

**Verification Continuum™**

## **ZeBu® User Guide**

---

**Version S-2021.09-2, October 2023**

**SYNOPSYS®**

## **Copyright Notice and Proprietary Information**

©2023 Synopsys, Inc. All rights reserved. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

## **Destination Control Statement**

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

## **Disclaimer**

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

## **Trademarks**

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

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

## **Free and Open-Source Software Licensing Notices**

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

## **Third-Party Links**

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

[www.synopsys.com](http://www.synopsys.com)

# Contents

---

<b>Preface.....</b>	<b>13</b>
About This Book .....	13
Contents of This Book .....	13
Related Documentation .....	15
Synopsys Statement on Inclusivity and Diversity .....	16
<b>1. Emulator System .....</b>	<b>17</b>
<b>1.1. Overview Of Emulators.....</b>	<b>18</b>
<b>1.2. ZeBu Emulation Flow.....</b>	<b>23</b>
<b>1.3. Introduction to zCui .....</b>	<b>24</b>
<b>2. Assembling a Design for Emulation .....</b>	<b>25</b>
<b>2.1. Mapping Your Design and Test Environment .....</b>	<b>26</b>
2.1.1. Importing a Design From a Simulation Environment .....	26
2.1.2. Preparing a DUT-Only Environment .....	27
<b>2.2. Clock Modeling .....</b>	<b>27</b>
<b>2.3. Reset Modeling.....</b>	<b>28</b>
<b>2.4. Memory Modeling .....</b>	<b>28</b>
2.4.1. Inference From a Design .....	28
2.4.2. Memory Model IP .....	32
<b>2.5. Designating Signals for Access During Runtime .....</b>	<b>33</b>
<b>2.6. Verilog System Tasks .....</b>	<b>34</b>
2.6.1. \$memset() System Task.....	40
2.6.2. \$plusargs System Task.....	41
2.6.3. \$fopen and \$fclose System Tasks .....	43
2.6.4. \$finish System Task .....	45
<b>2.7. Verilog Procedural Interface .....</b>	<b>46</b>
<b>3. Compilation.....</b>	<b>49</b>
<b>3.1. Overall Unified Compile Flow.....</b>	<b>50</b>
3.1.1. Introduction to UTF .....	51
3.1.2. Basic UTF Commands.....	51
<b>3.2. Synthesis .....</b>	<b>52</b>

3.2.1. Synthesis Log Files .....	52
<b>3.3. Compilation Script for VCS (UUM) .....</b>	<b>54</b>
3.3.1. Setting Up VCS .....	54
3.3.2. Setting Up the synopsys_sim.setup File .....	54
3.3.3. Setting Up Compilation Commands Using UTF .....	55
3.3.4. Compilation Settings for DesignWare Blocks .....	56
<b>3.4. Running a Design in Lint Mode .....</b>	<b>57</b>
<b>3.5. Specifying the Hardware Architecture for Compilation .....</b>	<b>58</b>
<b>3.6. Flow Selection During Compilation .....</b>	<b>58</b>
<b>3.7. Grid/LSF Handy Commands.....</b>	<b>58</b>
<b>3.8. Gate-Level Netlist Compilation .....</b>	<b>61</b>
<b>3.9. Handling Design Clocks Using Fetch Mode.....</b>	<b>62</b>
3.9.1. Limitations .....	62
<b>3.10. Removing Strongly Connected Components (SCC).....</b>	<b>63</b>
3.10.1. Use Model - Compilation .....	63
3.10.2. Use Model - Runtime Debug .....	64
3.10.3. Runtime Relaxation for Full Loop Breaking .....	66
3.10.4. Limitations of the Removing SCC Feature .....	66
<b>3.11. Achieving Compilation Objectives.....</b>	<b>66</b>
<b>3.12. Compiling a Project Using zCui .....</b>	<b>67</b>
3.12.1. Compiling With zCui in the Batch Mode .....	67
3.12.2. Compiling With zCui in the GUI Mode.....	68
3.12.3. Creating Runtime Archives After Compilation .....	71
3.12.4. Verdi Database Tasks in zCui .....	73
<b>3.13. Important ZeBu Log Files .....</b>	<b>75</b>
3.13.1. Analyzing Compilation Results Using zBatchExplorer .....	75
3.13.2. Viewing Timing Information in the zTime Reports .....	77
3.13.3. Native Compilation Profiler .....	80
3.13.4. Multi-Process Flow .....	84
3.13.5. Reporting XMR Information in a Design.....	87
3.13.6. Viewing Consolidated Information Using zAudit .....	89
3.13.7. Visualizing Compile-Time Data .....	107
<b>3.14. Handling Misfiring SVAs with Latches and Memories in fanin .....</b>	<b>111</b>
<b>3.15. Replicating Multiport Memories in zMem .....</b>	<b>113</b>
3.15.1. Usage .....	113
3.15.2. Reporting High Fanout Nets .....	113
3.15.3. Limitations.....	115
<b>3.16. Enhanced Verilog force/release.....</b>	<b>115</b>
3.16.1. Reporting Verilog force/release Information .....	115

3.16.2. Limitations.....	117
<b>3.17. Enhanced Multi-Gigabit Transceiver Support.....</b>	<b>119</b>
<b>3.18. Multi-Bin Based FPGA Complexity .....</b>	<b>119</b>
3.18.1. Enabling Multi-Binning.....	120
<b>3.19. Performance Driven Multi-die Multiplexing Support .....</b>	<b>120</b>
3.19.1. Enabling PDM Support.....	120
3.19.2. zCui Flow.....	121
<b>4. Clock Modeling in ZeBu .....</b>	<b>123</b>
<b>4.1. Types of Clock Supporting Emulation .....</b>	<b>123</b>
<b>4.2. Cycle-Based Clock Control.....</b>	<b>124</b>
<b>4.3. Time-Based Clock Control.....</b>	<b>126</b>
4.3.1. ClockDelayPort.....	126
4.3.2. RTL Delays .....	134
4.3.3. Waveform Dumping for the Clock Delay Feature .....	139
4.3.4. Reducing Disk Usage .....	139
4.3.5. Timescale and Precision .....	140
4.3.6. Compilation Results .....	140
4.3.7. Timestamp Clock.....	141
4.3.8. Global Time .....	141
4.3.9. Limitations of the Clock Delay Feature.....	143
<b>4.4. C_COSIM and Vertical Solution Transactors .....</b>	<b>144</b>
<b>4.5. zDPI and ZEMI-3 .....</b>	<b>145</b>
<b>5. Runtime .....</b>	<b>147</b>
<b>5.1. Controlling Runtime Parameters .....</b>	<b>148</b>
5.1.1. Emulation Speed .....	148
5.1.2. Clock Definition .....	148
5.1.3. Memory Initialization .....	150
<b>5.2. zRci for Controlling Clocks and Managing Memory .....</b>	<b>151</b>
5.2.1. Controlling Clocks.....	151
5.2.2. Managing Memory .....	153
<b>5.3. Cycle-Based C/C++ Co-Simulation .....</b>	<b>154</b>
<b>5.4. Reading Signal Values During Runtime .....</b>	<b>157</b>
<b>5.5. Changing the Signal Values During Runtime .....</b>	<b>159</b>
5.5.1. Depositing a Value on a Signal .....	159
5.5.2. Injecting a Value on a Signal.....	160

5.5.3. Forcing a Value on a Signal .....	161
5.5.4. Releasing a Forced Signal .....	162
<b>6. RunManager.....</b>	<b>163</b>
<b>6.1. Enabling RunManager.....</b>	<b>164</b>
<b>6.2. RunManager APIs.....</b>	<b>164</b>
<b>7. Using SystemVerilog Assertions.....</b>	<b>167</b>
<b>7.1. SVA Compilation.....</b>	<b>167</b>
<b>7.2. Runtime Usage .....</b>	<b>168</b>
7.2.1. Controlling Assertions From the C++ Testbench .....	168
7.2.2. Controlling Assertions Using zRci .....	174
7.2.3. Runtime Options for Assertion Control Tasks.....	176
<b>7.3. Post Processing SVA.....</b>	<b>177</b>
<b>7.4. Supported SVA subset .....</b>	<b>178</b>
<b>8. ZeBu DPI Technologies .....</b>	<b>183</b>
<b>8.1. Supported DPI Function Calls .....</b>	<b>184</b>
8.1.1. Unified DPI .....	185
8.1.2. Unified DPI Versus zDPI.....	185
<b>8.2. Compilation .....</b>	<b>187</b>
8.2.1. zDPI (Default Technology) .....	187
8.2.2. Unified DPI .....	188
8.2.3. Compiling ZeBu DPI files .....	194
<b>8.3. Writing DPI Import Functions .....</b>	<b>194</b>
<b>8.4. ZeBu API to Control DPI Functions at Runtime.....</b>	<b>195</b>
8.4.1. C++ API to Control DPI Functions .....	196
8.4.2. zRci Tcl Commands to Control the DPI Functions .....	202
8.4.3. Runtime for Unified DPI.....	203
<b>8.5. Processing DPI Function Calls Offline .....</b>	<b>204</b>
8.5.1. Identifying the DPI Functions .....	205
8.5.2. Enabling the DPI Offline Feature During Emulation .....	205
8.5.3. Creating a Shared Library .....	211
8.5.4. Creating an Input File for the zdpiReport Tool .....	211
8.5.5. Using the zdpiReport Tool .....	211
<b>8.6. Optimization Guidelines .....</b>	<b>216</b>
8.6.1. Enhancing Data Transfer.....	216

8.6.2. Defining Sampling Clock Frequency.....	218
8.6.3. Investigating Emulation Slowdowns .....	218
<b>8.7. Limitations of the ZeBu DPI Technologies .....</b>	<b>220</b>
8.7.1. zDPI .....	220
8.7.2. Unified DPI .....	221
<b>8.8. Example of DPI Function Calls to Print Counter Value .....</b>	<b>221</b>
8.8.1. Design .....	221
8.8.2. Implementation of DPI Functions.....	225
8.8.3. Controlling DPI Calls from a C++ Testbench .....	226
8.8.4. Controlling the Calls from zRci.....	228
<b>9. Saving the Design State and Restarting from a Saved State..</b>	<b>231</b>
9.1. Save and Restore - Recommendations .....	232
9.2. C++ Method for Save and Restore.....	232
9.3. C++ Example .....	235
9.4. zRci Method for Save and Restore .....	235
9.5. zRci Example.....	236
<b>10. Clock Cone Visualization in zBrowser .....</b>	<b>237</b>
10.1. Enabling Clock Cone Visualization .....	239
10.2. Finding Clocks with Maximum Depth .....	239
10.3. Finding Clocks with Maximum TFI .....	240
10.4. Tracing a Known Clock Signal.....	241
10.5. Tracing the Clock Path Contributing to Maximum Logic Levels ....	244
10.6. Tracing the Clock Path Contributing to Maximum IO cut .....	246
10.7. Tracing an Intermediate Clock Signal to its Final Output.....	247
10.8. Analyzing Clock Localization .....	247
10.8.1. Displaying CORE Level Mapping of Clock Instances.....	248
10.8.2. Use Model .....	248
<b>11. Appendix .....</b>	<b>251</b>
<b>11.1. designFeatures File .....</b>	<b>252</b>
11.1.1. Syntax Rules.....	252
11.1.2. Location of the Calibration Files .....	253
11.1.3. Declaring the Process Name.....	253
11.1.4. Parameters for Design Clocks .....	256

11.1.5. Parameters for Transactors.....	265
11.1.6. Initializing Memories .....	266
11.1.7. Programming the driverClk Reset Signal.....	267
11.1.8. Programming the DUT Reset .....	267
11.1.9. Declaration for Smart Z-ICE .....	268
11.1.10. Symbolic Parameters for Timing Settings.....	269
11.1.11. Reducing Disk Usage With Clock Delay Feature .....	270
<b>11.2. Runtime Clock file .....</b>	<b>270</b>
11.2.1. Clock File Content .....	271
11.2.2. Clock File Syntax .....	271
<b>11.3. Memory Content File.....</b>	<b>273</b>
11.3.1. ZeBu-Proprietary Binary Format.....	273
11.3.2. Memory Binary Format for Shorter Load Time .....	273
11.3.3. Memory Content File Text Format.....	274

