting

The **Memory Optimization** section of `zTopBuild_report.log` contains the details of decomposed large ZRM memories, if any.

Limitations

Decomposition of ZRM memories is not supported for the presence of tristated outputs (OE pins).

Memory Coalescing

Memory resource requirements can be reduced by combining multiple design memories into a single physical implementation. This technique of combining memories to optimize capacity utilization is called memory coalescing. However, memory coalescing increases the degree of multi-porting of memories, which reduces their operating frequency. If coalesced memories lie on the **zTime** critical path, it may have an adverse effect on the emulation frequency.

Enable Coalescing BRAM Memories

To enable memory coalescing, use the following `zTopBuild` UTF command:

```
ztopbuild -advanced_command {memory_opt coalesce
 [-exclude_instances <EXCLUDE_RTL_INSTANCES>][-exclude_modules
 <EXCLUDE_RTL_MODULES>][-include_instances
 <INCLUDE_RTL_INSTANCES>][-include_modules
 <INCLUDE_RTL_MODULES>][-min_freq][-include_ramluts][-set_sys_freq
 <SET_SYS_FREQ>][-max_util_percentage <utilization percentage>]}
```

Feedback

- `-exclude_modules`: Specifies the list of the RTL modules that are to be exempted from coalescing. Note that all instances of the specified modules are exempted from coalescing.
- `-exclude_instances`: Specifies the list of RTL instances that are to be exempted from coalescing.
- `-include_instances`: Specifies the list of RTL instances to be considered for coalescing. All other memories are exempted from coalescing.
- `-include_modules`: Specifies the list of RTL modules to be considered for coalescing. All other memories are exempted from coalescing.
- `-min_freq`: Specifies a minimum operating frequency below which a coalesced memory is not generated. This parameter decides the degree of multi-porting that the tool is allowed to apply on the coalesced memories, and consequently determines the number of memories that can be packed together. If this is not specified, the default value of 5000 kHz is considered for the minimum operating frequency.
- `-include_ramluts`: The Boolean switch to permit LUTRAM memories to be considered for coalescing. If set to `true`, several LUTRAM-based memories are combined together into a single BRAM.
- `-set_sys_freq`: Specifies the operating system frequency for the coalesced memories. By default, the coalescing operation preserves the system frequency of the memories getting coalesced. `<SET_SYS_FREQ>` is an `enum` type with one of the following values: `clk_25`, `clk_50`, `clk_100`, `clk_200` or `auto`. The default is `auto`.
- `-max_util_percentage`: Allows the user to specify a threshold primitive utilization percentage. Any memory considered for coalescing must have a lower utilization percentage than this threshold value. The default value is 5%.

Note:
The following options accept RTL paths, and not EDIF paths.

- `-exclude_instances`
- `-include_instances`
- `-exclude_modules`
- `-include_modules`

Reporting

When enabled, the Memory Optimization section in the **zTopBuild_report.log** report consists of a sub-section with details about coalesced memories. This sub-section describes the following information:

- Sets of user memories grouped together into a new coalesced memory with information such as hierarchical paths, dimensions, ports, new form factor of coalesced memory, and operating frequency.
- Aggregate statistics that estimate the total projected resource savings gained from all the coalescing operations applied to the design.

4

Clock Modeling in ZeBu

In ZeBu, clocks are instantiated in the RTL source files, either through replacement of the simulation clock generator with a dedicated macro or by re-using the delay coding of the simulation environment (#delay feature).

Regardless of the method in which user clocks are instantiated, the ZeBu compiler automatically handles its internal clocks processing for a predictable behavior during emulation and provides an estimation of the maximum driver clock frequency that can be achieved. The corresponding ZeBu internal clocks parameters are automatically considered for emulation runtime. User clocks frequencies can be configured to operate at any user specified frequency.

During emulation, depending on the type of clocks instantiated in RTL, you can use either of the clock controls: cycle-based or time-based control.

This section describes the following sub-topics

- Types of Clock Supporting Emulation
- Cycle-Based Clock Control
- Time-Based Clock Control
- C_COSIM and Vertical Solution Transactors
- zDPI and ZEMI-3

Feedback

Types of Clock Supporting Emulation

There are two types of clock controls instantiated in RTL design that support emulation. These are: cycle-based control or time-based control. The main points of differences between the two types are as follows:

Table 8 Types of Clocks

Cycle-Based Control Clocks	Time-Based Control Clocks
Design clocks are generated by instantiating the zceiClockPort macro in RTL	Design clocks are generated by either of the following ways:Using #design to implement delays in the RTL code when simulating the design or by instantiating the clockDelayPort macro in RTL
Maximum number of primary clocks that can be used is 16	There is no limitation on the number of primary clocks that can be used
Clocks can be configured before starting emulation as part of the runtime settings in the designFeatures file	Clocks additionally support on-the-fly control emulation. Can also be configured using Synopsys Design Constraints (SDC) file as part of the design intent
Control runtime progress in term of cycles	Control runtime progress with time (100ns)
-	Provides timestamps for waveforms, DPI calls and SVAs. It also supports the $time system task, which is required in the clock generation mode when targeting simulation acceleration or power estimation features

Cycle-Based Clock Control

To generate controlled clocks for ZeBu, you should instantiate the `zceiClockPort` clock generation macro in your RTL for each primary clock, connecting the `cclock` output to the design clock signals.

Figure 39 Instantiate the zceiClockPort clock generation macro

Feedback

Note that additional `creset` and `cresetn` ports should be left unconnected. It is no longer recommended as a reset control for the design. Instead, it is recommended that reset input of the design should be manually controlled.

You can instantiate the `zceiClockPort` macro at any hierarchical level in the design, and there are no specific constraints on the instance name.

```
zceiClockPort <instance name> (.cclock ( <design_clock>));
```

When using cycle-based control, you can instantiate one clock generator for each design clock. Also, you can instantiate up to 16 primary clocks in the RTL design.

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 4, multiple clock groups can be used. However, there is still only one time group.
- 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
```

Time-Based Clock Control

Time-based control clocks support emulation in two ways: by using simulation-like coding with delays in the initial and always blocks in the RTL design or by using the `ClockDelayPort` macro.

Feedback

This section describes the following topics:

- ClockDelayPort
- RTL Delays
- Waveform Dumping for the Clock Delay Feature
- Reducing Disk Usage
- Timescale and Precision
- Compilation Results
- Timestamp Clock

ClockDelayPort

This section describes time-based clock control using `ClockDelayPort` in the following topics:

- Using the clockDelayPort Macro
- Optimizing clockDelayPorts
- Configuring clockDelayPort Through designFeatures
- Using ZeBu Clock Delay Primitive
- Tolerance Support
- Limitations With clockDelayPort

Using the clockDelayPort Macro

`clockDelayPort` is simpler mechanism to generate clocks and reset using delays. It can be used in replacement of a previously used clock generator.

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 in the UTF file without any option to enable the clock delay feature:

```
clock_delay
```

This 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.

This macro generates two signals, `clock` and `reset`. In addition, it provides initialization values for the following:

- Low level duration and high level duration
- Phase of the clock signal
- Active level
- Duration of the reset signal.

Some defaults are set for these parameters to generate a clock with 50% duty cycle starting at a high level, and an active low reset, equivalent to `ClockDelayPort #(1,1,0,0,1)`.

Initial values for `clock` signal can be modified through the `designFeatures` file before runtime or by using runtime control API. Active level and duration for `reset` signal can be modified using the C++ API. The following table provides a list of equivalent code to when using this macro.

Table 9 Using clockDelayPort Macro

Existing Method	Equivalent ClockDelayPort Macro

`initial begin`
`  reset <= 1'b0;`
`#2 reset <= 1'b1`
`end`
`always begin`
`  #5 clk1 <=1'b1`
`  #9 clk1 <=1'b0`
`end`
	`clockDelayPort #(5,9,0,0,2) def_clk1(clk1, reset)`

`always begin`
`  #7 clk2 <=1'b1`
`  #11 clk2 <=1'b0`
`end`
	`clockDelayPort #(7,11,0) def_clk2(clk2)`

Note:
The `clockDelayPort` macro ignores the timescale directive in the Verilog file and the default timescale of `clockDelayPort` is "1ps".

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

Feedback

C++ API	C API	Description
`ClockDelayPort::doNewReset`	`ZEBU_ClockDelayPort_doNewReset`	Takes the hierarchical name of the *clockDelayPort* 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.
`ClockDelayPort::reconfigureClockShape`	`ZEBU_ClockDelayPort_reconfigureClockShape`	Takes the hierarchical name of the *clockDelayPort* 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.
`ClockDelayPort::disableClock`	`ZEBU_ClockDelayPort_disableClock`	Takes the hierarchical name of the *clockDelayPort* 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.
`ClockDelayPort::Configuration getConfig`	`ZEBU_ClockDelayPort_Configuration getConfig();`	Returns the current delay settings of particular clock delay port from hardware.

Optimizing clockDelayPorts

The `ClockDelayPort` replication LITE mode allows clock replication without the overhead of `zceiMessagePort` increase. Previously, for designs with large number of FPGAs and `clockDelayPorts`, replication of `clockDelayPorts` in the full mode increased. The number of required transactor ports exceeding the limit that a single PCIe card can handle. Therefore, it leads to the following runtime error:

```
ZeBu : zServer : ERROR : DEV0003E : initUpPages failed. Unable to program
 PCIE uptable pages
```

To enable the `clockDelayPort` replication LITE mode, specify the following command:

```
ztopbuild -advanced_command {rtl_clock_mode -enable -replicate_lite}
clock_delay -perf_mode 1 (or 2)
```

`ClockDelayPort` that does not require message ports (CDP Lite) starts to support SDC MCP alignment mode 2. SDC MCP alignment provides tolerance with fastest clock period tolerance window. To enable the SDC MCP alignment mode, specify the following command:

```
clock_delay -sdc_mcp_mode 2
```

You can verify the activation of this mode in the following log files:

- `CDP lite`: In `zebu.work/rt_clock.xref`, search for the following line

  ```
  set rtl_clock_cdp_lite_mode true
  ```

- `SDC MCP mode 2`: In `zebu.work/rt_clock.xref`, search for the following line

  ```
  set rtl_clock_sdc_mcp_clock_group_mode 2
  ```

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";
```

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`”.

Using ZeBu Clock Delay Primitive

ZeBu clock delay primitive can be used as follows:

```
clockDelayPort #(
    .initial_dutyLow       ( 25 ),
    .initial_dutyHigh      ( 15 ),
    .initial_phase         ( 0  ),
```

Feedback

```
    .initial_resetValue    ( 0  ),
    .initial_resetDuration ( 1  )
  ) u_clk_gen (
    .clock ( clk ),
    .reset ( reset )
  );
```

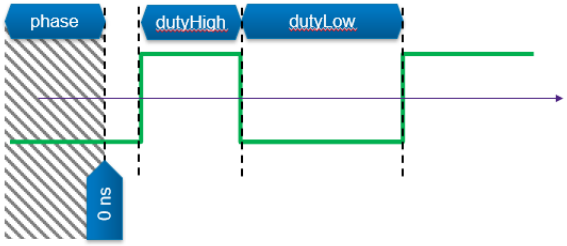
where,

Duration of `dutyLow`, `dutyHigh`, phase and `resetDuration` are expressed in ps

When `phase` = `0`, the clock start at `0`

When `phase` = `dutyLow`, the clock start at `1`

Figure 40 Clock Delay Primitive



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.

Feedback

Example

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

- `A0 #13`
- `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 can loop in three `driverclock` cycles compared to eight in the regular scheduler.

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

Limitations With clockDelayPort

Limitation With Dynamic Reconfiguration of ClockDelayPort

When you reconfigure a clock in the clock alignment mode 2, the following limitations are applicable:

- The fastest clock cannot become slower than any other clocks due to the reconfiguration of the clock.
- The fastest clock cannot be slowed down to an extent where slower clocks move to a faster clock group.
- Slower clocks cannot be made as fast where slower clocks move to a faster clock group.
- Clocks are grouped into frequency groups identified by a group number N such that the clock is at least 2^(N-1) slower and less than 2^N slower than the fastest clock.

Limitation With the Clock Alignment Feature

When more than one clockDelayPort is used to generate reset signals, the front-end issues the following warning and ignores generation of all reset signals except the one generated using the `clockDelayPort` of the fastest clock. The front-end is likely lead to runtime failure.

```
Warning-[ZEBU-CLK-DELAY-SDCMCP-CLKDELAY-RESET-NOSPT] Reset is not
 supported
```

Feedback

Avoid generation of reset signals using `clockDelayPorts` if you want to keep the design functional while taking advantage of the clock alignment feature.

RTL Delays

This section describes time-based clock control using RTL delays in the following topics:

- Enabling Clock Delay Feature
- Supported Verilog Language Subset
- Unsupported Constructs

Enabling Clock Delay Feature

When using delays in the RTL design, enable the clock delay feature with the following UTF command:

```
clock_delay -module <module_list>
```

`<module_list>` specifies all design modules in which delays are to be supported. All modules listed in these commands are supported by time-based control.

```
initial clk = 0;
always #10 clk = !clk;
initial forever #10 clk = !clk;
```

The following options are available with the `clock_delay` command:

- To enable the delay synthesis on a specific module, specify the `-module {…}` option.

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 this option, 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 specify performance modes, use `-perf_mode`. Default value is `0`. The following options are supported:
  - `0`: No additional performance optimizations
  - `1`: Enables module level clock delay replication
  - `2`: Enables FPGA level clock delay replication
  - `3`: No replication, clock delays and `$time` consumers are moved to terminal FPGA

- `4`: No replication, clock delays are moved to terminal FPGA. `$time` consumers stay at any partitioned FPGA

  Note:

  This is applicable only for ZeBu Server 4.

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

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)
date_of_event = $time;
```

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

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

- Transport delays

```
always @(posedge clk) begin
a <= #10 b;
end
```

- Intra assignment delays

```
initial begin
clk = #10 ~clk;
end
```

- (use either of the constructs mentioned here)

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

Partially Supported Constructs

Inertial delay in both gate primitives and continuous assignments is supported using the `-gate_delay true` option. Only single input gate primitives like `buf` and `not` are currently supported.

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

Unsupported Constructs

The following constructs are not supported.

- `$timeformat` system function

```
-$timeformat
```

- Delay applied to a net declaration:

```
wire #delay w;
```

- Delay applied to multi input gate primitives:

```
AND #delay inst(…);
```

Waveform Dumping for the Clock Delay Feature

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.

Reducing Disk Usage

During runtime, the clock delay feature has an option that enables you to reduce disk usage in case of offline post-processing of waveform formats like FWC, QiWC, DPI, SVA. For more information, see Appendix A, “Reducing Disk Usage With Clock Delay Feature.”

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 UTF 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.

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.

Timestamp Clock

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.

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;
#ifdef __cplusplus
```

Feedback

```
  _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);
```

Limitations of the Clock Delay Feature

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.

- #delay in tasks is only supported in transactor hierarchies. The following warning message is displayed in case #delay is applied to DUT hierarchies:

```
Warning-[UC-NSDF] Non-synthesizable delay found test.v, 25
'Delay control' construct found in module 'top'.
Warning-[UC-NSDF] Non-synthesizable delay found test.v, 26
'Delay control' construct found in module 'top'.
```

- Tolerance is not supported with variable delays.

Any settings for reset value and reset duration cannot be changed from **designFeatures**.

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 4

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

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);
}
```

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
- **zRci:** A Tcl-based program that controls the ZeBu system emulation runtime
- Verdi: A graphical interface that uses **zRci** to control the emulation

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
- zRci for Controlling Clocks and Managing Memory
- Cycle-Based C/C++ Co-Simulation
- Reading Signal Values During Runtime
- Changing the Signal Values During Runtime

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

Feedback

testbench. You can control the following runtime parameters using the **designFeatures** file:

- Emulation Speed
- Clock Definition
- Memory Initialization

Emulation Speed

By default, the emulation speed is set by the compilation. To analyze what affects emulation speed, see `zTime.log` in Viewing Timing Information in the zTime Reports. 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.

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`:

```
$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 = "_-_-"
```

Feedback

```
$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.

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.

zRci for Controlling Clocks and Managing Memory

The ZeBu Runtime Control Interface (**zRci**) provides a subset of commands for ZeBu and it is compatible with Tcl 8.6. You can use any Tcl command with **zRci**. To leverage Tcl 8.6 64-bit integer support in **zRci**, you need the 64-bit version of Tcl.

For details on using **zRci,** see the *ZeBu Unified Command-Line User Guide*.

For common zRci command-line options, see the **Running zRci in Batch Mode and GUI Mode** section in the *ZeBu Unified Command-Line User Guide*.

For more information, see the following subsections:

- Controlling Clocks
- Managing Memory

Controlling Clocks

This section describes the commands to control the ZeBu clocks.

Feedback

Note:

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

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 `get -clock_groups` 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 `get -clocks <group_name>` command returns the list of clocks for a specific clock group.

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.

zRci supports the following commands to enable clocks:

- `run`
- `run -clock <Clock's Name|Clocks Group Name>`
- `run <n> [-clock <Clock's Name>] [-autocheckpoint <cycles>]`
- `run <n>ms|us|ns|ps [-clock <Clock's Name>]`
- `run -wallclock <seconds>s|<minutes>m [-clock <name>]`

For details, see the **Tool Advancing Commands** in the *ZeBu Unified Command-Line User Guide*.

Disabling Clocks

zRci supports the following command to disable clock:

```
run -disable <Clock's Name|Clocks Group Name|all>
```

Getting the Number Clock Cycles Executed

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

Syntax:

```
get [-cycles] default_clock|primary_clocks|secondary_clocks
```

where, `[-cycles]` returns the current cycle for the returned clock, if specified.

For details, see the `get commands` section in the **ZeBu Unified Command-Line User Guide**.

For example:

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

```
zRci % get -cycles default_clock
hw_top.clk_11 0
zRci % get -cycles primary_clocks
{hw_top.clk_11 0} {hw_top.clk_00 0} {hw_top.clk_03 0}
{hw_top.clk_14 0} {hw_top.clk_05 0} {hw_top.clk_12 0}
{hw_top.clk_01 0} {hw_top.clk_07 0} {hw_top.clk_15 0}
{hw_top.clk_09 0} {hw_top.clk_10 0} {hw_top.clk_13 0}
{hw_top.clk_08 0} {hw_top.clk_06 0} {hw_top.clk_04 0}
{hw_top.clk_02 0}
```

Managing Memory

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

- Saving Memory Contents
- Loading Memory Contents

Saving Memory Contents

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

- `memory -store <memory> -file <filename>`: This command stores the memory content into a file. This command is dynamic, that is, it is available while emulation is running.
- `memory -store <memory> -buffer <buffer>`: This command stores the memory content in the buffer passed as a parameter.

Feedback

For details on **zRci,** see the **memory commands** section in the *ZeBu Unified Command-Line User Guide*.

Loading Memory Contents

You can load content into a memory either from a memory file using `memory -write <memory>` or from a buffer using `memory -load <memory> -buffer <buffer>`.

For details on **zRci,** see the **memory commands** section in the *ZeBu Unified Command-Line User Guide*.

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;
```

Feedback

```
    resetn    = dut_ccosim_drv.resetn;
    load      = dut_ccosim_drv.load;
    ena       = dut_ccosim_drv.ena;

    // reset the design
    *resetn     = b0;
    dut_ccosim->run(2);
    // force some signals
    *resetn     =  b1;
    *ena        =  b1;
    *load       =  b1;
    *din        = "0x00000000000000ffffffff0";
    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);
}
```

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.

For more information, see the following examples:

- C++ Example
- zRci Example

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;
```

zRci Example

```
// Reading a signal/register
get top.u_stb.cnt_error_in -radix dec
```

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.

For more information, see the following subsections:

- Depositing a Value on a Signal
- Injecting a Value on a Signal
- Forcing a Value on a Signal
- Releasing a Forced Signal

Depositing a Value on a Signal

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).

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 );
```

zRci Example

```
force top.u_stb.rstn 0 -deposit
force top.u_stb.proba_wr\[9:0\] [proba2hex $wr] -radix hexa -deposit
```

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.

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();
```

zRci Example

```
force top.u_stb.rstn 0 -deposit
force top.u_stb.proba_wr\[9:0\] [proba2hex $wr] -radix hexa -deposit
```

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.

C++ Example

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

zRci Example

```
force top.u_stb.rstn 0 -freeze
force top.u_stb.proba_wr\[9:0\] [proba2hex $wr] -radix hexa -freeze
```

Releasing a Forced Signal

C++ Example

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

zRci Example

```
release top.u_stb.proba_wr\[9:0\]
```

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 consists of the following topics:

- Enabling RunManager
- RunManager APIs

Enabling RunManager

By default, RunManager is available in runtime with ZeBu Server 4. 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, five RunManagers are instantiated in ZeBu Server 4 by default.

RunManager APIs

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

Table 10 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.
`RunManager::getClockStatus`	Returns the clock's status. The following values are enumerated:`ZEBU_RM_CS_RUNNINGZEBU_RM_CS_STOPPEDZEBU_RM_CS_STOPPED_TRIGGERZEBU_RM_CS_STOPPED_OTHER ZEBU_RM_CS_UNKNOWN`
`RunManager::advanceClock`	Allows to advance the clock using the number of edges specified as a parameter. 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.
`RunManager::advanceTime`	Allows to advance the simulation time using the amount of time specified as a parameter. 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.
`RunManager::advanceClockNonBlocking`	Allows to advance the controlled clock passed in an argument using the number of edges specified as a second argument.

Feedback

Table 10 APIs Supported with RunManager (Continued)

API Name	Description
`RunManager::interrupt`	Allows to interrupt the execution of the `RunManager::advanceClock` and `RunManager::advanceTime` methods.
`RunManager::freeGroup`	Allows to run a clock group indefinitely.
`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 runtime.

To use SVA in ZeBu, perform the following steps:

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

This section consists of the following topics:

- SVA Compilation
- Runtime Usage
- Post Processing SVA
- Supported SVA Subset

SVA Compilation

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

Commonly used options to control the assertion synthesis through UTF 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.

Runtime Usage

The synthesized assertions can be controlled at runtime through **zRci** or 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.

This section consists of the following sub-sections:

- Controlling Assertions From the C++ Testbench
- Controlling Assertions Using zRci
- Runtime Options for Assertion Control Tasks

Controlling Assertions From the C++ Testbench

ZeBu provides a 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.hh` header file.

For more information, see the following subsections:

- Starting Assertion Processing
- Stopping SVA Processing
- Resetting SVA Fail Counters

Feedback

- Dealing with Assertion Failures
- Changing the Reported Severity Level
- Disabling/Enabling Assertions and Signaling SVA Failures

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, `zsvaReport` (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.