# List of Figures

---

<b>ZeBu Server 5 System .....</b>	<b>18</b>
<b>ZeBu Server 5 Architecture .....</b>	<b>19</b>
<b>ZeBu EP1 System .....</b>	<b>19</b>
<b>ZeBu EP1 Architecture With Up to 16 Modules .....</b>	<b>20</b>
<b>ZeBu EP1 Architecture With Up to 8 Modules .....</b>	<b>20</b>
<b>ZeBu Server 4 System .....</b>	<b>21</b>
<b>ZeBu Server 4 Architecture .....</b>	<b>21</b>
<b>ZeBu Emulation Flow.....</b>	<b>23</b>
<b>zCui Default View .....</b>	<b>24</b>
<b>Importing a DUT and Test Environment From a Simulation Environment</b>	<b>26</b>
<b>Preparing a DUT-Only Environment.....</b>	<b>27</b>
<b>ZeBu Unified Compile Flow .....</b>	<b>50</b>
<b>Comparison Between Original and Reduced SCCs.....</b>	<b>64</b>
<b>zCui Options.....</b>	<b>68</b>
<b>zCui GUI .....</b>	<b>69</b>
<b>Project Messages Panel.....</b>	<b>70</b>
<b>Launching Compilation in zCui .....</b>	<b>70</b>
<b>Viewing the VCS Task in zCui .....</b>	<b>71</b>
<b>Archive Menu in zCui.....</b>	<b>71</b>
<b>Runtime Files Export.....</b>	<b>72</b>
<b>Tasks to Job Queue Panel .....</b>	<b>73</b>
<b>Viewing the Verdi Task in zCui .....</b>	<b>74</b>
<b>Viewing the Verdi Task in zCui Log.....</b>	<b>74</b>
<b>Opening a Directory using zBatchExplorer .....</b>	<b>76</b>
<b>Selecting Log File and zCui Global File .....</b>	<b>76</b>
<b>zTime HTML Report: Header .....</b>	<b>78</b>
<b>Color Code in zTime Report .....</b>	<b>79</b>
<b>Alias Column in zTime Report.....</b>	<b>79</b>
<b>Native Compilation Profiler .....</b>	<b>81</b>

<b>Example of Compilation Summary .....</b>	<b>82</b>
<b>Example of Compile Time Table With Delay .....</b>	<b>83</b>
<b>Example of Compile Time Table Without Delay.....</b>	<b>84</b>
<b>Smart Job Scheduling for Complex FPGAs .....</b>	<b>85</b>
<b>Compilation details Tab .....</b>	<b>109</b>
<b>SLL Tab .....</b>	<b>110</b>
<b>Xdrs distribution Tab.....</b>	<b>111</b>
<b>PDM Flow in zCui.....</b>	<b>122</b>
<b>Instantiate the zceiClockPort clock generation macro.....</b>	<b>124</b>
<b>Clock Delay Primitive .....</b>	<b>132</b>
<b>Supported ZeBu DPI Technologies .....</b>	<b>183</b>
<b>Example 1: Analyzing of Import DPI Called in the Initial Block .....</b>	<b>190</b>
<b>Example 2: Analyzing of Import DPI Called in the Initial Block .....</b>	<b>190</b>
<b>Example 1: List of In and Out Ports of Import DPI Called in Initial Block</b>	<b>191</b>
<b>Example 2: List of In and Out Ports of System Task (\$plusargs) .....</b>	<b>194</b>
<b>Clock Panel in zBrowser .....</b>	<b>237</b>
<b>Attributes of Clock.....</b>	<b>238</b>
<b>Top 10 Clocks by Depth .....</b>	<b>239</b>
<b>Clock Signals .....</b>	<b>240</b>
<b>Clocks by TFI.....</b>	<b>241</b>
<b>Clock Signals in a Module .....</b>	<b>243</b>
<b>Clock Attributes.....</b>	<b>243</b>
<b>Clock Signal Contributing to Maximum Logic Level.....</b>	<b>244</b>
<b>Maximum Logic Level Path .....</b>	<b>245</b>
<b>Tracing an Intermediate Point.....</b>	<b>245</b>
<b>Saving Tracing to a File .....</b>	<b>246</b>
<b>Tracing Clock Path Contributing to Maximum IO Cut .....</b>	<b>246</b>
<b>Tracing Intermediate Clock Signal.....</b>	<b>247</b>
<b>Core Level Mapping of Clock Instances.....</b>	<b>249</b>

# List of Tables

---

<b>Supported Verilog System-Tasks.....</b>	<b>34</b>
<b>UTF Help Commands .....</b>	<b>51</b>
<b>Basic UTF Commands .....</b>	<b>51</b>
<b>Synthesis Logs .....</b>	<b>53</b>
<b>Sun Grid Engine and Platform LSF Recommendation.....</b>	<b>59</b>
<b>zAudit Commands .....</b>	<b>91</b>
<b>Types of Clocks .....</b>	<b>124</b>
<b>Using ClockDelayPort Macro.....</b>	<b>128</b>
<b>APIs Supported with RunManager.....</b>	<b>164</b>
<b>Supported SystemVerilog SVA Subset.....</b>	<b>178</b>
<b>Unified DPI Versus zDPI - Configuraion Setup .....</b>	<b>186</b>
<b>zdpiReport Mandatory Parameters .....</b>	<b>212</b>
<b>zdpiReport Options .....</b>	<b>212</b>
<b>Mapping Between SystemVerilog Types with C Types .....</b>	<b>217</b>



## 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...
<i>Emulator System</i>	Introduces ZeBu Server and its features.
<i>Assembling a Design for Emulation</i>	Describes how to prepare your design for emulation before compilation.
<i>Compilation</i>	Describes how to compile your design in ZeBu.
<i>Clock Modeling in ZeBu</i>	Describes how to use the Clock Delay feature to enable simulation style support of native clock generation in the design.
<i>Runtime</i>	Describes how to control ZeBu runtime.
<i>RunManager</i>	Describes how to control the emulation runtime to overcome the limitation of the C-COSIM and ZEBU::Clock object.
<i>Using SystemVerilog Assertions</i>	Describes SystemVerilog assertions for functional verification.
<i>ZeBu DPI Technologies</i>	Describes DPI imported function calls supported in ZeBu.

Chapter	Describes...
<i>Saving the Design State and Restarting from a Saved State</i>	Describes the save and restore feature that allows you to save and restore the state of the entire hardware DUT and the software testbench environment during the emulation runtime.
<i>Appendix</i>	Describes the input files for compilation and runtime operations specific to the ZeBu environment.

# Related Documentation

Document Name	Description
<i>ZeBu User Guide</i>	Provides detailed information on using ZeBu.
<i>ZeBu Debug Guide</i>	Provides information on tools you can use for debugging.
<i>ZeBu Debug Methodology Guide</i>	Provides debug methodologies that you can use for debugging.
<i>ZeBu Unified Command-Line User Guide</i>	Provides the usage of Unified Command-Line Interface (UCLI) for debugging your design.
<i>ZeBu UTF Reference Guide</i>	Describes Unified Tcl Format (UTF) commands used with ZeBu.
<i>ZeBu Power Aware Verification User Guide</i>	Describes how to use Power Aware verification in ZeBu environment, from the source files to runtime.
<i>ZeBu Functional Coverage User Guide</i>	Describes collecting functional coverage in emulation.
<i>Simulation Acceleration User Guide</i>	Provides information on how to use Simulation Acceleration to enable cosimulating SystemVerilog testbenches with the DUT
<i>ZeBu Verdi Integration Guide</i>	Provides Verdi features that you can use with ZeBu. This document is available in the Verdi documentation set.
<i>ZeBu Runtime Performance Analysis With zTune User Guide</i>	Provides information about runtime emulation performance analysis with zTune.
<i>ZeBu Custom DPI Based Transactors User Guide</i>	Describes ZEMI-3 that enables writing transactors for functional testing of a design.
<i>ZeBu LCA Features Guide</i>	Provides a list of Limited Customer Availability (LCA) features available with ZeBu.
<i>ZeBu Transactors Compilation Application Note</i>	Provides detailed steps to instantiate and compile a ZeBu transactor.
<i>ZeBu zManualPartitioner Application Note</i>	Describes the zManualPartitioner feature for ZeBu. It is a graphical interface to manually partition a design.
<i>ZeBu Hybrid Emulation Application Note</i>	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 describes the following topics:

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

## 1.1 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.

### **ZeBu Server 5**

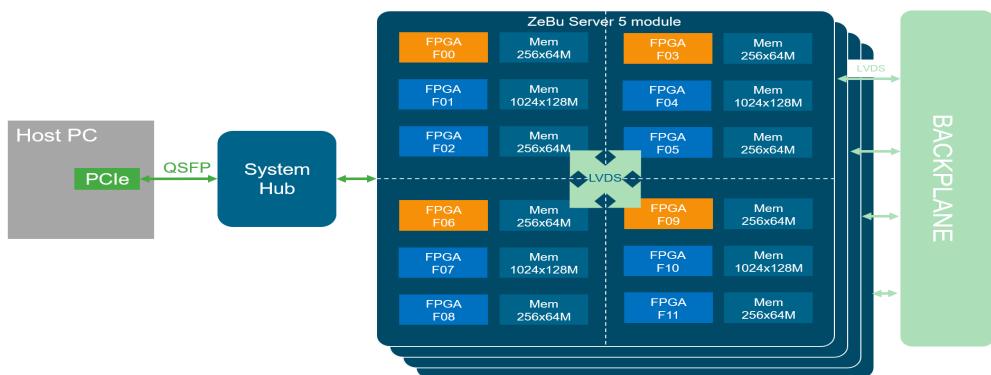
The following figure shows ZeBu Server 5:



**FIGURE 1.** ZeBu Server 5 System

The following figure displays the architecture of ZeBu Server 5.

## Overview Of Emulators

**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***.

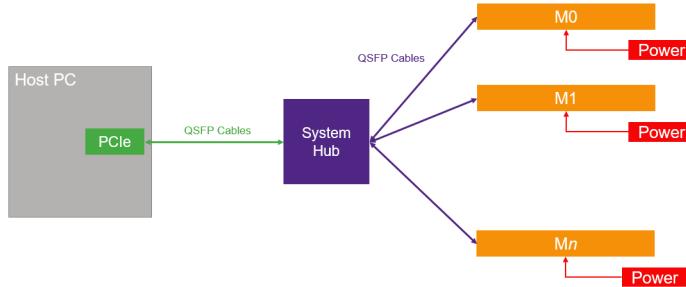
For detailed information about ZeBu Server 5 and software prerequisites for installing ZeBu, see ***ZeBu Server 5 Site Administration Guide***.

## ZeBu EP1

The following figure shows ZeBu EP1:

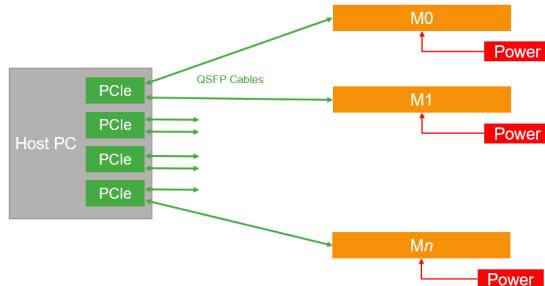
**FIGURE 3.** ZeBu EP1 System

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**.

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

---

Overview Of Emulators

## ZeBu Server 4

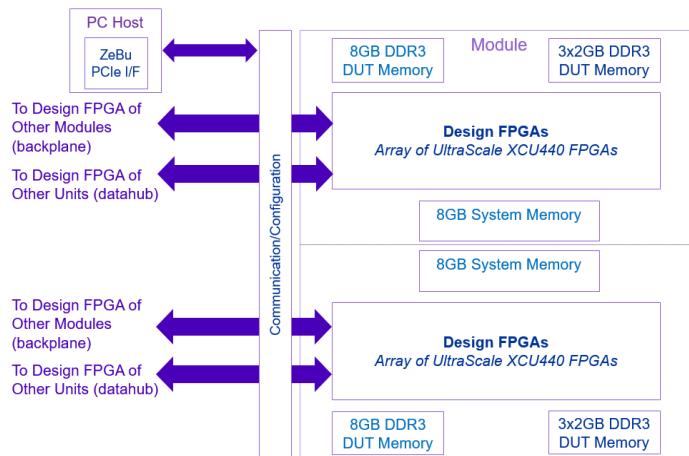
The following figure shows ZeBu Server 4:



**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***.

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

## ZeBu Emulation Flow

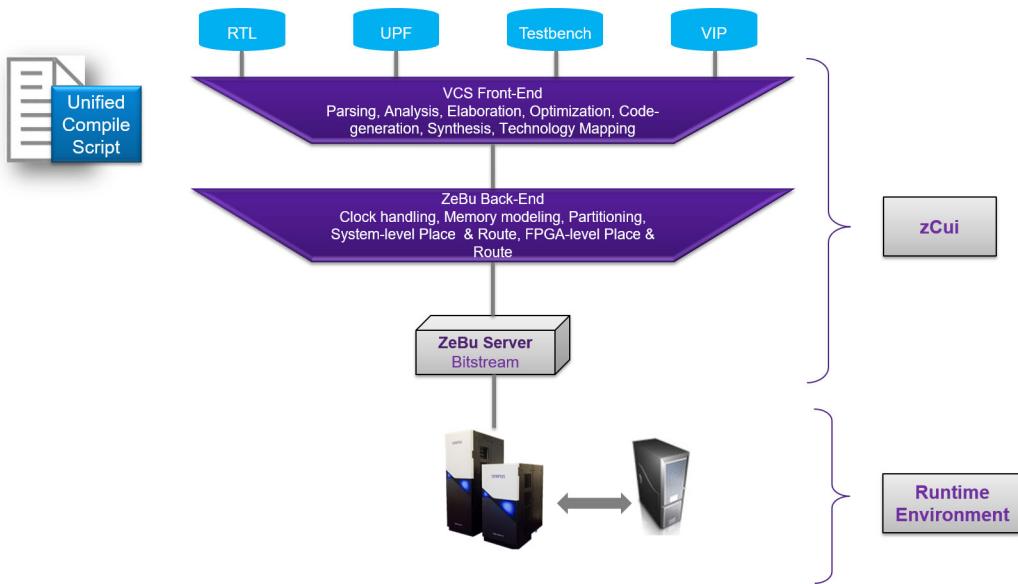
## 1.2 ZeBu Emulation Flow

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.



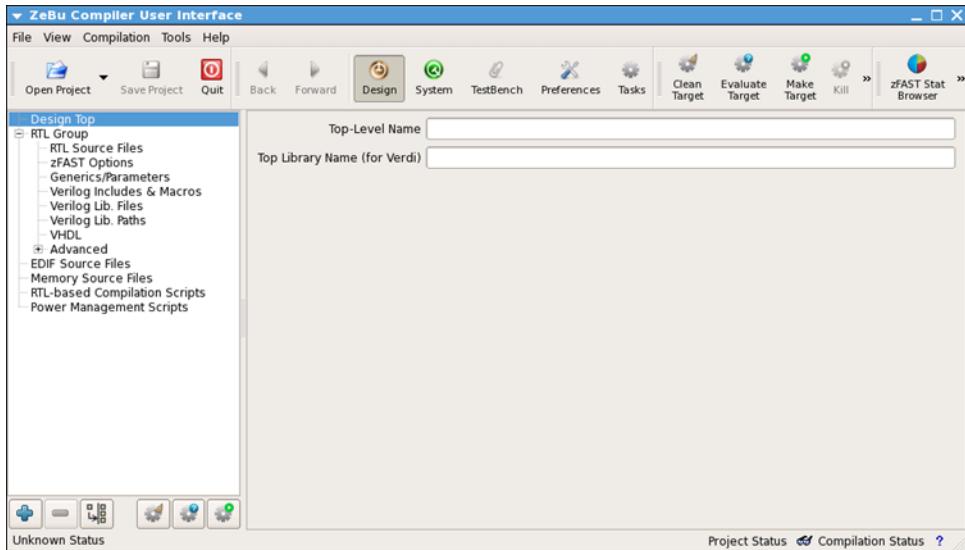
**FIGURE 8.** ZeBu Emulation Flow

The ZeBu emulation flow consists of the following steps:

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

## 1.3 Introduction to zCui

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



**FIGURE 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. This section discusses the following topics.

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

## 2.1 Mapping Your Design and Test Environment

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

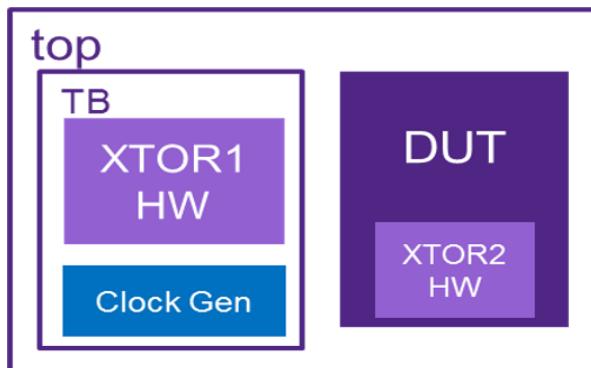
- When a simulation environment is available for your design, you can retain the same architecture with the existing top-level module. For more information, see [Importing a Design From a Simulation Environment](#).
- When the DUT and testbench are available as separate elements, you need to create an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

### 2.1.1 Importing a Design From a Simulation Environment

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

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

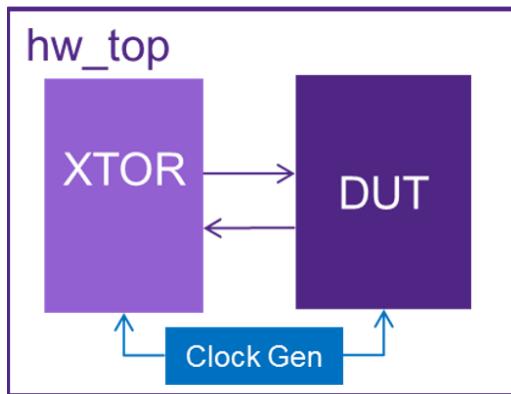
The following figure displays a clock generator and transactor instantiated in a top module.



**FIGURE 10.** Importing a DUT and Test Environment From a Simulation Environment

## 2.1.2 Preparing a DUT-Only Environment

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



**FIGURE 11.** Preparing a DUT-Only Environment

## 2.2 Clock Modeling

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

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

For example,

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

where,

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

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

## 2.3 Reset Modeling

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

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

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

- If you are using **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.*

## 2.4 Memory Modeling

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

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

Instantiate the available memory model Intellectual Property (IP).

This section consists of the following sub-sections:

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

### 2.4.1 Inference From a Design

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

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

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

#### 2.4.1.1 Guidelines for Performance

Memory performance depends on memory ports inference.

Following are some of the coding style guidelines and examples:

- Multi-ports and number of instances

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

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

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

- Read ports: Recommended coding style

- Continuously read ports

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

- Asynchronous read ports

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

The following are some of the user code transformation examples:

## Example 1:

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

### Original RTL

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

### Modified RTL

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

## Example 2:

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

## Original RTL

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

## Modified RTL

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

- Write ports: Recommended coding style
  - Asynchronous write ports

Avoid when possible as they may be slower.

**Read-modify-write ports**

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

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

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

## 2.4.2 Memory Model IP

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

- Complex memory models (for example, DDRx/GDDRx SDRAM and NAND/NOR Flash) are available as separate IPs with dedicated documentation.
- Ultra-large memory, which exceeds the memory capacity of ZeBu Server, can use the memory of the host PC through one of the dedicated transactors (for example, SRAMSW and ZLPDDR4\_Xtor).

---

## Designating Signals for Access During Runtime

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

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

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

## 2.5 Designating Signals for Access During Runtime

This section describes how signals can be accessed during runtime. See the following subsections for more information:

- *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 UFT command allows the testbench to access any signal at runtime:

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

---

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

### Forcing and Injecting Values

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

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

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

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

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

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

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

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

## 2.6 Verilog System Tasks

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

For information about ZEMI-3, see ***ZeBu Custom DPI Based Transactors User Guide***.

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

**TABLE 1** Supported Verilog System-Tasks

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

**TABLE 1** Supported Verilog System-Tasks

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

**TABLE 1** Supported Verilog System-Tasks

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

**TABLE 1** Supported Verilog System-Tasks

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

**TABLE 1** Supported Verilog System-Tasks

<b>Task</b>	<b>Supported in ZEMI-3</b>	<b>Supported in Design</b>	<b>Default</b>
\$sformat, \$sformatf	Yes	Yes (if Unified DPI enabled)	No
\$fflush	Yes	No	No
\$fstrobe, \$fstrobeb, \$fstrobeh, \$fstrobeo, \$fmonitor, \$fmonitorb, \$fmonitorh, \$fmonitoro, \$fscanf, \$fread, \$fseek, \$feof, \$fgetc, \$ungetc, \$fgets, \$sscanf, \$rewind, \$ftell, \$ferror	No	No	No
<b>Memory Load Tasks</b>			
\$readmemb, \$readmemh	Yes	Yes	Yes
<b>Memory Dump Tasks</b>			
\$writememb, \$writememh	No	No	-
\$memset	Yes	Yes	Yes
<b>Command Line Input</b>			
\$value\$plusargs (always block), \$test\$plusargs(always block)	No	Yes	No
<b>VCD Tasks</b>			
\$dumpvars, \$dumports See Note below.	Partial	Partial	-
\$dumpfile, \$dumpoff, \$dumpon, \$dumpall, \$dumplimit, \$dumpflash, \$dumpportsoff, \$dumpportson, \$dumpportsall, \$dumpportslimit, \$dumpportsflush	No	No	-

**Note**

*When specifying \$dumpvars, the design object name can be put under double quotes. For example:*

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

*If the design object name is specified with double quotes, a specialized algorithm is used to match design objects in \$dumpvars. This improves 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);
```

If the synthesis of these tasks is enabled by default, use the following UFT commands to enable unified DPI:

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

To enable the synthesis of these system tasks, use the following UFT command:

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

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

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

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

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

For details, see [ZeBu DPI Technologies](#).

## 2.6.1 \$memset() System Task

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

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

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

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

**Note**

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

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

## 2.6.2 \$plusargs System Task

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

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

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

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

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

### 2.6.2.1 Example

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

The format of plusargs\_file is as follows:

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

The strings in plusargs\_file are separated by a space.

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

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

### 2.6.2.2 Limitations of \$plusargs

- Second argument of \$value\$plusargs must be an int/integer. Other types are currently not supported.

#### Example

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

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

## Example

```
$test$plusargs("enable_diag") // supported case
string en = "enable_diag";
$test$plusargs(en) //unsupported case

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

---

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

### 2.6.3 \$fopen and \$fclose System Tasks

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

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

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

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

---

### 2.6.3.1 Example

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

### 2.6.3.2 Limitations of \$fopen and \$fclose tasks

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

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

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

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

## 2.6.4 \$finish System Task

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

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

- Calls to `$finish` must appear only in the DUT.
- `$finish` is supported 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.*