Feedback

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);
```

By default, all failures are reported on the screen. However, in the live-processing mode, by using the following C++ API you can choose to report only the first SystemVerilog Assertion failure message of each SVA along with the total failure count:

```
void SVA::DisableReportingAfterFirstFailure ( const bool
disableReporting );
```

Alternatively, you can use the following C API

```
int ZEBU_SVA_DisableReportingAfterFirstFailure ( const bool
disableReporting );
```

See the following examples:

```
// Report only first failure of each SVA; default is true
SVA::DisableReportingAfterFirstFailure ();
SVA::DisableReportingAfterFirstFailure ( true );
// Report all SVA messages on screen

SVA::DisableReportingAfterFirstFailure ( false );
```

Note:

The `SVA::Stop()` API supersedes the `SVA::DisableReportingAfterFirstFailure()` API. Therefore, the `SVA::DisableReportingAfterFirstFailure()` API is disabled after `SVA::Stop()` API is called.

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);
```

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 Unified Command-Line User Guide*.

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 unisgned 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.

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.

Feedback

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 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.

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.

Controlling Assertions Using zRci

The synthesized assertions can be controlled at runtime through **zRci**. By default, assertions are disabled and no assertion failure message is reported.

For more information, see the following subsections:

- Starting Assertion Processing
- Stopping SVA Processing
- Getting SVA Failure Counters
- Resetting SVA Fail Counters

Starting Assertion Processing

To start the SVA processing, the `sva` UCLI command must be called. Assertions can be enabled in the following two modes:

- Live Processing Mode
- Post-Processing Mode

Live Processing Mode

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

```
sva -enable -report
```

Post-Processing Mode

In this mode, all assertion failure messages during design run are captured into an ZTDB for post-emulation analysis while using:

```
sva -start -file sva.ztdb
```

The **zsvaReport** post-processing tool (see Post Processing SVA), reads this report file and generates an assertion failure report.

Stopping SVA Processing

To stop SVA processing, run the `sva -stop` UCLI command.

Getting SVA Failure Counters

To record the failure counters, use the `sva -failures [-counters]` command. The `sva -failures` command returns the list of failed SVA assertions in a Tcl `dict` object.

SVA assertion failures are now reported based on the default clock during Stimuli Replay.

Feedback

Example 1

zRci Script

```
puts "1. FAILURES: [sva -failures]"
```

Output

```
1. FAILURES: xtor_32.ASSERTION_MODULE.after_reset
 xtor_32.ASSERTION_MODULE.valid_ready_wraddr
 xtor_32.ASSERTION_MODULE.valid_ready_wrdata
 xtor_32.ASSERTION_MODULE.wlast
```

Example 2

zRci Script

```
puts "2. FAILURES & COUNTS: [sva -failures -counters]"
```

Output

```
2. FAILURES & COUNTS: xtor_32.ASSERTION_MODULE.after_reset 2
 xtor_32.ASSERTION_MODULE.rlast 0
 xtor_32.ASSERTION_MODULE.valid_ready_wraddr 2
 xtor_32.ASSERTION_MODULE.valid_ready_wrdata 2
 xtor_32.ASSERTION_MODULE.wlast 3
```

For more UCLI commands on controlling assertions using **zRci,** see the **Assertions Commands** section in the *ZeBu Unified Command-Line User Guide*.

Resetting SVA Fail Counters

To reset the failure counters, use the `config sva_reset_counters` command.

For more UCLI commands on controlling assertions using **zRci,** see the **Assertions Commands** section in the *ZeBu Unified Command-Line User Guide*.

Runtime Options for Assertion Control Tasks

To disable/enable ACTs at runtime, the following C++ APIs are provided:

- `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()`.

Feedback

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()`.

---

zsva_trigger

`zsva_trigger` is an implicit dynamic trigger (`zceiTrigger`) that is available with assertions. It is enabled during the compilation with the following UTF command:

```
assertion_synthesis -enable all
```

`zsva_trigger`: The `zsva_trigger` Dynamic Trigger fires when an enabled SystemVerilog Assertion (SVA) fails and it is synchronous with the SVA failure.

Note:

For the same failing assertion, the SystemVerilog Assertion failure is reported using FWC technology, which is asynchronous. Therefore, the reporting of failure can happen several cycles later.

To stop the emulation on any assertion failure, enable the reporting using the following command after the `sva -start` command:

```
sva -enable -notifier
```

If you want to execute some code on any assertion failure, set the callback execution when `zsva_trigger` is encountered as shown in the following example:

```
proc act_proc { } {
  puts " SVA trigger executed at cycle [run 0]"
  run 1
  run 0
  puts "act_proc terminating run at cycle [run 0] because of [sva
 -failures] assertion failure"
  exit
}

sva -start
sva -enable -notifier
stop -enable zsva_trigger -action act_proc
run
```

When the preceding example is executed, the following type of output is generated at runtime:

```
- ZeBu : zRci: Stop condition `zsva_trigger` reached.
```

Feedback

```
-- ZeBu : zRci:

Callback added to queue: `act_proc` SVA trigger executed at cycle 49 **
Error: unnamed$$_2: Assertion error..

Time: 49ps Scope: top.unnamed$$_2

File: ../common/hdl/top.v Line: 84 act_proc terminating run at cycle 50
because of {top.unnamed$$_2} assertion failure
```

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:

```
$ zsvaReport  [--ztdb <.zsva filename>] [--log <filename>] [--zebu-work
 <zebu.work>] [--output <filename>] [--severity <severity>] [ --no-error]
 [ --max-errors-sva <number>] [--max-errors-all <number>] [ --begin
 <time>] [ --end <time] [ --timescale <timescale>] [--force]
```

where,

- `-l [ --log ] <filename>`: Specifies the log file name (default: `zsvaReport.log`)
- `-z [ --zebu-work ] <zebu.work>`: Specifies the ZeBu work directory.
- `-i [ --ztdb ] <.zsva filename>`: Specifies the ZTDB input directory.
- `-o [ --output ] <filename>`: Specifies the output file to save the report.
- `-s [ --severity ] <severity>`: Specifies the minimum severity level that is reported. Values can be `fatal|error|info|display|success`. Default is `error`.
- `-n [ --no-error ]`: Displays only a report and does not display the message report.
- `-m [ --max-errors-sva ] <number>`: Specifies the maximum number of error messages that are reported for each SVA.
- `--max-errors-all <number>`: Specifies the maximum number of error messages displayed for all SVA.
- `-b [ --begin ] <time>`: Specifies the begin time of the conversion (default: first captured time).
- `-e [ --end ] <time>`: Specifies the end time of the conversion (default: last captured time).
- `-t [ --timescale ] <timescale>`: Specifies time per cycle (for example, 10ns).

Feedback

- `--force`: Forces conversion on unfinished ZTDB.
- `-h [ --help ]`: Displays the help.

For example,

```
zsvaReport -i dump.zsva -z <zebu.work>
```

Where, `dump.zsva` is the directory passed to the `SVA::Start` method during runtime.

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	`cycle_delay_range`	`## integral_number`	S
		`##(const_expr)`	S
		`##(const_expr : const_expr)`	S
		`##(const_expr : $)`	S*
	Consecutive repetition	`[* const_expr]`	S
		`[* 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*

Feedback

Supported SystemVerilog SVA Subset			Status
	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
		`$past`	S
		`$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

Feedback

Supported SystemVerilog SVA Subset			Status
	Sequences involving function calls		S
	Subroutine call on match of a sequence		U
	`Implicit first_match`		U
PROPERTIES	Property instances	Local instantiations	S
		Cross-module instantiations	U
		Recursive instantiations	U
	`not`		S
	`or`		S
	`and`		S
	`if...else`		S*
	`disable iff`		
	Implication		S*
	Combination of implication and first_match not supported		S*
	Property instances		S
	Local variables	`accept_on`	U
	Abort properties	`reject_on`	S
CLOCK DETECTION	Simple clock resolution		S
	Complex clock resolution		S
	Multi-clocked sequences		S
	Multi-clocked properties		S
	Clock flow analysis		S

Feedback

Supported SystemVerilog SVA Subset		Status
ASSERTION CONTROL TASKSFeature supported through C++ API	`$assertkill`	S
	`$asserton`	S
	`$assertoff`	U
CONCURRENT ASSERTIONS	`assert`	S
	`assume`	S
	`cover`	S*
	Concurrent assertions outside procedural code	S
	Concurrent assertions inside procedural blockAssertions with clocking event different from clocking event of procedural block not supported	S*
	Concurrent assertions inside procedural loopAssertions 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

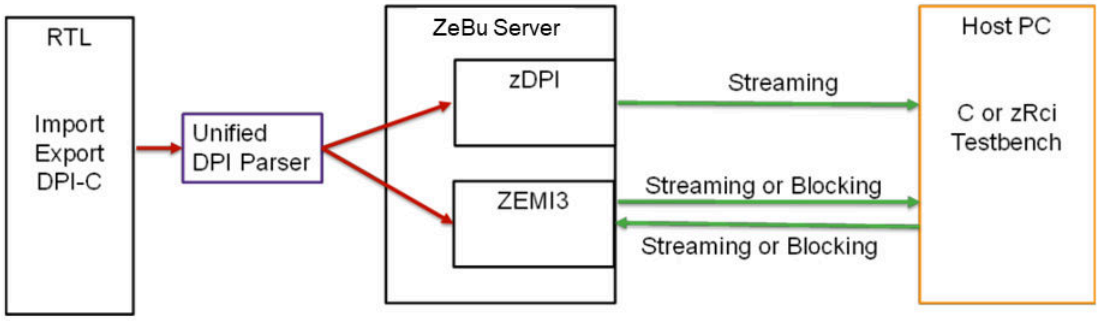
ZeBu DPI Technologies

ZeBu supports SystemVerilog Direct Programming Interface (DPI) imported function calls inside the DUT. The ZeBu hardware-software infrastructure supports two types of architecture for DPI:

- zDPI
  - This is used for one way communication only, that is, for streaming DPI.
  - It uses *FWC* as a communication channel.
- ZEMI3
  - This is used for two way communication, that is, streaming and non-streaming DPI.
  - It is required to make synthesis with behavior compiler or ZEMI3 compiler.
  - It uses `zceiMessagePort` as a communication channel.

ZeBu also provides a unified parser called **Unified DPI** that enables all DPI calls to automatically synthesize as either ZEMI3 or zDPI. Unified DPI supports all types of out-of-the-box DPI, without having to declare ZEMI3 transactors block transactions: ZEMI3 behavior compiler is used for DPI that cannot be zDPI.

Figure 41 Supported ZeBu DPI Technologies



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 `zRci` (Tcl commands) or using a C++ API.

This section discusses the following topics.

- Supported DPI Function Calls
- Compilation
- Writing DPI Import Functions
- ZeBu API to Control DPI Functions at Runtime
- Processing DPI Function Calls Offline
- Optimization Guidelines
- Limitations of the ZeBu DPI Technologies
- Example of DPI Function Calls to Print Counter Value

Supported DPI Function Calls

With zDPI technology, ZeBu supports the following types of DPI function calls in 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.
- Always block: only functions call in always block are supported
- System-functions: not totally supported, see Verilog System Tasks for details

Unified DPI

With Unified DPI technology , ZeBu supports the following types of DPI function calls in SystemVerilog source files:

- Imports and export: Imported and call exported functions.
- Functions only: Only operations that do not consume time are possible.
- Inputs and outputs: Since data flows in both directions, the function can have outputs.
- import functions (with return value) are allowed, that is, with outputs.