### 2.6.4.1 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

### 2.6.4.2 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  
  
start_zebu rundir  
...
```

---

## 2.7 Verilog Procedural Interface

vpi\_get\_time() is supported with clocks as follows:

- When clock delay support is used, it returns the simulation time at the precision derived from settings for the design clocks.

---

Verilog Procedural Interface

- When ZCEI clocks are used, it returns the tick count (every posedge of driverClk on which a design clock changes value).



# 3 Compilation

---

This chapter describes how to compile your design in ZeBu.

It discusses the following topics:

- *Overall Unified Compile Flow*
- *Synthesis*
- *Compilation Script for VCS (UUM)*
- *Running a Design in Lint Mode*
- *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*
- *Handling Misfiring SVAs with Latches and Memories in fanin*
- *Replicating Multiport Memories in zMem*
- *Enhanced Verilog force/release*
- *Enhanced Multi-Gigabit Transceiver Support*
- *Multi-Bin Based FPGA Complexity*

## 3.1 Overall Unified Compile Flow

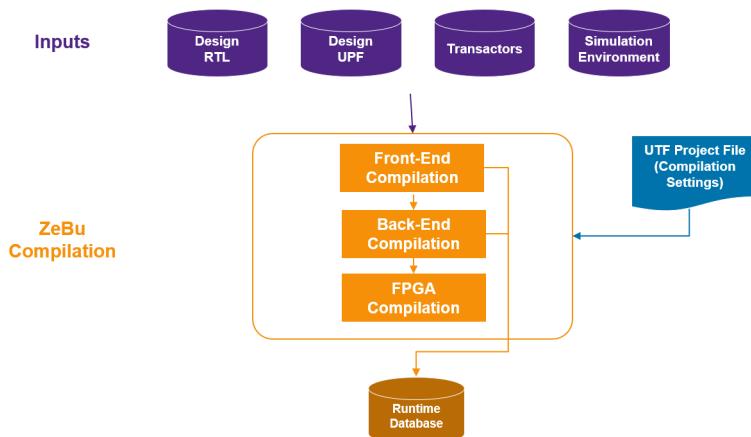
With the Unified Compile flow, the design and its test environment are compiled together by the VCS™ compiler, which is directed by a ZeBu project file written in Unified Tcl Format (UTF) .

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*

### 3.1.1 Introduction to UTF

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

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

The following table describes the UTF command categories.

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

**TABLE 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.

### 3.1.2 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.

**TABLE 3** Basic UFT Commands

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

## 3.2 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.

### 3.2.1 Synthesis Log Files

The following table lists the various logs created by synthesis.

## 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

## 3.3 Compilation Script for VCS (UUM)

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

- [\*Setting Up VCS\*](#)
- [\*Setting Up the synopsys\\_sim.setup File\*](#)
- [\*Setting Up Compilation Commands Using UUF\*](#)
- [\*Compilation Settings for DesignWare Blocks\*](#)

### 3.3.1 Setting Up VCS

Perform the following steps to set up VCS:

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

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

### 3.3.2 Setting Up the `synopsys_sim.setup` File

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

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

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

### 3.3.3 Setting Up Compilation Commands Using UUF

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

The script should consist of the following two commands:

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

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

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

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

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

where,

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

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

Note that you can ignore the RTL code (Verilog, SystemVerilog, or VHDL) specified between the `translate_on`/`translate_off` pragmas when doing synthesis for emulation. To do so, use the switch, `-skip_translate_body`, with any of the following VCS commands: `vlogan`, `vhdlan` or `vcs`.

Note that this switch is not enabled by default.

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

```
% vcs -full164 -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**.

### 3.3.4 Compilation Settings for DesignWare Blocks

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

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

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

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

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

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

## 3.4 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.

## 3.5 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}
```

## 3.6 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
```

## 3.7 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 5** Sun Grid Engine and Platform LSF Recommendation

<b>Sun Grid Engine/Platform</b>	<b>Typical Remote Command</b>	<b>Typical Kill Command</b>
Sun Grid Engine	qrsh	qdel
Platform LSF with blocking jobs and a non-interactive queue	bsub -K	bkill
Platform LSF with blocking jobs and an interactive queue	bsub -Is	bkill

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

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

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

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

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

The <given\_queue\_name> can be:

- Zebu
- ZebuAlternateSynthesis
- ZebuHeavy
- ZebuIse
- ZebuLight
- ZebuRelaunchedIse
- ZebuRelaunchedIseLevel2

- ZebuSuperHeavy
- ZebuSynthesis

For example,

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

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

---

Typical <submit\_command> with qrsh is as follows:

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

---

## 3.8 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`.

## 3.9 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.

### 3.9.1 Limitations

The following limitations exist:

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

## 3.10 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\*](#)

### 3.10.1 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:

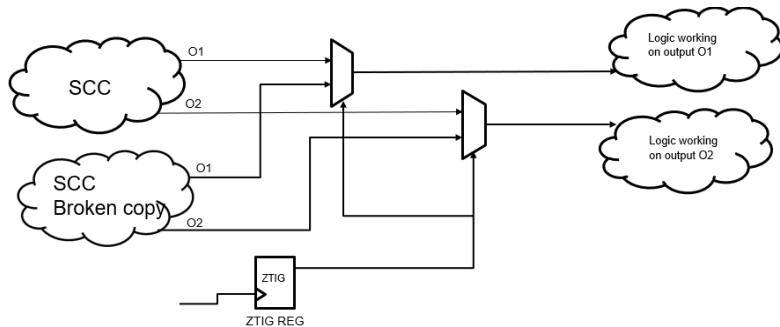
```
ztopbuild -advanced_command {\  
    loop break -full_break -max_lut_overflow 5 -max_reg_overflow 5 \  
    -rtl=yes}
```

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.

### 3.10.2 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:

## Removing Strongly Connected Components (SCC)

- 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:

- 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
```

- Use the following commands `zSelectProbes` to get ZRDB coordinates:

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

- To generate the ZTDB, use the following `zRci` script:

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

- Convert ZTDB to FSDB using the following command:

```
zConvertToFsdb --ztbd dump.ztbd --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.

### 3.10.3 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.

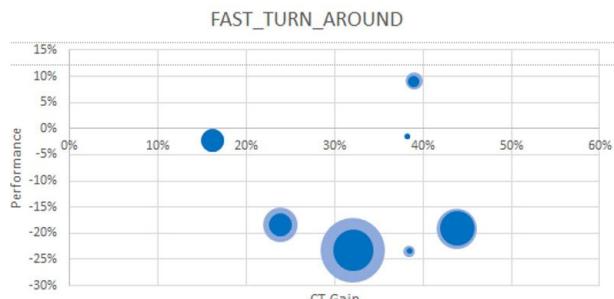
### 3.10.4 Limitations of the Removing SCC Feature

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

## 3.11 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.



---

## Compiling a Project Using zCui

### 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.

## 3.12 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](#)
- [Creating Runtime Archives After Compilation](#)
- [Verdi Database Tasks in zCui](#)

### 3.12.1 Compiling With zCui in the Batch Mode

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

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

---

where,

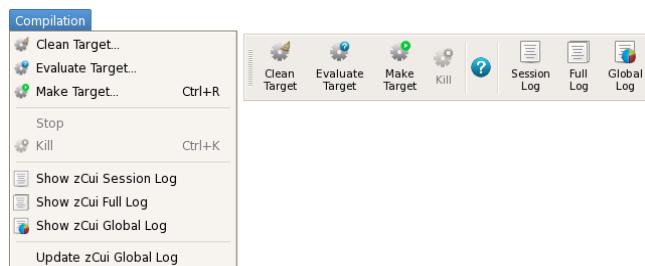
- **-c** launches compilation.
- **-n** runs **zCui** in the batch mode (no GUI).

- <project\_name>.utf is the existing UTF project file containing information to compile a design for ZeBu using Unified Compile flow.
- <zcu\_uc\_work\_dir> is the optional working directory for compilation; by default, this is ./zcu.work.

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

### 3.12.2 Compiling With **zCui** in the GUI Mode

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



**FIGURE 14.** zCui Options

The following operations are available in the GUI mode:

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

You can use the following command to launch **zCui** to monitor and manage the

## Compiling a Project Using zCui

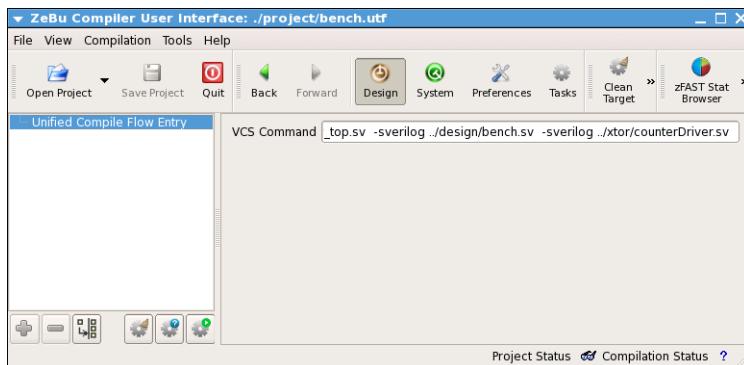
compile process with an existing UTF project file:

```
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.

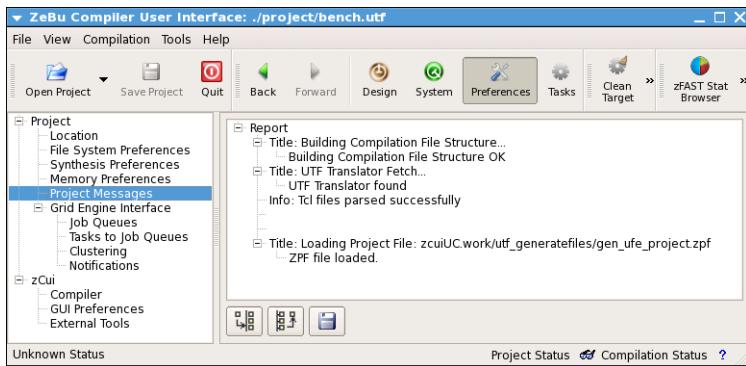
While using **zCui** to compile using a UTF file, it is not possible to save the compilation settings in the UTF file when they are modified in the GUI. Such modifications are only applicable for the next compilation within the GUI, for test purposes. Modification of the UTF file must be done in the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

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:



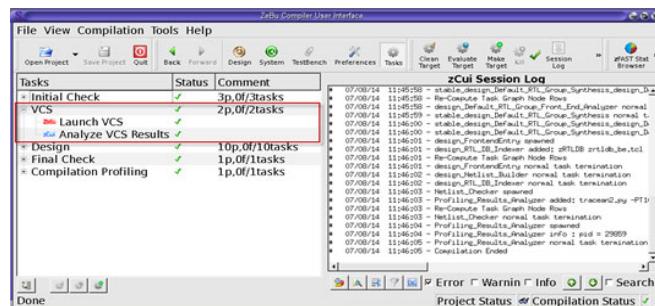
**FIGURE 17.** Launching Compilation in zCui

In the **Compilation Chooser** dialog (see *Figure 17*), you must select one of the following:

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

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

## Compiling a Project Using zCui

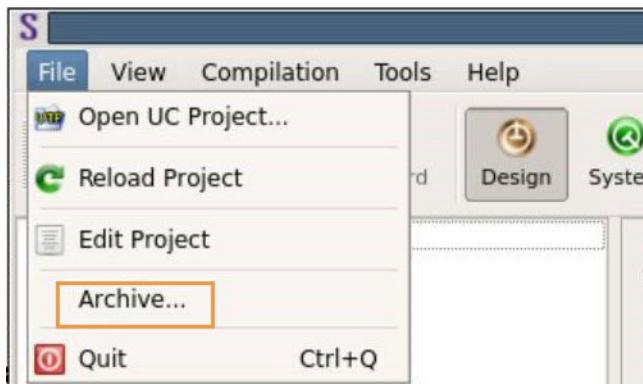


**FIGURE 18.** Viewing the VCS Task in zCui

### 3.12.3 Creating Runtime Archives After Compilation

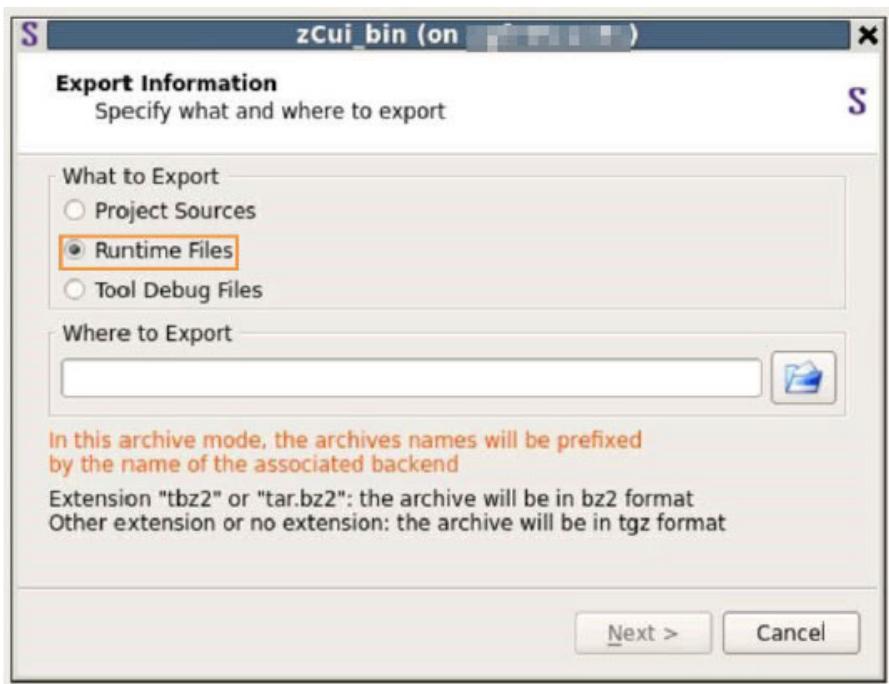
#### GUI Mode

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

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]
```