- Always and initial block: support functions call in initial and always block
- System-functions are handled correctly

Unified DPI Versus zDPI

- Regarding communication channel
  - zDPI use FWC as communication channel
  - Unified DPI can use:
  - FWC as communication channel for DPI calls that are handled with zDPI ZeBu architecture
  - `zceiMessagePort` as communication channel for DPI calls that are handled with ZEMI3 architecture
- Regarding streaming mode
  - zDPI is used to stream data HW-> SW, does not stop hardware clocks (unless FIFOs are full) => Input-Only Import functions, ideal for "DPI Monitors"
  - Unified DPI handle stream from HW -> SW and from SW-> HW, it synthesize automatically all DPI calls as ZEMI3 or zDPI with unified parser
  - Regarding system task/function:
  - zDPI handle system task/function but not totally. See Verilog System Tasks for details.
  - Unified DPI handle system task/function correctly

The following table lists what is supported for every architecture for different configuration setup:

Table 12 Unified DPI Versus zDPI - Configuraion Setup

HasOutputs	Context	Process Type	Pragma module or function	zDPI Default	ZEMI3 transator	Unified DPI
Yes				Ignored	Blocking	Blocking
No	Yes	Initial		Ignored	Blocking	Blocking
No	No	Initial: Only 1 call		Ignored	Blocking*	Blocking*

Feedback

Table 12 Unified DPI Versus zDPI - Configuraion Setup (Continued)

HasOutputs	Context	Process Type	Pragma module or function	zDPI Default	ZEMI3 transator	Unified DPI
No	No	Initial: More than 1 call		Ignored	Streaming ZEMI3	Streaming ZEMI3
No	No	Always		Streaming zDPI	Streaming ZEMI3	Streaming zDPI
No	Yes	Always		Streaming zDPI	Blocking	Streaming zDPI**
No	Yes	Always	zemi3_allow_export			Blocking
		Export		Ignored	Supported	Supported

Note:

If No-Context No-Output function is called only once and only from initial, it is **blocking**. If it is called more than once from initial, or called from initial and process, it is **streaming**. These behaviors can be overwritten by `zemi3_stream` pragma.

Any function that is zDPI but that shares a process with ZEMI3 changes to ZEMI3-Streaming.

Compilation

This section consists of the following topics:

- zDPI (Default Technology)
- Unified DPI
- Compiling ZeBu DPI files

zDPI (Default Technology)

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

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

```
dpi_synthesis -enable ALL
```

Feedback

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

`dpi <instance name>`	Selects the given DPI function
`module <module name>`	Selects all DPI calls in all instances of the given module
`tree <hierarchy name>`	Selects all DPI calls below the given hierarchy
`path <hierarchy name>`	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_synthesis -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:

```
+-------------------------------------------------------------------+
|        Function Name|          Path|# Inp|# Out|
      Source File|Source L|Call |                   Filtered Reason|DPI
 Type|       Status|
+-------------------------------------------------------------------+
|fifo_usage_spy_notify|fifo_usage_spy|    6|
 0|/remote/user1/sources/fifo_usage_spy.sv|      22|    1|  DPI synthesis
 enabled for module|  import|  synthesized|
+-------------------------------------------------------------------+
```

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
  ```

Feedback

- 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`.

Unified DPI

- Unified DPI is controlled with the `dpi_synthesis` UTF command:

  ```
  dpi_synthesis -enable ALL -enable_wls true
  ```

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

  - `-timestamp <bool>` : Enable Timestamps with RTL Clocks
  - `-offline <bool>` : Support Offline Mode (for use with `zdpiReport`)
  - `-timestamp <bool>`: Support `global_serialized_opt`
  - `-optimize_width <bool>` : Enable optimization of input bits for DPI call
  - `-perf_mode <int>`: Enable capacity optimization by 1 register per DPI bit argument
  - `-disable_initial_blocks <bool>`: Disable the handling of DPI calls in initial blocks
  - `-disable_exports <bool>`: Disable the handling of DPI export DPI

- To enable the synthesis of system task/function used in DUT hierarchy with Unified DPI, must add the following UTF command:

  ```
  system_tasks -task "name of sytem task/function"
  ```

  Example:

  ```
  system_tasks -task "\$display"
  ```

  It is possible to have a finer control on the synthesis of system task/function with the following options:

  - `-task #task_name`: Specify system task.
  - `-replace #replace_value`: Replace system task with given string.
  - `-exclude #list_of_hierarchies_to_exclude`: Specify list of hierarchies to exclude.
  - `-remove`: SVA Global option to remove all system tasks from SVAs .

Feedback

- `-enable`: Enable built-in support for system task. Currently only supports `$display`, `$displayh`, `$displayb`.
- `-module #list_of_modules`: Specify list of modules under which system task are not removed and are given to synthesis.

Exception:

`$value$plusargs` and `$test$plusargs` are synthesized when enabling Unified DPI even if the `system_tasks` command is not used.

Analysis of Unified DPI Log Files

The results of enabling Unified DPI can be found in the following log files under ZeBu compilation database: `zcui.work/design/zxtor/zdpi/zdpi.log`.

The log file contains the following information:

Note from VCS

This part contain the Notes from VCS that concern the DPI and the system task/function synthetized with "Unified DPI"

The Note contains the raison of synthesizing with "Unified DPI", the path of the RTL file and the line number in which the DPI or the system task function is defined

Figure 42 Example 1: Analyzing of Import DPI Called in the Initial Block

```
Note-[ZEMI-ADIF] Analysing DPI imported function
../common/hdl/hw_top.sv, 8
"fn1"
  Analysing (pure) DPI imported function fn1 (fn1) in module xtor

Note-[ZEMI-DPI-NON-STREAMING] Import DPI called in initial block made non-streaming
../common/hdl/hw_top.sv, 8
"fn1"
  Non-context, non-output import DPI fn1 in module xtor is called in initial
  block only once
  Use (* zemi3_stream = 1 *) to make it streaming.

Note-[ZEMI-ADIF] Analysing DPI imported function
../common/hdl/hw_top.sv, 10
"fn3"
  Analysing (context) DPI imported function fn3 (fn3) in module xtor
```

Figure 43 Example 2: Analyzing of Import DPI Called in the Initial Block

```
Note-[ZEMI-AST] Analysing system task
../common/hdl/toto.sv, 4
"$test$plusargs("HELLO")"
  Analysing system task $test$plusargs in module toto

Note-[ZEMI-AST] Analysing system task
../common/hdl/toto.sv, 5
"$test$plusargs("HE")"
  Analysing system task $test$plusargs in module toto
```

DPI Exports Summary

This table contains the list of export functions defined in DUT hierarchy.

name	type	in bits	out bits	protocol	streaming

DPI Imports Summary

This table contains the list of import functions defined in DUT hierarchy.

name	type	in bits	out bits	context	protocol	streaming	prefetchable
fn1	function	1	0	No	--	No	No
fn3	function	1	0	Yes	--	No	No

Message Ports Summary

This table contains the list of In and Out ports generated for all DPI calls and system task/ Function synthesis by "Unified DPI".

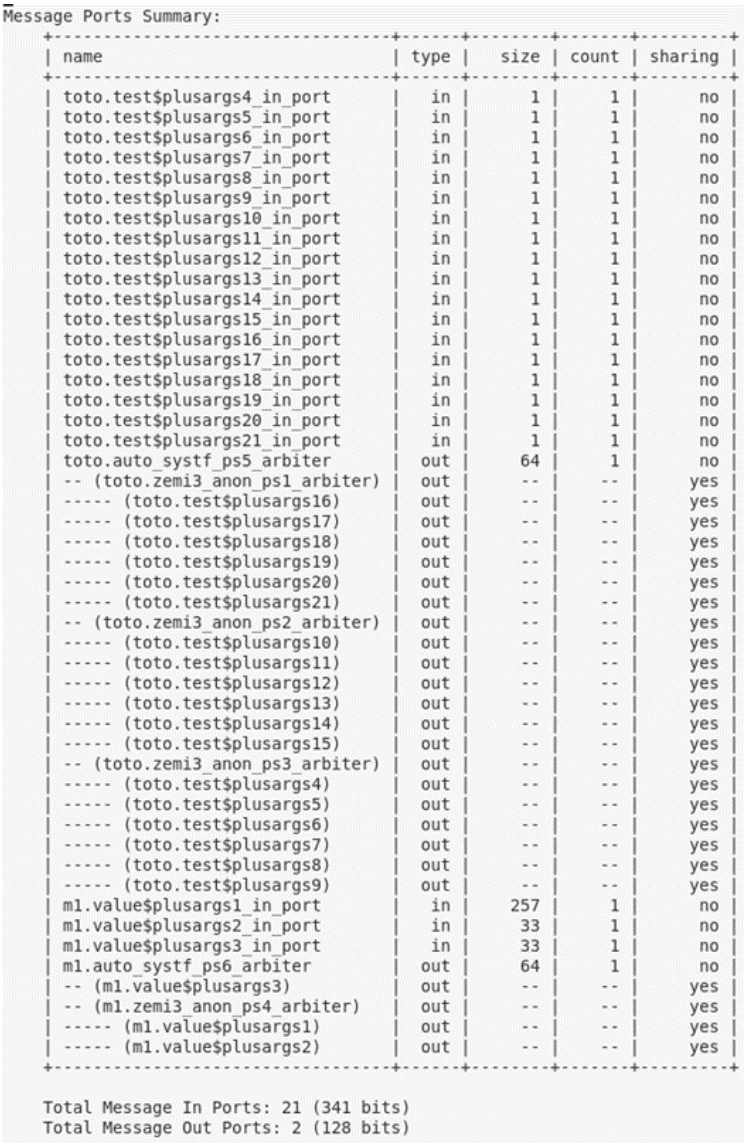
Figure 44 Example 1: List of In and Out Ports of Import DPI Called in Initial Block

name	type	size	count	sharing
xtor.fn3_in_port	in	1	1	no
xtor.fn1_in_port	in	1	1	no
xtor.zemi3_anon_ps1_arbiter	out	64	1	no
-- (xtor.fn1)	out	--	--	yes
-- (xtor.fn3)	out	--	--	yes

```
Total Message In Ports: 2 (2 bits)
Total Message Out Ports: 1 (64 bits)
```

Feedback

Figure 45 Example 2: List of In and Out Ports of System Task ($plusargs)



```
Message Ports Summary:
+----------------------------------+------+--------+-------+---------+
| name                             | type |   size | count | sharing |
+----------------------------------+------+--------+-------+---------+
| toto.test$plusargs4_in_port      |   in |      1 |     1 |      no |
| toto.test$plusargs5_in_port      |   in |      1 |     1 |      no |
| toto.test$plusargs6_in_port      |   in |      1 |     1 |      no |
| toto.test$plusargs7_in_port      |   in |      1 |     1 |      no |
| toto.test$plusargs8_in_port      |   in |      1 |     1 |      no |
| toto.test$plusargs9_in_port      |   in |      1 |     1 |      no |
| toto.test$plusargs10_in_port     |   in |      1 |     1 |      no |
| toto.test$plusargs11_in_port     |   in |      1 |     1 |      no |
| toto.test$plusargs12_in_port     |   in |      1 |     1 |      no |
| toto.test$plusargs13_in_port     |   in |      1 |     1 |      no |
| toto.test$plusargs14_in_port     |   in |      1 |     1 |      no |
| toto.test$plusargs15_in_port     |   in |      1 |     1 |      no |
| toto.test$plusargs16_in_port     |   in |      1 |     1 |      no |
| toto.test$plusargs17_in_port     |   in |      1 |     1 |      no |
| toto.test$plusargs18_in_port     |   in |      1 |     1 |      no |
| toto.test$plusargs19_in_port     |   in |      1 |     1 |      no |
| toto.test$plusargs20_in_port     |   in |      1 |     1 |      no |
| toto.test$plusargs21_in_port     |   in |      1 |     1 |      no |
| toto.auto_systf_ps5_arbiter      |  out |     64 |     1 |      no |
| -- (toto.zemi3_anon_ps1_arbiter) |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs16)     |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs17)     |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs18)     |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs19)     |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs20)     |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs21)     |  out |     -- |    -- |     yes |
| -- (toto.zemi3_anon_ps2_arbiter) |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs10)     |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs11)     |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs12)     |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs13)     |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs14)     |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs15)     |  out |     -- |    -- |     yes |
| -- (toto.zemi3_anon_ps3_arbiter) |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs4)      |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs5)      |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs6)      |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs7)      |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs8)      |  out |     -- |    -- |     yes |
| ----- (toto.test$plusargs9)      |  out |     -- |    -- |     yes |
| m1.value$plusargs1_in_port       |   in |    257 |     1 |      no |
| m1.value$plusargs2_in_port       |   in |     33 |     1 |      no |
| m1.value$plusargs3_in_port       |   in |     33 |     1 |      no |
| m1.auto_systf_ps6_arbiter        |  out |     64 |     1 |      no |
| -- (m1.value$plusargs3)          |  out |     -- |    -- |     yes |
| -- (m1.zemi3_anon_ps4_arbiter)   |  out |     -- |    -- |     yes |
| ----- (m1.value$plusargs1)       |  out |     -- |    -- |     yes |
| ----- (m1.value$plusargs2)       |  out |     -- |    -- |     yes |
+----------------------------------+------+--------+-------+---------+

Total Message In Ports: 21 (341 bits)
Total Message Out Ports: 2 (128 bits)
```

zDPI Imports Summary

This table contain the list of DPI call that are synthetized with zDPI.

name	type	in bits	out bits	context
fn2	function	1	0	No
fn4	function	1	0	Yes

A specific header file that includes the prototypes of the synthesis DPI function calls:

```
grep extern zcui.work/design/zxtor/zdpi/<hw_top>.h
```

Compiling ZeBu DPI files

After `zebu.work` is built, ensure that the ZeBu DPI files are compiled using one of the following commands:

In case of zDPI:

```
make -C ./zebu.work -f grp0_ccall.mk clean comp
```

In case of ZEMI3:

```
make -C ./zebu.work -f dpixtor.mk clean all
```

Writing DPI Import Functions

This section consists of the following topics:

- C/C++ Language Compatibility
- Header Files

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.

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"
```

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.

This section consists of the following topics:

- C++ API to Control DPI Functions
- zRci Tcl Commands to Control the DPI Functions
- Runtime for Unified DPI

C++ API to Control DPI Functions

The following sections provide more details on C++ APIs:

- Clocking Mode
- Enabling Synchronization of DPI Calls
- Selecting Function Calls Only on Input Changes

- Loading Dynamic Libraries
- Starting DPI Call Processing
- Stopping DPI Call Processing

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

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.

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 ornegedge 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.

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

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.

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.

Feedback

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.

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 not `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.

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 (.).

zRci Tcl Commands to Control the DPI Functions

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

- `ccall -sampling_clock -group <group_name>`
- `ccall -sampling_clock -expression <clock_expression>`
- `ccall -start`
- `ccall -load <so>`
- `ccall -enable`
- `ccall [-scope <scope_name> [-name <function_name> | -n <call_number>] ] -enable|-disable`
- `ccall -flush`
- `ccall -enable_synchronization|-disable_syncronization`
- `ccall -offline_spec <filename list>`
- `ccall -dump_offline [-dump_all] <filename>`

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 **zRci** script to control the **zDPI** feature, see Controlling the Calls from zRci.

Runtime for Unified DPI

- Enable ZEMI3 Manager.
- Enable zDPI/CCall, as required.

Example:

```
zemi3 -lib ./tb.so
zemi3 -lib ./zebu.work/zebu_top.so
zemi3 -enable
start_zebu zrci_run_dir
ccall -load ./tb.so
ccall -load ./zebu.work/grp0_ccall.so
ccall -enable_synchronization
ccall -enable
run 100
exit
```

You can provide `plusargs` at ZeBu runtime using any of the following methods:

- Through `designFeatures` file
- Through command-line arguments and then calling the Board API, `Board::setCommandLineArgs(argc, argv)`, from the user test bench.

In addition, you can read the `plusargs` at ZeBu runtime using any of the following methods:

- `vpi_get_vlog_info`: Support for this VPI function is present in ZeBu. You need to include the header file, `vpi_user.h`, for this function from the following location: `$(VCS_HOME)/include` and link it to the `libZebuVpi.so` library.
- `mc_scan_plusargs`: Include the header file, `veriuser.h`, for this function from following location: `$(VCS_HOME)/include` and link it to the `libZebuVpi.so` library.

You can use either of these methods to read `plusarg`s from your DPI C code.

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).

Feedback

Identifying the DPI Functions

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

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.

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.

Feedback

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.

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.

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 `dumpeddumpAllActivatedTracers=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");
```

Feedback

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
```

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.

Using the zdpiReport Tool

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

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

Table 13and Table 14 list mandatory parameters and options, respectively, for the **zdpiReport** command.

Table 13 zdpiReport Mandatory Parameters

Parameters	Description
`-f <flist filename>`	Specifies the filename containing the names of DPI functions to be executed by **zdpiReport**. Supports system tasks such as `$display`.Selects system task in any scope to be enabled. **Example:** To enable system task only in `top.d1` `scope:$display scope=top.d1`
`-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.

Feedback

Table 13 zdpiReport Mandatory Parameters (Continued)

Parameters	Description
`-delete-processed-fwc`	Specifies deletion of processed FWC data from ZTDB when **zdpiReport** is running in the on-the-fly mode. This helps to reduce the disk space usage by **zdpiReport**. **Note:** This option deletes the data for all the FWCs captured in the ZTDB irrespective of the data marked to be executed by **zdpiReport**.

Table 14 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.
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.	
To diagnose decoding	

Feedback

Table 14 zdpiReport Options (Continued)

Parameters	Description
`[-diag <features>]`	Enables specific diagnostic features. Where, • `<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: ◦ 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.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: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. The switch can be used multiple times: `zdpiReport -diag dump -diag from=12.` If neither `dump`, `nocall` nor `profile` is used, `dump` is implied.

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.

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]]`.

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
```

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
   ...
```

Optimization Guidelines

This section contains recommendations for optimal use of the **zDPI** feature.

It consists of the following topics:

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

Feedback

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.

Note:

The `svOpenArrayHandle` data type is not supported. If this data type is found in any design, a warning message is logged in `vcs_splitter_VCS_Task_Builder.log`.

The following table lists the mapping between SystemVerilog types with C types:

Table 15 Mapping Between SystemVerilog Types with C Types

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

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.

Defining Sampling Clock Frequency

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

$$\frac{50\ MHz}{((size\_in\_bits \div 32) + 5)}$$

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:

$$5\ MHz \times 8\ cycles = 40\ MHz$$

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

Feedback

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*.

Checking With zDPI Enabled

When the `ZEBU_IGNORE_DPI` environment variable is set, s 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.

Limitations of the ZeBu DPI Technologies

zDPI

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

It consists of the following topics:

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.

Performance

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

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.

Unified DPI

- If using Stimuli Capture and Replay technology, must be @speed
- All zemi3 serviced in a single loop
- No automatic merging of 2 processes

- ZEMI3 supported behavioral constructs do not work if process does not include DPI.
- `$value$plusargs` and `$test$plusargs` are not supported in continuous assignment expression.

Example of DPI Function Calls to Print Counter Value

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.

This example demonstrates two different methods to control the DPI function calls:

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

Design

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

- System Verilog Code
- Synthesis

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

    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
```

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
```

Feedback

```
print_count        {counter}  65                  ../../../src/counter.v
 38          0
```

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
```

Controlling DPI Calls from a C++ 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 **zRci**. However, they cannot be controlled simultaneously by both the C++ testbench and **zRci**.

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 = z->getSignal("reset");
    Signal *load  = z->getSignal("d");
    Signal *data  = z->getSignal("data");
    Signal *ci    = z->getSignal("ci");
    Signal *pce   = z->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;
    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
```

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
```

Controlling the Calls from zRci

The DPI calls implemented by the dynamic library (`dpi.so`). The control is done directly with UCLI commands in **zRci**.

The following is an example of the script (`script_dpi.tcl`) that controls the DPI function calls:

```
# open the board
start_zebu

# load of the DPI dynamic library
ccall -load my_fifo_monitor.so
# Start of all the DPI calls
ccall -enable
# run the clocks clk
ZEBU_Clock_enable clk 200000

run 100ms

# stop dpi processing
ccall -disable
# close the board
close_zebu
```

9

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 section consists of the following topics:

- Save and Restore - Recommendations
- C++ Method for Save and Restore
- C++ Example
- zRci Method for Save and Restore
- zRci Example

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)

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)
  ```

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.

Feedback

C++ Example

The following is an example of C++ method for save and restore:

```
FastHardwareState hwState;
hwState.initialize( _zebu );
hwState.capture();
hwState.save( "hw_state" );
hwState.clean();

[…]
_zebu = Board::restoreHardwareState( "hw_state", "zcui.work/zebu.work" );
```

zRci Method for Save and Restore

To save and restore the hardware state using **zRci**, use the following:

```
checkpoint    # Save or restore HW or HW and SW state
       checkpoint -save <hw state name>
       checkpoint -restore <hw state name>
       checkpoint -fullsave <hw/sw state name>
       checkpoint -check_suspend
       checkpoint -info [<name>] -cycles | -time
       checkpoint -info [<name>] -text
       checkpoint -list [-full] [-text]
```

where,

- `[-save <state name>]`: Saves the HW state
- `[-fullsave <state name>]`: Saves HW and SW states
- `[-restore <state name>]`: Restores the HW state
- `[-list]`: Lists all saved HW states
- `[-text]`: To be used with `-info` or `-list`, to print data stored within **zRci**'s database in readable format
- `[-full]`: Makes `-list` to print all saved HW+SW states instead
- `[-check_suspend]`: Checks for suspend requests
- `[-cycles]`: To be used with `-info`, to fetch data stored within **zRci**'s database
- `[-time]`: To be used with `-info`, to fetch data stored within **zRci**'s database for RTL clocks
- `[-info [<name>]]`: Returns information such cycle, reference clock and if DMTCP related to a given checkpoint or all, if no name is passed.

zRci Example

The following is an example usage of **zRci** for save and restore:

```
checkpoint -save myHwState0

[…]

checkpoint -restore myHwState0
```

10

Clock Cone Visualization in zBrowser

A **Clock** panel is introduced in **zBrowser** to support the following:

- Pre-computes clock-cone statistics
- Captures the registers in the clock cone as a CSV file to search for known registers
- Displays a clock path of interest to its primary input based on maximum logic levels culminating at every intermediate point in the path
- Displays a clock path of interest to its primary input based on maximum Total Fan-In (TFI) culminating at every intermediate point in the path
- Traces any clock of interest to its final output
- Finds the top-10 (or top-N^+) deepest clocks by logic levels
- Finds the top-10 (or top-N^+) clocks by TFI
- Displays global statistical information about the clock cone
- Displays properties such as Depth, Cluster-Id, TFI and Total Fan-Out (TFO) of an instance/signal in clock-cone selected in **zBrowser**

The following figure shows the **Clock** panel in **zBrowser**.