### 3.12.4 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}
```

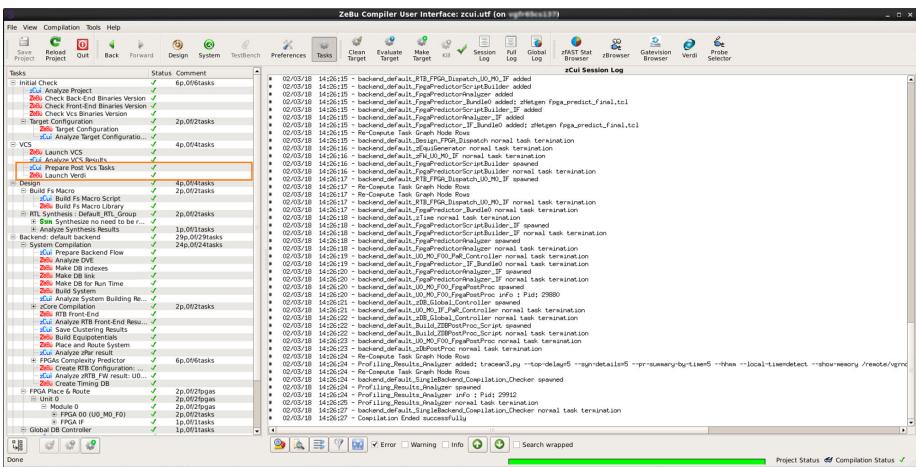
where, **newQueue** is another existing job queue or a new job queue created by  
`grid_cmd -queue {newQueue} -submit {...} -delete {...} -njobs ...`

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.

```
# 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_symp.tcl -log Bundle_0.log -zilog 1
# 27/02/18 09:00:27 - design_Default RTL_GroupBundle_0_Synthesis_Bundle_0_analyzer added
# 27/02/18 09:00:27 - Re-Compute Task Graph Node Rows
# 27/02/18 09:00:27 - VCS_Task_Analyzer normal task termination
# 27/02/18 09:00:28 - Verdi_Compilation spawned
# 27/02/18 09:00:28 - Verdi_Compilation info : Pid: 15261
# 27/02/18 09:00:28 - design_Default RTL_GroupBundle_0_Synthesis spawned
# 27/02/18 09:00:28 - design_Default RTL_GroupBundle_0_Synthesis info : Pid: 15262
# 27/02/18 09:00:30 - Verdi_Compilation normal task termination
# 27/02/18 09:00:33 - design_Default RTL_GroupBundle_0_Synthesis normal task termination
```

**FIGURE 23.** Viewing the Verdi Task in zCui Log

## 3.13 Important ZeBu Log Files

This section describes the following:

- [\*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\*](#)

### 3.13.1 Analyzing Compilation Results Using zBatchExplorer

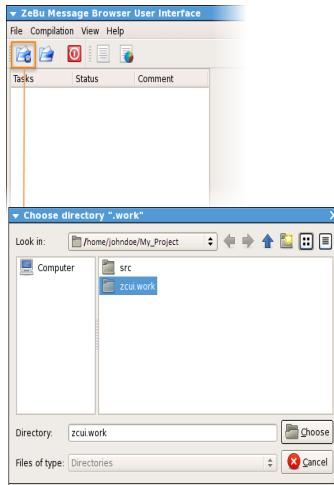
**zBatchExplorer** is a GUI that displays the task tree (similar to the **Tasks** workspace in **zCui**) after compilation. It analyzes log files of all the tasks, **zCui** full log, and global log.

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

```
$ zBatchExplorer
```

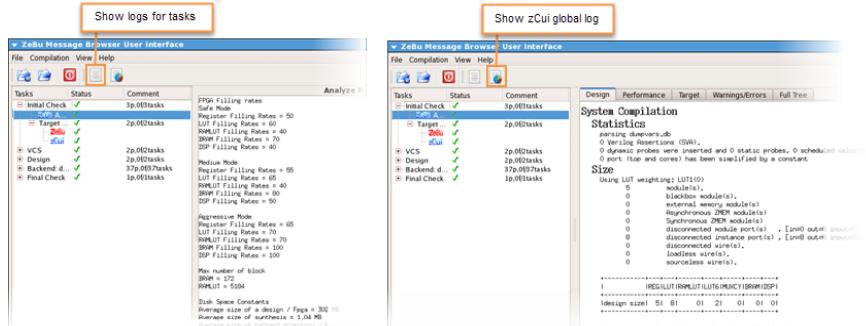
---

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



**FIGURE 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

## 3.13.2 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.

For more information, see the following subsections:

- [\*Accessing the zTime Reports\*](#)
- [\*Viewing Timing Summary\*](#)

### 3.13.2.1 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 [\*Analyzing Compilation Results Using zBatchExplorer\*](#)).

### 3.13.2.2 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:

### **zTime Report: Critical paths**

Date:

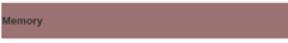
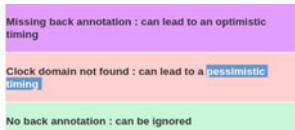
Version: S-2021.09-1

Slack	Required Time	Delay	Total Ptype	Logic Ptype	From
8.ns	267 ns ( 1 driver clock )	267 ns	1	1	U0_M0_F1.U0_M0_F1_6
219.ns	267 ns ( 1 driver clock )	48 ns	3	3	U0_M0_F1.P01_ts_clktx
919.ns	3467 ns	488 ns	16	16	U0_M0_F1.P01_ts_clktx

**FIGURE 26.** zTime HTML Report: Header

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)	
Color for paths with memories (in both reports)	
Specific colors for back-annotation in post-FPGA	

## Important ZeBu Log Files

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

Block #	Required now	Delay #	Fpga #	Clock domains now > 0	Paths from > 0	Sum of XORS #	Details							
							Fpga	Delay	Arrival	XDN	XTYPE	CR		
0 ns	144 ns (2 x 8ns clock)	144 ns	2	Clock_zhzbus0TB.clk_ipad_0 (grossdelay)	path0@zhzbus0TB.dut.ram_wb0.ram_wb_3d_0.mem[0] -> U0_M0_F1_U0_M0_F1_core.dut.or1200_top0.or1200_dc_top.or1200_dc_ipad_0.ram.mem[0]#H8D6E1D8D[7]	0							DutClock domain	
				Clock_zhzbus0TB.clk_ipad_0 (grossdelay)			U0_M0_F1_U0_M0_F1_core.dut.or1200_top0.or1200_dc_top.or1200_dc_ipad_0.ram.mem[0]#H8D6E1D8D[7]	44 ns	744 ns					U0_M0_F1_U0_M0_F1_core.dut.or1200_top0.or1200_dc_top.or1200_dc_ipad_0.ram.mem[0]#H8D6E1D8D[7]
				internal			U0_M0_F1_U0_M0_F1_core.dut.or1200_top0.or1200_dc_top.or1200_dc_ipad_0.ram.mem[0]#H8D6E1D8D[7]	0 ns	744 ns					U0_M0_F1_U0_M0_F1_core.dut.or1200_top0.or1200_dc_top.or1200_dc_ipad_0.ram.mem[0]#H8D6E1D8D[7]
				internal			U0_M0_F1_U0_M0_F1_core.dut.or1200_top0.or1200_dc_top.or1200_dc_ipad_0.ram.mem[0]#H8D6E1D8D[7]	744 ns						DutClock domain

**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.

Fpga	Delay	Arrival	XOR	XTYPE	FB	xCore	Port	Wire	Alias
Internal									
U0M0F075	1 ns	1 ns					Part_0	DriverClock domain	
								U0_M0_F5_U0_M0_F5_core.oipoc_tap_crossim.DutF0[0].in[3].Q (fde)	
								->zclockpt_1524319125415055743 ()	
U0M0F075	5 ns		1	LVDS					
U0M0F075	15 ns	21 ns							
U0M0F075	5 ns		1	LVDS					
U0M0F075	26 ns				Part_0			zclockpt_1524319125415055743 ()	U0_M0_F5_U0_M0_F5_core.dut.tag.tap0.sqen0.tms_reset.O (fde)
								->zclockpt_1524319125415055743 ()	
Internal	2 ns	28 ns					DutClock domain		

**FIGURE 28.** Alias Column in zTime Report

### 3.13.2.3 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

### 3.13.3 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.

#### Prerequisites

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

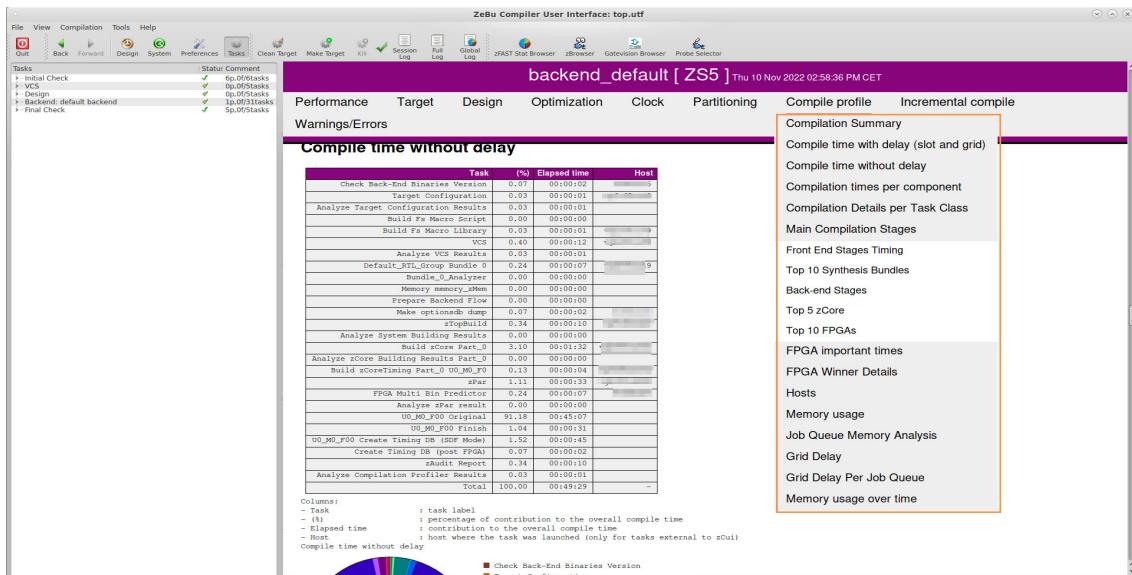
```
profile -compile true
```

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:

## Important ZeBu Log Files



**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*

### 3.13.3.1 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:

```

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 | Host       | Total # of jobs fired | Max memory consumption |
|------------------------------|--------|------------|------------|--------------|------------|-----------------------|------------------------|
| VCS                          | 13.04  | 00:00:00   | 00:06:08   | 00:33:11     | XXXXXXXXXX | 1                     | 5.42GB                 |
| zTopBuild                    | 1.66   | 00:00:00   | 00:00:13   | 00:04:14     | XXXXXXXXXX | 1                     | 17.82GB                |
| zCoreBuild                   | 8.39   | 00:00:01   | 00:40:28   | 00:21:21     | XXXXXXXXXX | 1                     | 19.09GB                |
| zCoreTiming                  | 0.36   | 00:00:02   | 00:08:55   | 00:00:55     | XXXXXXXXXX | 9                     | 10.73GB                |
| zPar                         | 0.40   | 00:00:00   | 00:00:14   | 00:01:01     | XXXXXXXXXX | 1                     | 3.96GB                 |
| FPGA Compile                 | 69.28  | 00:00:00   | 14:09:03   | 02:56:19     | XXXXXXXXXX | 12 Orig               | 31.56GB                |
| Create Timing DB (post FPGA) | 0.33   | 00:00:00   | 00:00:51   | 00:00:50     | XXXXXXXXXX | 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
- Host        : host where the task was launched (only for tasks external to zCui)
- Total # of jobs fired : number of jobs with same phase fired
- Max memory consumption : job in the phase with max memory taken

```

**FIGURE 30.** Example of Compilation Summary

### 3.13.3.2 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:

---

## Important ZeBu Log Files

Compile time with delay (slot and grid)

Task	(%)	Slot delay	Grid delay	Elapsed time	Spawn Time	Finish Time
Target Configuration	0.01	00:00:00	00:00:10	00:00:01	00:00:00	00:00:11
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
VCS	13.04	00:00:00	00:06:08	00:33:11	00:00:36	00:39:55
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
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
Make RTL DB link	0.02	00:00:00	00:00:12	00:00:03	02:33:33	02:33:48
zTopBuild	1.66	00:00:00	00:00:13	00:04:14	02:33:48	02:38:15
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
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
zPar	0.40	00:00:00	00:00:14	00:01:01	03:50:02	03:51:17
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
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
U0_M0_F07 Finish	0.20	00:00:00	00:00:09	00:00:31	21:15:23	21:16:03
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
Create Timing DB (post FPGA)	0.33	00:00:00	00:00:51	00:00:50	21:24:18	21:25:59
zAudit Report	0.05	00:00:00	00:00:01	00:00:07	21:26:00	21:26:08
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

**FIGURE 31.** Example of Compile Time Table With Delay

---

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

Compile time without delay

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

**FIGURE 32.** Example of Compile Time Table Without Delay

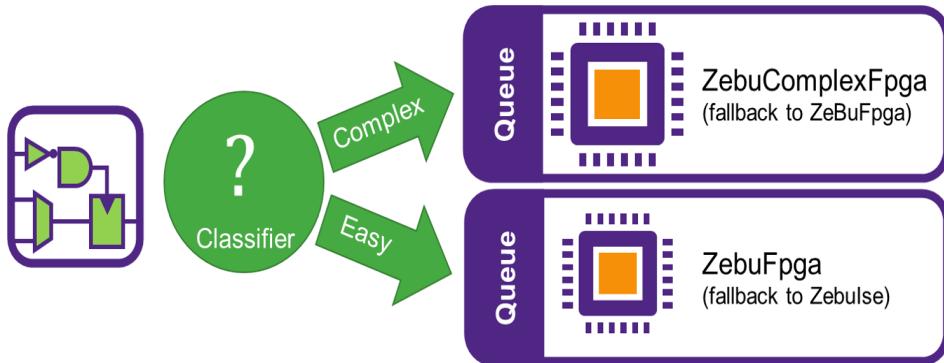
### 3.13.4 Multi-Process Flow

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.

## Important ZeBu Log Files

**FIGURE 33.** Smart Job Scheduling for Complex FPGAs

MPF feature requires Vivado 2020.1 or later versions.

To enable MPF, use the following UCF 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>}  
  
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\_MPFSUBMIT and VIVADO\_REMOTE\_MPFKILL 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_MPFSUBMIT <corresponding string for grid command>
```

- Multi-machine MPF remote termination

```
VIVADO_REMOTE_MPFKILL < 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.

---

## Important ZeBu Log Files

- ZebuFpga queue should be on a NORMAL queue so that it can use a slower or a faster host.

### 3.13.5 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 UFT command to your UFT 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
```

---

## Important ZeBu Log Files

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)

```

## 3.13.6 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 Examples\*](#)

### 3.13.6.1 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`

### 3.13.6.2 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
```

### 3.13.6.3 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 6** zAudit Commands

Compilation Analysis	<zAudit_command>
Global Compilation Summary	summary
Activated Technologies	technology
FPGA Metrics	fpga
Memory Costs	memory

**TABLE 6** zAudit Commands

<b>Compilation Analysis</b>	<b>&lt;zAudit_command&gt;</b>
Mapping Information	mapping
Information on zPar	zpar
Level of Combinational Logic	combinatory
Timing Statistics	timing
Information on User clocks	clock
Warnings & Recommendations	warning
Timing Bottlenecks	bottleneck
Improvement Suggestions	diagnostic
Displays all starred reports	report

### 3.13.6.4 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 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
```

## Important ZeBu Log Files

Summary	
Category	Details
General	Platform: ZS4, ZeBu Release: U-2023.12-Alpha, Vivado Release: 2021.1 AR76730
Memories	Total: 40, By Type: {BRAM: 40}, With # Ports: {1-3: 40, 4-7: 0, 8-15: 0, 16-31: 0, 32+: 0}
Clocks	Fetch Mode: yes, RTL Clocks: no, RLC Strategy: {zTopBuild: Not Enabled, zCores: {FULL: 0..3}}
Size	REG: 31,995,864, LUT: 29,473,171, LUT6: 12,515,060 (42% of LUT), BRAM: 46,088, RAMLUT: 0, DSP: 10,368, MUXCY: 0
Mapping	Units: 1, Modules: 4, FPGAs: 46 Avg Fill: {REG: 13%, LUT: 24%, BRAM: 38%} Max Board Fill: {REG: 14%, LUT: 26%, BRAM: 41%}, Max FPGA Fill: {REG: 16%, LUT: 42%, BRAM: 49%} Avg CUT: {Module: 157, FPGA: 158}, Max CUT: {Module: 315, FPGA: 300}
Routing	routerType: EZCR, routing Effort: min_distan, pinAssignmentEffort: zap, SocketMode: component, SAG: disabled
Timing	driverClk: 7142 kHz (140 ns) Pre-Post delta before rounding: {driverClk: 10000 -> 7494 (-25%)} bottleneck: routing data paths : 133ns, Longest Delays: {data: 133 ns, memory: 40 ns}, #MGT in top path: {data: 0, filter: 0} Max values in top 186 paths: {delay: 133, routing delay: 78, nodes: 6, logic fpgas: 4, routing fpgas: 3, modules: 3, units: 1, clkhubs: 0, mgts: 0}
FPGA	Total Num of FPGAs: 48 Original Winners: 48 PARFF Winners: 0 PARFF Winning FPGAs: None PARFF Winning FPGAs Strategy: {} Num of Failed FPGAs: 0, Failed FPGAs: [] Num FPGA compile ongoing: 0, FPGA compile ongoing: [] Long Pole FPGA Compile Times: {U0_M0_F03: 259.933, U0_M2_F02: 253.933, U0_M0_F06: 250.9, U0_M2_F07: 249.933, U0_M3_F05: 248.933} FPGA compile status: PASS
Compilation	Front-End: 1167 secs Back-End: 6376 secs FPGA-Compile: 16713 secs Total: 24256 secs

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

A html version of general statistics table is generated. It has links to compile profiler and log files from where these data are extracted:

**Basic info**

Keyword	Description
zAudit	ZeBu Compile Analysis, Diagnostics, and Recommendations
zcui work	/slowfs/vgzebucae@mrarafra/Native DPI/Multi RiscV light V2/zebu/CompilDir 2109-1/zcui.work
zebu work	/slowfs/vgzebucae@mrarafra/Native DPI/Multi RiscV light V2/zebu/CompilDir/zcui.work/backend default
zAudit db	zcui.work/zAudit.db
Compile profiler db	Path to compile Profiler
Stage	Completed

**Summary**

Category	Details	Log file Path
General	Platform: ZS4, ZeBu Release: S-2021.09-1, Vivado Release: 2021.1 AR76730	zcui.log
Memories	Total: 40, By Type: {BRAM: 40}, With # Ports: {1:3; 40, 4:7; 0, 8-15; 0, 16-31; 0, 32+; 0}	Path to zMem.dir
Clocks	Fetch Mode: yes, RTL Clocks: no, RLC Strategy: {zTopBuild: N/A, zCores: {FULL: 0}}	zTime.log
Size	REG: 1,866,146, LUT: 1,725,722, LUT6: 707,906 ( <b>41%</b> of LUT), BRAM: 2,560, RAMLUT: 0, DSP: 576, MUXCY: 142	Bundle.log
Mapping	Units: 1, Modules: 1, FPGAs: 3 Avg Fill: {REG: 3%, LUT: 5%; BRAM: 8%} Max FPGA Fill: {REG: 16%, LUT: 42%; BRAM: 50%} Avg CUT: {FPGA: 80}, Max CUT: {FPGA: 85}	zTopBuild.log
Routing	routerType: ZClik, routing Effort: min_distan, pinAssignmentEffort: zap, SocketMode: component, SAC: disabled	zParlog
Timing	Pre-Post delta before rounding: {driveClik: 10000 -> 10000 (0%)} bottleneck: clock constraint : 100ns, Longest Delays: {data: 71 ns, memory: 40 ns}, #MGT in top path: {data: 0, filter: 0} Max values in top 8 paths: {delay: 113, routing delay: 5, nodes: 2, logic fpgas: 2, routing fpgas: 0, modules: 1, units: 1, clkhubs: 0, mgt: 0}	zTime_fpga.log
FPGA	Total Num of FPGAs: 3 Original Winners: 3 PARFF Winners: 0 PARFF Winning FPGAs: None PARFF Failed FPGAs: [] Strategy: {} Num of Failed FPGAs: 0, Failed FPGAs: [] Num FPGA compile ongoing: 0, FPGA compile ongoing: [] Long Pole FPGA Compile Times: {U0_M0_F04: 195.933, U0_M0_F02: 171.917, U0_M0_F05: 159.917}	
Compilation	Front-End: 200 secs Back-End: 1630 secs FPGA-Compile: 12195 secs Total: 14025 secs	ECO_reserve_scope

**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
- Number of QiWC
- FPGA total compile time
- Vivado placer compile time
- Vivado router compile time
- FPGA IO cut
- Winner PARFF strategy

## Important ZeBu Log Files

- 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

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
WHS(Post_Place(ps))	562
WHS(Post_Place(ps))	-474
DataPathLogicDelay(ns)	0.123
DataPathNetDelay(ns)	0.392

---

## Important ZeBu Log Files

### ■ Information about MPF - activated

LEVEL-2 MPF INFO	
Metric	Value
FPGA	U0_M0_F04
Single_machine	0
Multi_machine	0

### ■ 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
```

zAudit memory command displays the following tables:

## Important ZeBu Log Files

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

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)					reset	init
zmem_name	width	depth	inferred type	inferred_reason	reset	init
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 display always the list of all memories in the design.*

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

`zAudit -z <zcu1.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`

---

## Important ZeBu Log Files

- `--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`
- `--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 <zcu111.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
- 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 <zcu111.work> mapping <core_identifier>
```

| zCoreBuild (Part_0) Post-split design statistics |     |           |      |           |      |       |        |     |       |  |
|--------------------------------------------------|-----|-----------|------|-----------|------|-------|--------|-----|-------|--|
| FPGA                                             | IO  | REG       | REG% | LUT       | LUT% | BRAM  | RAMLUT | DSP | MUXCY |  |
| UNIT0.M0D0.F2                                    | 143 | 557,428   | 11%  | 701,694   | 27%  | 895   | 1,376  | 180 | 209   |  |
| UNIT0.M0D0.F4                                    | 142 | 556,703   | 10%  | 701,454   | 27%  | 895   | 1,376  | 180 | 0     |  |
| UNIT0.M0D0.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

## Important ZeBu Log Files

- 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 <zcu10x.work> combinatory
```

This command provides two tables:

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

| Modules with large logic depth |       |       |
|--------------------------------|-------|-------|
|                                | Depth | LUTs  |
| TLDSetupModuleInner            | 249   | 3338  |
| File                           | 48    | 74    |
| TLAtomicAutomata               | 72    | 508   |
| TLTLMX4_1                      | 72    | 206   |
| NALDV                          | 72    | 1472  |
| ALU                            | 58    | 859   |
| ROM                            | 44    | 23    |
| Frontend                       | 48    | 461   |
| Rocket_0008                    | 39    | 3205  |
| Rocket_0009                    | 39    | 3205  |
| Rocket_0004                    | 39    | 3205  |
| Rocket_0003                    | 39    | 3205  |
| Rocket_0002                    | 39    | 3205  |
| Rocket_0000                    | 39    | 3205  |
| Rocket_000C                    | 39    | 3205  |
| Rocket_000E                    | 39    | 3205  |
| Rocket_000B                    | 39    | 3205  |
| Rocket_0009                    | 39    | 3205  |
| Rocket_000A                    | 39    | 3205  |
| OCache                         | 39    | 3596  |
| Rocket_0000                    | 39    | 3205  |
| Rocket_0001                    | 39    | 3205  |
| Rocket_0007                    | 39    | 3205  |
| Rocket_0006                    | 39    | 3205  |
| Rocket_0005                    | 39    | 3205  |
| AXI4Interleaver                | 40    | 6     |
| RVCExtractor                   | 29    | 193   |
| TPLPIC                         | 23    | 200   |
| LRAM                           | 23    | 314   |
| crq                            | 22    | 7     |
| PTW                            | 22    | 800   |
| TL_1                           | 19    | 328   |
| AXI4UserRanker_1               | 19    | 312   |
| PTW_Kernel                     | 18    | 3     |
| top_4_no_SWM                   | 18    | 89375 |
| TLBroadcast                    | 17    | 660   |
| top_4_no_SWM_0000              | 17    | 89375 |
| top_4_no_SWM_0001              | 17    | 89375 |
| top_4_SWM                      | 17    | 89375 |
| TLMonitor_1                    | 16    | 18    |
| AXI4UserRanker                 | 16    | 178   |
| AXI4ALU                        | 16    | 249   |
| EBuf                           | 15    | 18    |
| TLMonitor_35                   | 14    | 2395  |
| TLMonitor_36                   | 14    | 2344  |
| CLINT                          | 13    | 454   |
| TLToX14                        | 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:

## Important ZeBu Log Files

| TIMING TABLE                                                 |       |       |         |     |       |       |         |       |       |         |       |        |     |            |
|--------------------------------------------------------------|-------|-------|---------|-----|-------|-------|---------|-------|-------|---------|-------|--------|-----|------------|
| POST-FPGA   for pre-fpga: zAudit -z <zcui.work> timing --pre |       |       |         |     |       |       |         |       |       |         |       |        |     |            |
| top 35 timing paths                                          |       |       |         |     |       |       |         |       |       |         |       |        |     |            |
| id                                                           | slack | delay | routing | max | nodes | logic | routing | fpgas | fpgas | modules | units | clkhub | mgt | has memory |
| data_1                                                       | 0     | 981   | 122     | 16  | 8     | 6     | 2       | 1     | 1     | 0       | 0     | 0      | no  |            |
| data_2                                                       | 3     | 978   | 122     | 16  | 8     | 6     | 2       | 1     | 1     | 0       | 0     | 0      | no  |            |
| data_3                                                       | 19    | 962   | 117     | 16  | 7     | 5     | 2       | 1     | 1     | 0       | 0     | 0      | no  |            |
| data_4                                                       | 21    | 960   | 115     | 16  | 7     | 5     | 2       | 1     | 1     | 0       | 0     | 0      | no  |            |
| data_5                                                       | 22    | 959   | 115     | 16  | 7     | 5     | 2       | 1     | 1     | 0       | 0     | 0      | no  |            |
| data_6                                                       | 26    | 955   | 115     | 16  | 7     | 5     | 2       | 1     | 1     | 0       | 0     | 0      | no  |            |
| data_7                                                       | 30    | 951   | 115     | 16  | 6     | 5     | 2       | 1     | 1     | 0       | 0     | 0      | no  |            |
| data_8                                                       | 56    | 925   | 110     | 16  | 4     | 4     | 2       | 1     | 1     | 0       | 0     | 0      | no  |            |
| data_9..19                                                   | 78    | 903   | 150     | 56  | 4     | 3     | 1       | 1     | 1     | 0       | 0     | 0      | no  |            |
| data_20..29                                                  | 90    | 891   | 78      | 8   | 4     | 3     | 1       | 1     | 1     | 0       | 0     | 0      | no  |            |
| data_30                                                      | 92    | 889   | 78      | 8   | 4     | 3     | 1       | 1     | 1     | 0       | 0     | 0      | no  |            |
| data_31                                                      | 91    | 889   | 78      | 8   | 4     | 3     | 1       | 1     | 1     | 0       | 0     | 0      | no  |            |
| data_32..33                                                  | 93    | 888   | 78      | 8   | 4     | 3     | 1       | 1     | 1     | 0       | 0     | 0      | no  |            |
| data_34                                                      | 93    | 888   | 150     | 56  | 4     | 3     | 1       | 1     | 1     | 0       | 0     | 0      | no  |            |
| data_35                                                      | 94    | 887   | 78      | 8   | 4     | 3     | 1       | 1     | 1     | 0       | 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 |       |     |       |         |     |       |         |       |       |         |       |        |     |
| slack                                                            | paths | num | delay | routing | max | logic | routing | fpgas | fpgas | modules | units | clkhub | mgt |
| 0                                                                | 1     | 903 | 122   | 16      | 8   | 6     | 2       | 1     | 1     | 0       | 0     | 0      | no  |
| 3                                                                | 1     | 978 | 122   | 16      | 8   | 6     | 2       | 1     | 1     | 0       | 0     | 0      | no  |
| 19                                                               | 1     | 962 | 117   | 16      | 7   | 5     | 2       | 1     | 1     | 0       | 0     | 0      | no  |
| 21                                                               | 1     | 960 | 115   | 16      | 7   | 5     | 2       | 1     | 1     | 0       | 0     | 0      | no  |
| 22                                                               | 1     | 959 | 115   | 16      | 7   | 5     | 2       | 1     | 1     | 0       | 0     | 0      | no  |
| 26                                                               | 1     | 955 | 115   | 16      | 7   | 5     | 2       | 1     | 1     | 0       | 0     | 0      | no  |
| 30                                                               | 1     | 951 | 110   | 16      | 7   | 5     | 2       | 1     | 1     | 0       | 0     | 0      | no  |
| 56                                                               | 1     | 925 | 110   | 16      | 6   | 4     | 2       | 1     | 1     | 0       | 0     | 0      | no  |
| 78                                                               | 11    | 903 | 150   | 56      | 4   | 3     | 1       | 1     | 1     | 0       | 0     | 0      | no  |
| 90                                                               | 10    | 891 | 78    | 8       | 4   | 3     | 1       | 1     | 1     | 0       | 0     | 0      | no  |

| 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%       | 100%      | -      | FPGA: 100%   |
| 78                                                       | 11    | *      | 100%       | -         | -      | *            |
| 90                                                       | 10    | -      | 100%       | -         | -      | -            |

\* indicates 0%, \* indicates the first row that the column is 0%

| Unit, Module, FPGA crossings |       |        |        |         |       |       |        |        |        |        |         |
|------------------------------|-------|--------|--------|---------|-------|-------|--------|--------|--------|--------|---------|
| slack                        | paths | Unit 0 | Unit 1 | Unit 2+ | Mod 0 | Mod 1 | Mod 2+ | FPGA 0 | FPGA 1 | FPGA 2 | FPGA 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:

```
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 <zcu1.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
```

## Important ZeBu Log Files

### 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 UFT commands:
timing_analyze -advanced_command {set zcu1_ztime_dump verdi.db true}
timing_analysis -advanced_command {set zcu1_ztime_enable rtl true}
global_Setenv ZEBU_SERIALIZE_ZFPGATIMING_BACK_ANNOTATION 1
=====

Vivado
=====
ISSUE:
Core 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:MinLut6Remap 10000}
synthesis -advanced_command {Design:MaxLut6Utilization 0.2}
Router -advanced_command {cluster set -max_fill_lut6 20}
QWIC: 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:
SS1 : FPGA is having high SLL demand
FPGAs: U0_M0_F05
RECOMMENDATION:
QWIC: FPGAs having high qiwic fill which can lead to chance of FPGA failure or long compile time
ztopbuild -advanced_command {cluster set -max_fill_qiwic 40}
=====
```

### 3.13.7 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.

#### 3.13.7.1 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

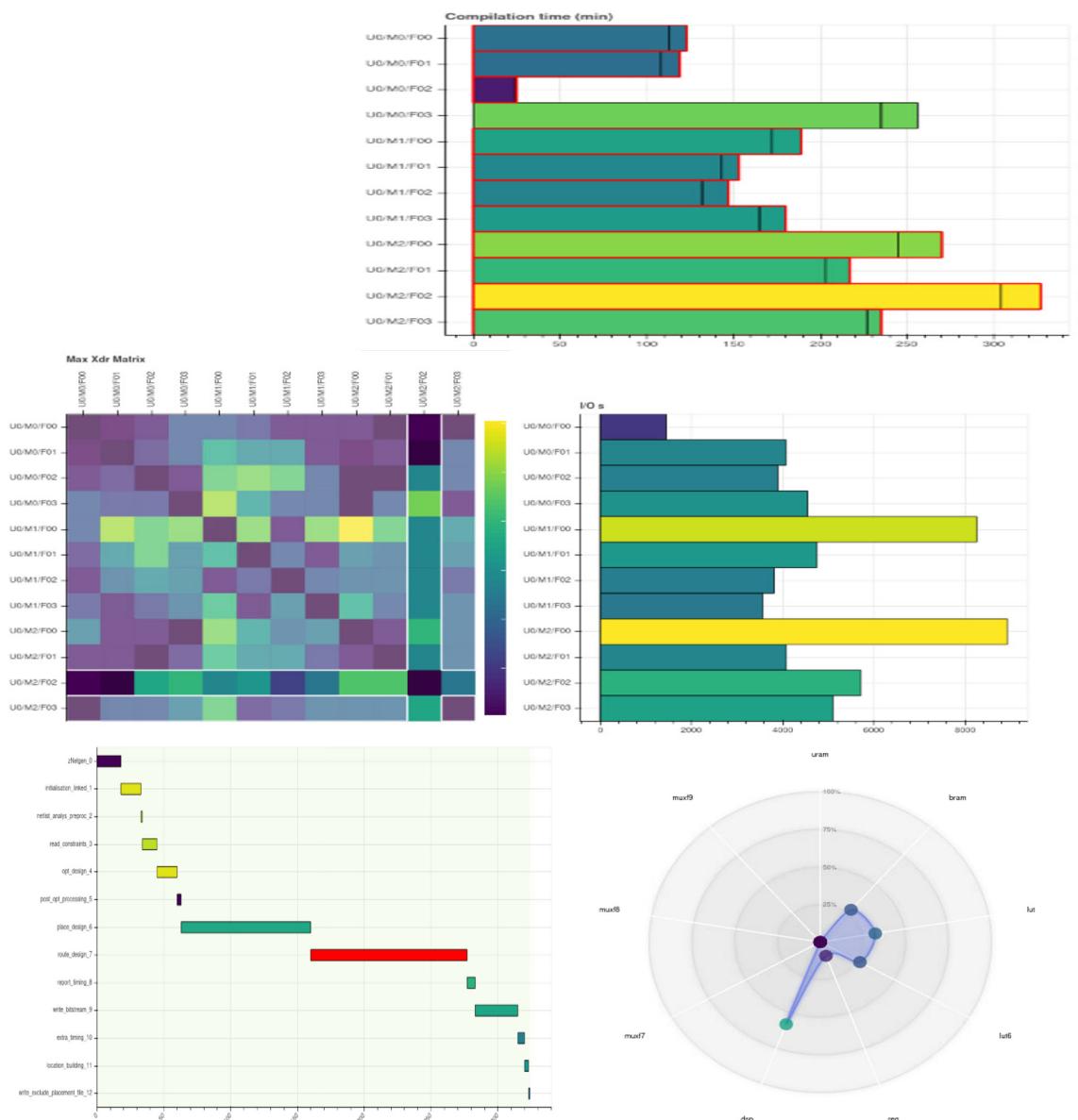
browser to open abc.html file. For example, `firefox abc.html`.

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 (highlighted in red in [Figure 34](#) )
  - PARFF triggered (highlighted in blue in [Figure 34](#) )
- 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.

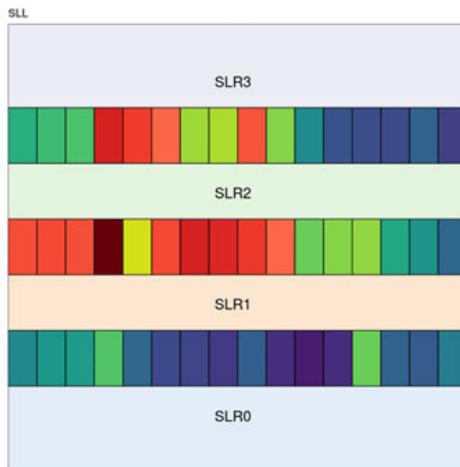
## Important ZeBu Log Files

**FIGURE 34.** Compilation details Tab

## SLL

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

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

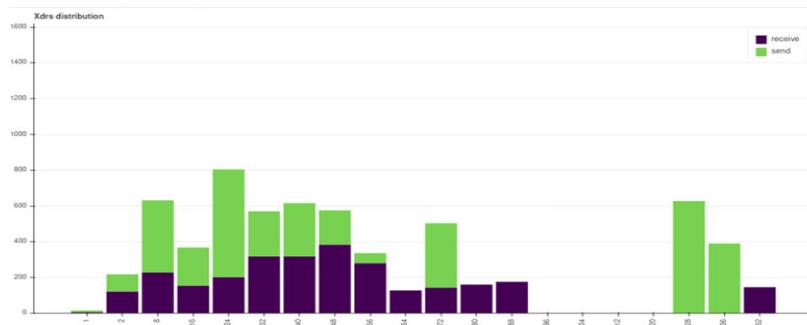


**FIGURE 35.** SLL Tab

## Xdrs distribution

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

## Handling Misfiring SVAs with Latches and Memories in fanin

**FIGURE 36.** Xdrs distribution Tab

## 3.14 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.

### 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

- UFS 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.*

## 3.15 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.*

### 3.15.1 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
```