Feedback

Figure 46 Clock Panel in zBrowser

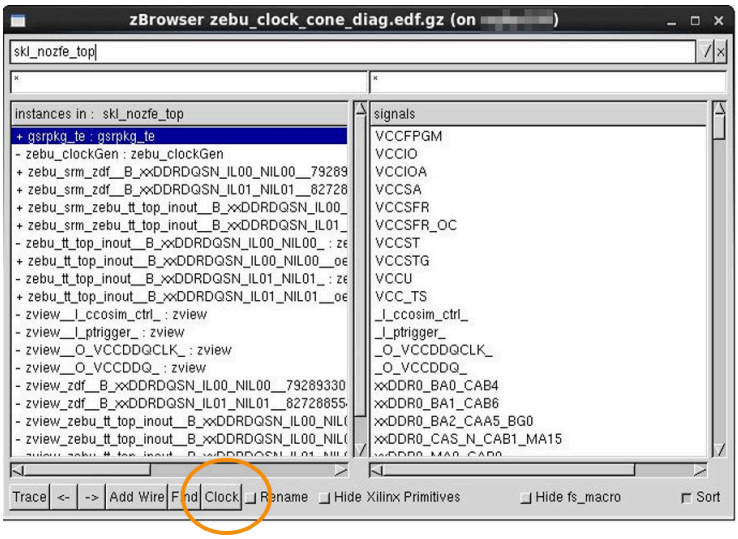
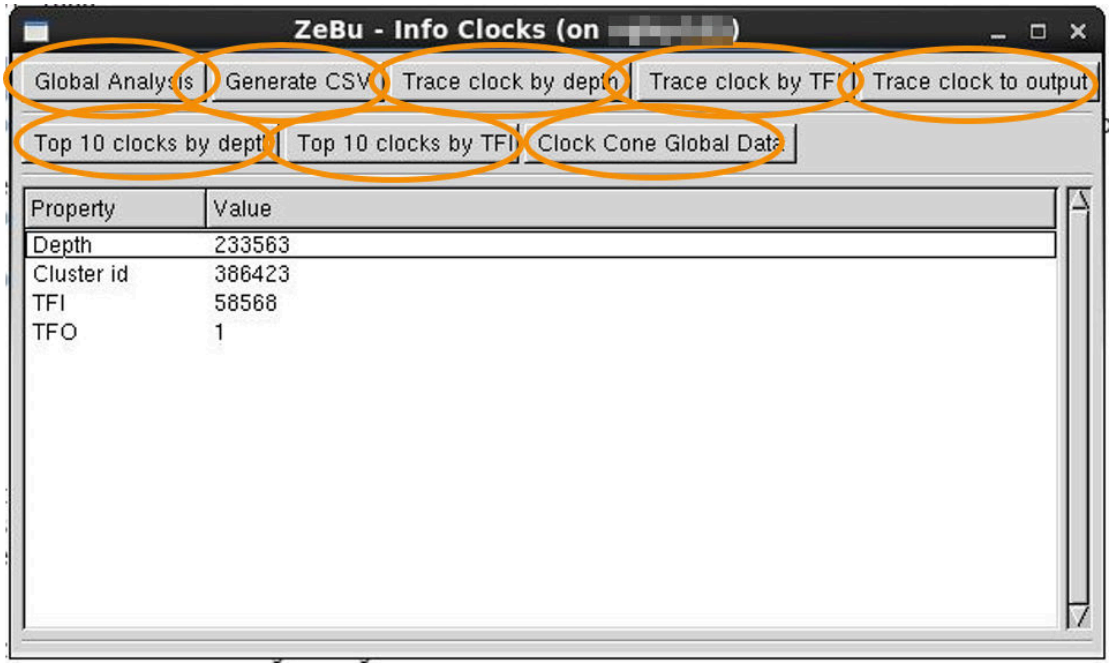


Figure 47 Attributes of Clock



This section consists of the following topics:

- Enabling Clock Cone Visualization
- Finding Clocks with Maximum Depth
- Finding Clocks with Maximum TFI

Feedback

- Tracing a Known Clock Signal
- Tracing the Clock Path Contributing to Maximum Logic Levels
- Tracing the Clock Path Contributing to Maximum IO cut
- Tracing an Intermediate Clock Signal to its Final Output
- Analyzing Clock Localization

Enabling Clock Cone Visualization

To enable clock cone visualization, set the following environment variable during **zTopBuild** compilation:

```
ZEBU_CLOCK_CONE_DIAG 1
```

The `zebu_clock_cone_diag.edf.gz` file is created in `zebu.work`. The EDIF is created before clock-cone is modified for FGS as follows:

```
zBrowser zebu_clock_cone_diag.edf.gz -clock
```

This makes it easier to correlate with RTL register names.

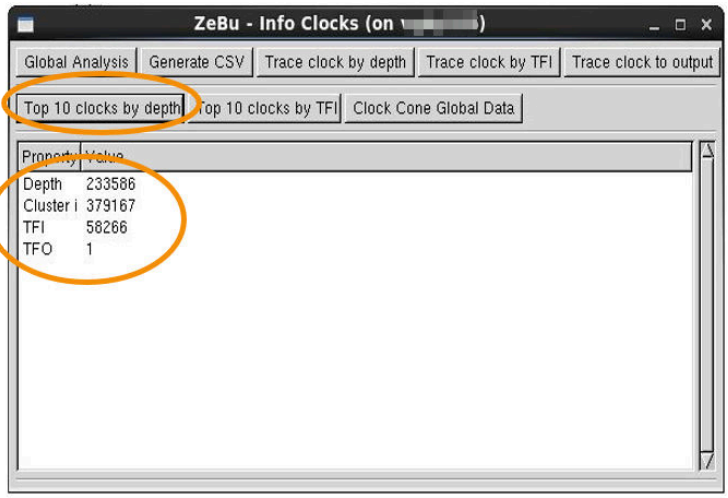
Finding Clocks with Maximum Depth

Finding the clocks with maximum depth helps to identify the following:

- A clock that consumes major resources to generate the database
- A clock that incorrectly pulls in some logic due to incorrect configuration
- Clock localization resource overflow that might cause Place & Route failure

To find the deepest clock, click **Top 10 clocks by depth** and the following pane is displayed:

Figure 48 Top 10 Clocks by Depth



Each row is clickable and displays the clock name, depth and TFI. When the row is clicked, the context of **zBrowser** changes accordingly. The attributes of a clock are also seen on the **Clocks** panel.

Figure 49 Clock Signals

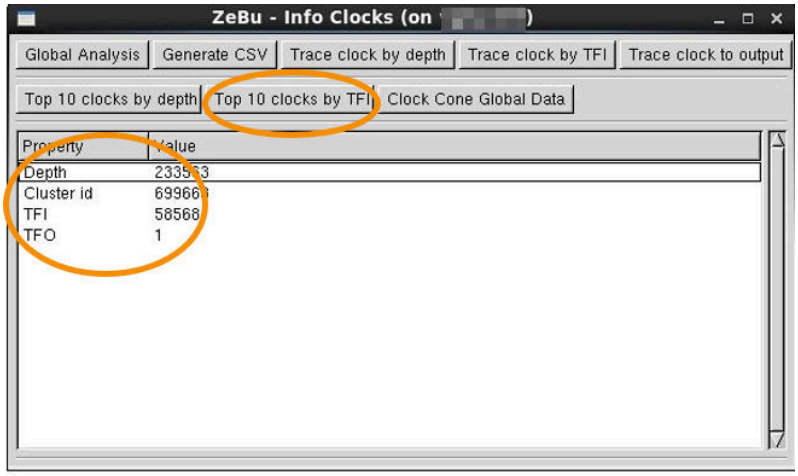
Finding Clocks with Maximum TFI

Clocks having a large TFI count can cause an unacceptable delta IO-cut at partition boundary. Therefore, it is important to find clocks that caused localization to fail.

To find the clocks with maximum IO-cut contribution, click **Top 10 clocks by TFI**.

Each row is clickable and displays the clock name, depth and TFI. When the row is clicked, the context of **zBrowser** changes accordingly. The attributes of a clock are also seen on the **Clocks Panel**.

Figure 50 Clocks by TFI



Tracing a Known Clock Signal

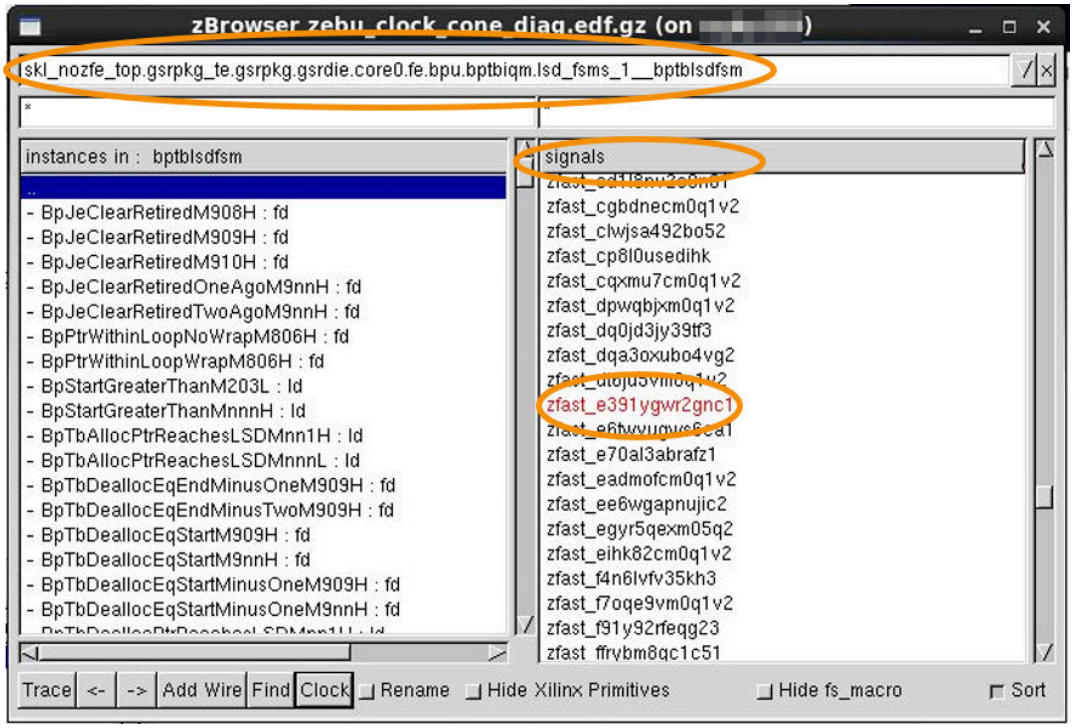
If the clock signal of interest is identified, click the clock signal of interest to change the **zBrowser** context.

Note:

The clock signal of interest is available at Top 10 clocks by Depth/TFI pane.

Feedback

1. To trace the clock signal of interest, copy the clock signal and paste the signal in the **zBrowser** trace window as shown:



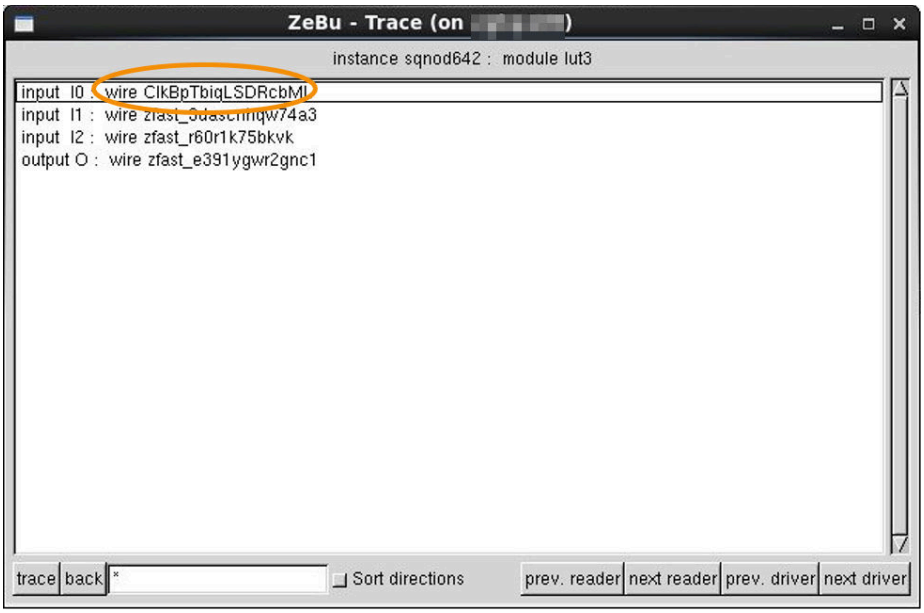
2. Click the “**ports/signals**” pane to “**signals**”.
3. Double-click the clock signal to open the trace window.

Example

The following figure shows the list of signals available within a module.

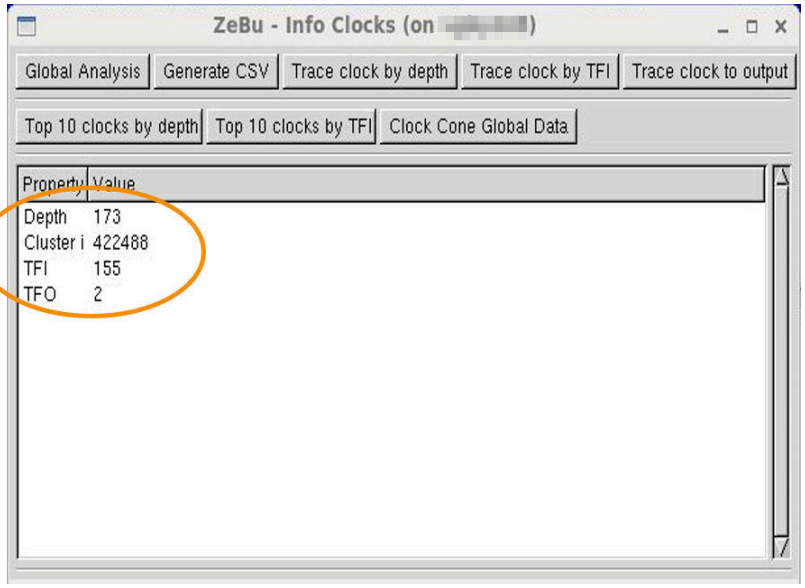
Feedback

Figure 51 Clock Signals in a Module



Double-click the signal/driving instance for which you want to view the attributes in the **Clocks** panel.

Figure 52 Clock Attributes



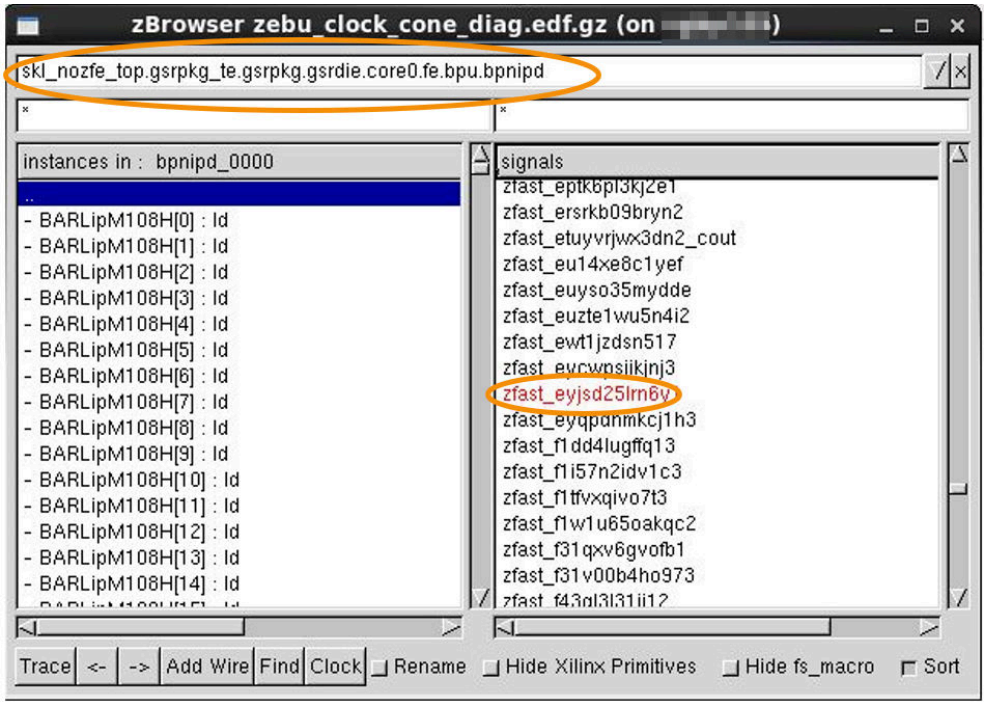
Feedback

Tracing the Clock Path Contributing to Maximum Logic Levels

To trace the clock path contributing to maximum logic levels, select any clock available on the **zBrowser** main window and then click **Trace clock by depth**.

The following figure shows an example signal.

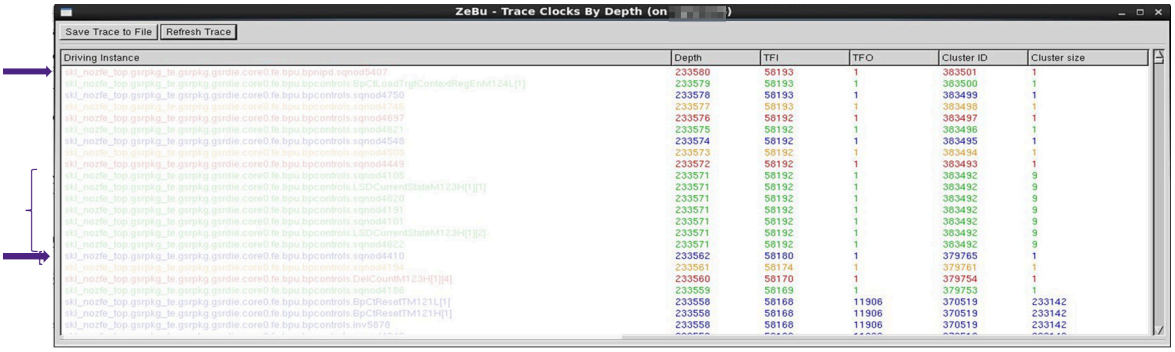
Figure 53 Clock Signal Contributing to Maximum Logic Level



Click **Refresh Trace** to find any changes.

A path with maximum logic levels to the primary input is traced and displayed as follows:

Figure 54 Maximum Logic Level Path



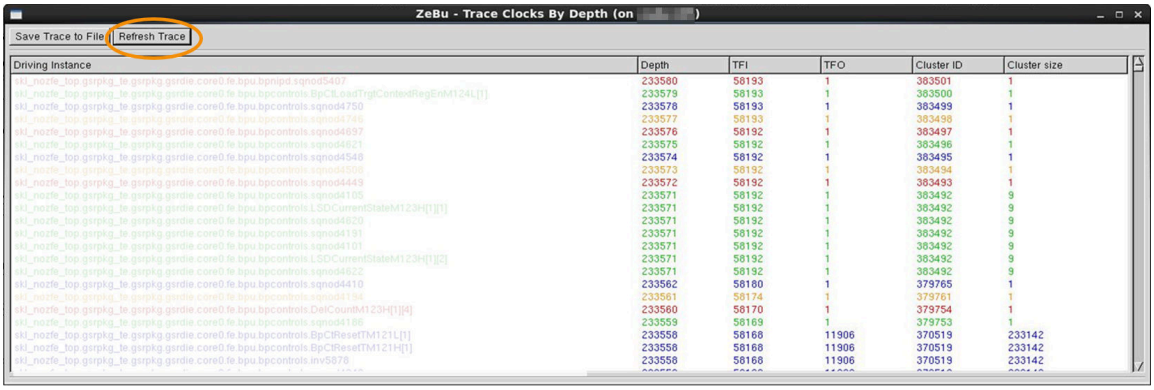
Feedback

In the **zBrowser** window, each entry shows a clock cone instance with its attributes (**depth, TFI, TFO, cluster ID, cluster size**). Consecutive rows with the same color implies that they belong to the same cluster. Change in color denotes a different cluster id. Each row is clickable and the **zBrowser** main window context changes while clicking the row.

Ties are broken arbitrarily if two driving instances contribute the same number of logic levels to the current instance.

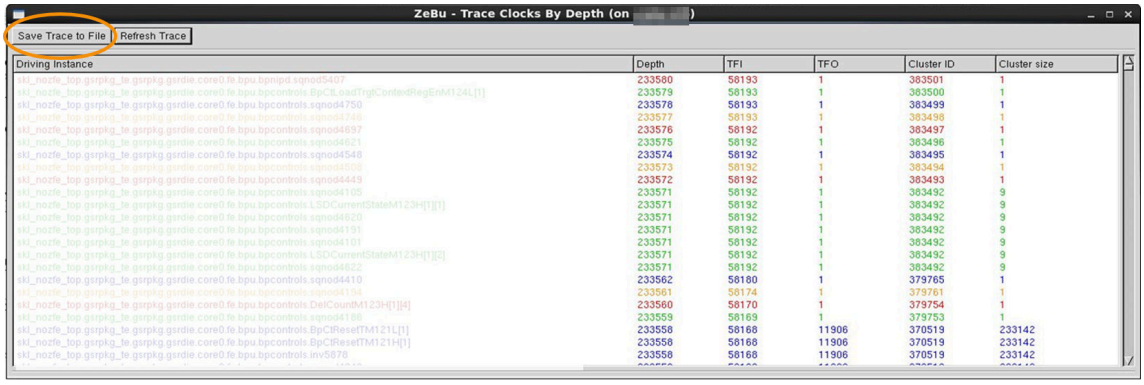
If you want to find a new trace from an intermediate point, select the point (row) and then click **Refresh Trace**.

Figure 55 Tracing an Intermediate Point



To save the trace path in a file, click **Save Trace to File**.

Figure 56 Saving Tracing to a File



Note:

This file helps you to find instances based on names, example: “*scan*”, “*dft*”, “*rst_sync*”, and so on.

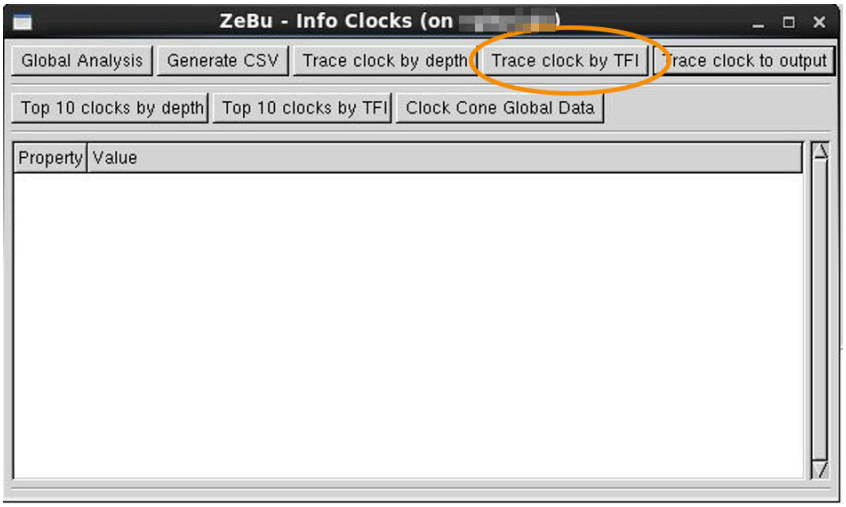
Feedback

Tracing the Clock Path Contributing to Maximum IO cut

To trace the clock path contributing to maximum IO cut, select any clock available on the **zBrowser** main window and then click **Trace clock by TFI**.

The following figure shows an example signal.

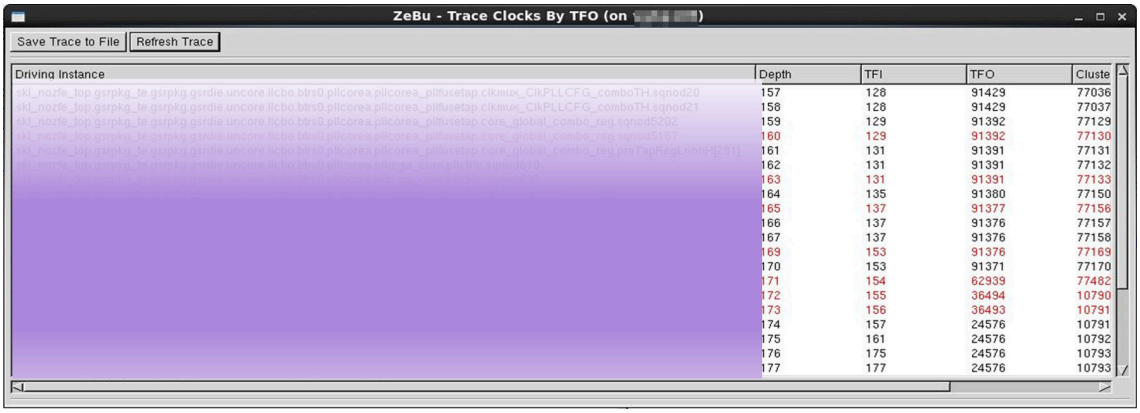
Figure 57 Tracing Clock Path Contributing to Maximum IO Cut



Tracing an Intermediate Clock Signal to its Final Output

To trace an intermediate clock cone signal to its final cone output(s), use **Trace clock to output** available in the **zBrowser** window.

Figure 58 Tracing Intermediate Clock Signal



Select the intermediate clock signal of interest from the **zBrowser** main window, and then click **Refresh Trace**.

The rows in red are the output clocks.

Analyzing Clock Localization

To find which clock cone inside a partition results in a highest IOCUT, a text report (`interface_rlc_debug`) is generated after clock localization in **zTopBuild** and **zCoreBuild**. The `interface_rlc_debug` file is available at `zebu.work`. The **zTopBuild** report is a single file generated for all zCores. The **zCoreBuild** report is generated for each zCore containing information about all FPGAs.

To enable this feature, set the following:

```
setenv ZEBU_RLC_DBG_PORT_FILE 1
```

To parse the interface RLC file in **zBrowser**, use the following:

```
zBrowser zebu_clock_cone_diag.edf.gz -clock -parse_interface_rlc_file
 interface_rlc_debug
```

Example

```
Partition: LOGIC
Clock: HWTop.Top.U_COUNTER.U_CLKMUX.Y  :  IOCUT=1034  REG=8   LUT=209
Clock: HWTop.Top.U_COUNTER.zebu_zbi_Counter  :  IOCUT=1  REG=0   LUT=0
```

In this example, the report shows the `U_CLKMUX.Y` clock instance in zCore “LOGIC” results in the highest IOCUT increase.

You can load the clock cone EDIF in **zBrowser** with the `-clock` option to trace which port(s) of this instance results in High IOCUT increase

Note:

The instance names seen in the `interface_rlc_debug` file might not necessarily match the ones in the clock cone EDIF because the netlist undergoes many intermediate changes.

Displaying CORE Level Mapping of Clock Instances

To parse the `defcoremap.tcl` file with the `-clock` option, use the following:

```
zBrowser  [ ]  -parse_defmap_file  zTopBuild_AC_defcoremap.tcl
```

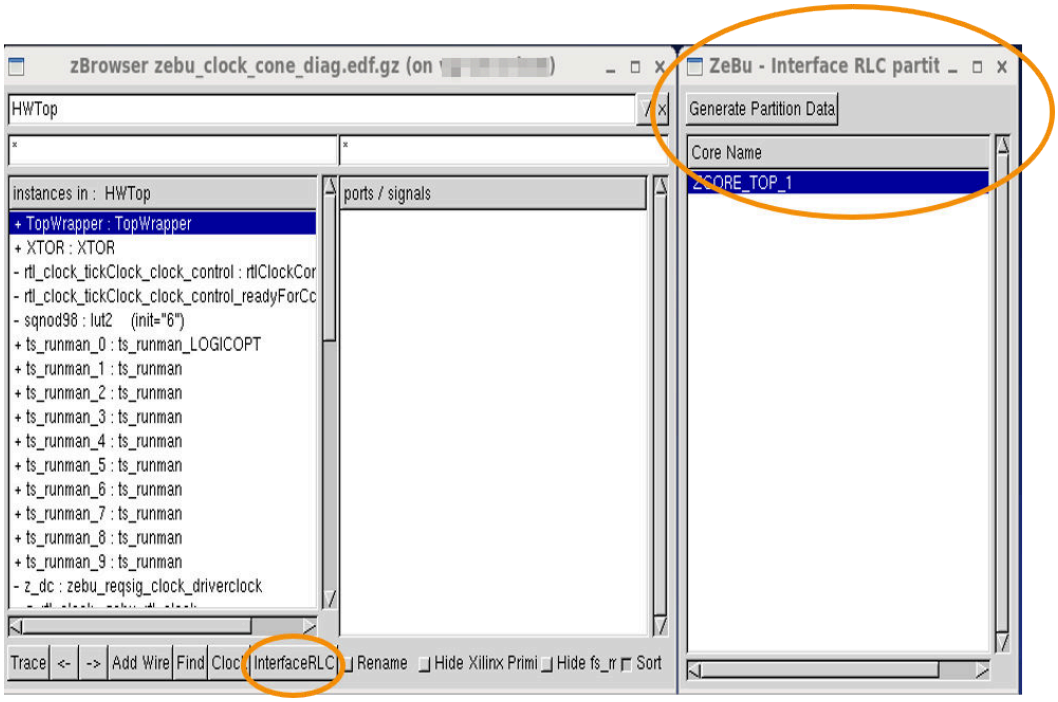
This option helps to study the extent of clock-cone across COREs. The **zBrowser** window refreshes after setting this option.

Feedback

Use Model

1. To generate the `interface-rlc-debug` file, set `ZEBU_RLC_DBG_PORT_FILE`.
2. Set `ZEBU_CLOCK_CONE_DIAG`.
3. Re-run **zTopBuild**.
4. zBrowser ... `-parse_interface_rlc_file interface_rlc_debug`
5. Click **Interface RLC** in the **zBrowser** window. The **ZeBu Interface RLC** Partition window opens.

Figure 59 Core Level Mapping of Clock Instances



11

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 consists of the following topics:

- designFeatures File
- Runtime Clock file
- Memory Content File
- Memory Content File Text Format

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:

This section has the following topics:

- Syntax Rules
- Location of the Calibration Files
- Declaring the Process Name
- Parameters for Design Clocks
- Parameters for Transactors

Feedback

- Initializing Memories
- Programming the driverClk Reset Signal
- Programming the DUT Reset
- Declaration for Smart Z-ICE
- Symbolic Parameters for Timing Settings
- Reducing Disk Usage With Clock Delay Feature

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 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`

Feedback

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>?;`

Declaring the Process Name

This section describes how to declare process names in a `designFeatures` file.

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";
```

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".
```

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".
```

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>";
```

Feedback

Parameters for Design Clocks

This section describes the parameters for design clocks in the `designFeatures` file.

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`.

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.

Feedback

```
$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";

# CLK1 overwrites the group setting to disable tolerance
$U0.M0.CLK1.VirtualFrequency = 5;
$U0.M0.CLK1.GroupName = "group1";
```

Feedback

```
$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:

```
"LUI1990E" : "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.

Feedback

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(*) }";
```

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).
```

Feedback

```
-- 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`.

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 the *ZeBu Getting Started 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.

Parameters for Transactors

This section describes the transactor parameters declared in the `designFeatures` file.

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.

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>;
```

Initializing Memories

This section describes how to initialize memories in the `designFeatures` file.

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;
```

Feedback

Note:

The automatic zrm transactional mode has been removed. Therefore, the value 3 is not supported anymore for the `zrmTransactionalMode designFeatures` parameter.

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;`

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>";
```

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.
- `<j>` is the index of the actual Smart Z-ICE connector at runtime.

Feedback

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.

Symbolic Parameters for Timing Settings

Symbolic parameters are available in the `designFeatures` file to compute the values of the timing parameters (`driverClock` frequency, Skew time and Filter time).

The calculation is based on the values computed by the post-FPGA timing analysis, except for the aggressive mode, which is based on the pre-FPGA timing analysis, as follows:

```
$timingSetting = "[default|safe|ultraSafe|aggressive]";
```

Depending on the mode used in `$timingSetting`, a ratio is applied to the values computed by the timing analysis to obtain the values that are used by the runtime.

An additional ratio can also be applied as follows:

```
$timingSettingRatio = <added_ratio>;
```

Note:

$timingSettingRatio also accepts float values.

The possible values for the `$timingSetting` parameter are as follows:

- `default`: The values computed by the post-FPGA timing analysis are used.
- `safe`: The values computed by the post-FPGA timing analysis are modified as follows:
  - The actual `driverClock` frequency is the value computed by the post-FPGA timing analysis divided by 4.
  - The actual Skew time and Filter time are the values computed by the post-FPGA timing analysis multiplied by 2.
- `ultraSafe`: The values are computed are as follows:
  - The actual `driverClock` frequency is the value computed by the post-FPGA timing analysis divided by 8.
  - The actual Skew time and Filter time are the values computed by the post-FPGA timing analysis multiplied by 4.
- `aggressive`: The values computed by the pre-FPGA timing analysis are used.

When `$timingSettingRatio` is used, the values computed in the manner described above are modified as follows:

- The `driverClock` frequency is multiplied by the value of `$timingSettingRatio`.
- Skew time and Filter time are divided by the value of `$timingSettingRatio`.

Note:

When Parallel data/clock propagation (`FetchMode`) is enabled, skew or filter time parameters are not relevant and only the `driverClock` frequency is computed.

The symbolic values are not compatible with the pre-existing parameters (`$driverClk.Frequency`, `$zClockSkewTime` and `$timingSettingRatio`). Only one set can be used in a given run.

All parameters in the `designFeatures` file are case insensitive.

Reducing Disk Usage With Clock Delay Feature

During runtime, in case of offline post-processing of waveform formats like FWC, QiWC, DPI, and SVA, you can enable the software timeline generation flow. This flow generates `*.ztdb/generic_cycle_time_file` during the run, which can then be used in replay or offline conversion jobs to generate the required timeline in software. This helps

Feedback

reducing disk space usage. To enable this flow, add the following variable setting in `designFeatures` file:

```
$clockDelayDumpTimestampInMainRun = 2;
```

In case of online post-processing of waveform formats, emulation jobs generate a disk consuming timestamp file (name pattern is `timestamp_cycle*`). However, if the jobs are not a part of the regression test suite, then disk usage is high, often full.

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.

Clock File Content

A separate clock file accepts the same parameters as the clock parameters of the `designFeatures` file (see Parameters for Design Clocks).

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_fequency '\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
        | in-situ
Tolerance_declaration :
        '$'GroupName.Tolerance '=' '"'tolerance_specifier'"' ';'
        | '$'CLOCK_NAME.Tolerance '=' '"'tolerance_specifier'"' ';'
tolerance_specifier :
        yes
        | no
delayClkEdge_<x> :
        "NbDriverClk" NUMBER "FROM" '{' list_of_edges '}'
                              "TO" '{' list_of_edges '}' ';'

list_of_edges :
        edge_declaration
        | list_of_edges, edge_declaration

edge_declaration :
        '"'edge_specifier'"' '(' '"'clock_identifier'"' ')'
        | '"'edge_specifier'"' '(' '*' ')'

edge_specifier :
        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]`

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).

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.

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 **zRci** 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.

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.

Feedback

- 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.

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
```

Feedback

```
@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

Address	Value
FF04	0044