### 3.15.2 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 #

```

---

Enhanced Verilog force/release

```
#      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_FIFOOr1_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`

### 3.15.3 Limitations

Replication of multiport memories with cross hierarchies are not supported.

## 3.16 Enhanced Verilog force/release

Verilog force release is supported by default on ZeBu. The instrumentations are handled by **zTopBuild** with legacy support. Therefore, an enhanced implementation has been suggested to be handled at synthesis level to be more accurate according to LRM.

### Usage:

To enable enhanced Verilog force release, specify the following UTF command:

`verilog_force_release -enable_wls true`

### 3.16.1 Reporting Verilog force/release Information

When enabling new 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 UFT command in your UFT 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
```

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

```

## 3.16.2 Limitations

The following limitations are applicable:

- New 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.
  - For complex signals such as interface and struct members, if the synthesizer finds force then it gives a warning and applies force only on the signal or the member.
  - 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:

```

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.
- 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;
```

- If there are force statements inside initial block with some conditions, which are not constant during synthesis time, VCS might not honor them. However, ZeBu synthesizer creates different logic under different supported options, which might create simulation synthesis mismatches.

**Example:**

```
initial  
begin  
    if (cond)  
        force a = 1'b1;  
end
```

## 3.17 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.

## 3.18 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).

### 3.18.1 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
```

## 3.19 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](#)

### 3.19.1 Enabling PDM Support

---

## Performance Driven Multi-die Multiplexing Support

To enable the PDM support, use the following UFT 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)

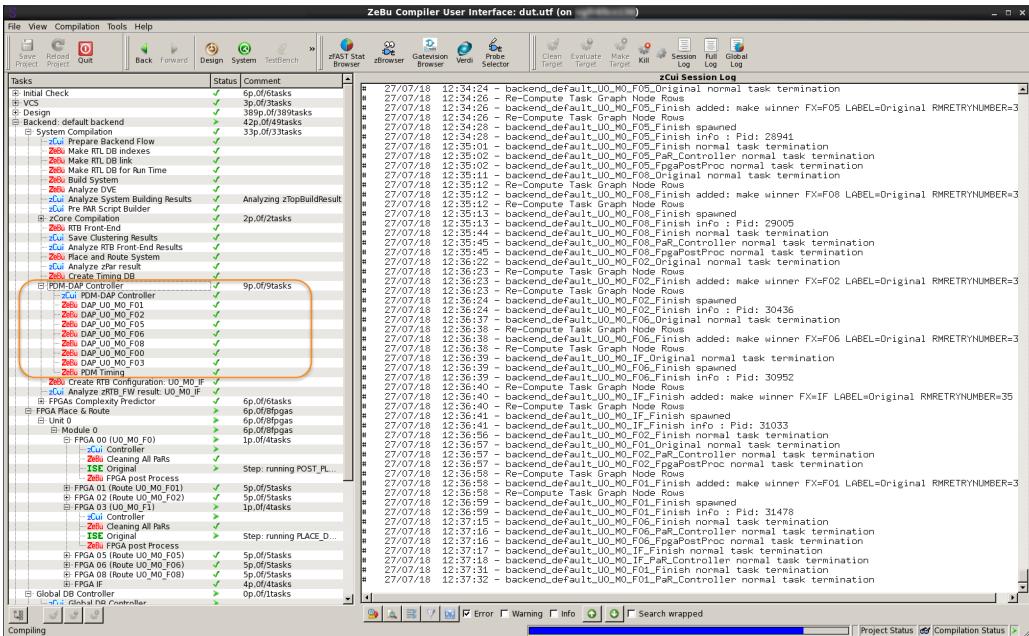
**Note**

*By default, the PDM\_DIRECT mode is set when PDM support is enabled.*

---

### 3.19.2 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 37.** PDM Flow in zCui

# 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\*](#)

## 4.1 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 7** 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:<br>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 |

## 4.2 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. This is shown below.

**FIGURE 38.** Instantiate the zceiClockPort clock generation macro

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. See the code below:

```
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
```

---

## 4.3 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.

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*
- *Global Time*
- *Limitations of the Clock Delay Feature*

### 4.3.1 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 of Dynamic Reconfiguration with clockDelayPort\*](#)

#### 4.3.1.1 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 as shown below:

```
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 default values 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 8** Using ClockDelayPort Macro

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

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

| C++ API                                            | C API                                                  | Description                                                                                                                                                                                                                                       |
|----------------------------------------------------|--------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ClockDelayPort::doNewReset</code>            | <code>ZEBU_ClockDelayPort_doNewReset</code>            | Takes the hierarchical name of the <code>clockDelayPort</code> instance (example, "hw_top.cdp1") and the delay at which to toggle the reset. The "reset" port of this instance toggles at the designated delay.                                   |
| <code>ClockDelayPort::reconfigureClockShape</code> | <code>ZEBU_ClockDelayPort_reconfigureClockShape</code> | Takes the hierarchical name of the <code>clockDelayPort</code> instance (e.g. "hw_top.cdp1") and the new duty lo, duty hi, phase and whether immediate reconfiguration is required. The "clock" port of the instance starts behaving accordingly. |

| C++ API                                              | C API                                                       | Description                                                                                                                                                                                                                 |
|------------------------------------------------------|-------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ClockDelayPort::disableClock</code>            | <code>ZEBU_ClockDelayPort_disableClock</code>               | Takes the hierarchical name of the <code>clockDelayPort</code> instance (e.g. "hw_top.cdp1"). This stops the "clock" port of this. But, it does not have any effect on any other clocks in the system. Emulation continues. |
| <code>ClockDelayPort::Configuration getConfig</code> | <code>ZEBU_ClockDelayPort_Configuration getConfig();</code> | Returns the current delay settings of particular clock delay port from hardware.                                                                                                                                            |

#### 4.3.1.2 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
```

### 4.3.1.3 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".

#### 4.3.1.4 Using ZeBu Clock Delay Primitive

ZeBu clock delay primitive can be used as follows:

```
clockDelayPort #(  
    .initial_dutyLow      ( 25 ),  
    .initial_dutyHigh     ( 15 ),  
    .initial_phase        ( 0  ),  
    .initial_resetValue   ( 0  ),  
    .initial_resetDuration( 1  )  
) u_clk_gen (  
    .clock ( clk ),  
    .reset ( reset )  
) ;
```

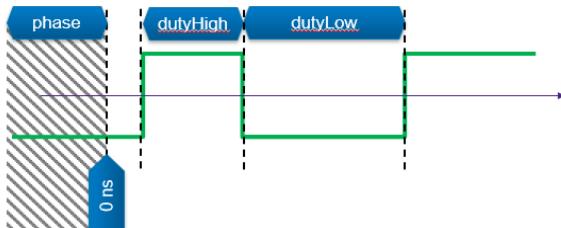
where,

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 39.** Clock Delay Primitive



#### 4.3.1.5 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  $T + \text{tolerance}$  value instead of  $T$ , and all events in the time window  $[T:T+\text{tolerance}]$  occur at the same driverClock. This reduces the number of driverClk transitions; but reduces accuracy.

## Example

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

- A0 #13
- A1 #15
- A2 #17
- A3 #10
- A4 #23
- A5 #28

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

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

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

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

Where,

- **Yellow cell:** Clock has an event, and second event is scheduled, remaining delay is 7.
- **Red cell:** Minimum delay.
- **No color cell:** Clock still has a pending delay and no event in this driver clock cycle.
- **Grey row:** The scenario with auto-tolerance 9 (since the smallest delay of a clock is 10) is as follows:
  - Delays between min and min+9 are granted.
  - Minimum delays are marked in red and additional delays granted are marked in yellow.

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

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

#### 4.3.1.6 Limitations of Dynamic Reconfiguration with `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.

#### 4.3.2 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*

#### 4.3.2.1 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`.

#### 4.3.2.2 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
```

---

## Time-Based Clock Control

## ■ 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_occurred)
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 (use either of the constructs mentioned below)**

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

---

Or

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

---

### 4.3.2.3 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);
```

---

#### 4.3.2.4 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(...);
```

#### 4.3.3 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.

#### 4.3.4 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](#)".

### 4.3.5 Timescale and Precision

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

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

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

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

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

### 4.3.6 Compilation Results

To find all synthesized delays, run the following command:

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

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

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

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

## 4.3.7 imestamp 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.*

---

## 4.3.8 Global Time

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

```
/*
 * \brief Type for returning the current value of time
 *   timeUnit is explained by example here:
 *     If timeUnit is -9, it refers to "1ns"
 *     If timeUnit is -8, it refers to "10ns"
 *   The range of values is from 2 to -15
 */

typedef struct _ZEBU_CurrentTimeType
{
    unsigned long long int highWord;
    unsigned long long int lowWord;
    ZEBU_TimeUnit timeUnit;

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

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

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

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

### 4.3.9 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**.

## 4.4 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.*
-

## 4.5 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*

# 5.1 Controlling Runtime Parameters

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

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

## 5.1.1 Emulation Speed

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

## 5.1.2 Clock Definition

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

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

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

## Controlling Runtime Parameters

```

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 = "-_-"
$U0.M0.clock2.VirtualFrequency = 1;
$U0.M0.clock2.GroupName = "clock_group"

```

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

```
setenv ZEBU_VIRTUAL_FREQ_ROUND_ENABLE 1
```

By default, this function is disabled.

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

```
setenv ZEBU_VIRTUAL_FREQ_ROUND_SEARCH_DEPTH 3
```

By default, the value is set to 3.

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

## Example

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

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

### 5.1.3 Memory Initialization

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

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

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

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

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

## 5.2 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](#)

### 5.2.1 Controlling Clocks

This section describes the commands to control the ZeBu clocks.

**Note**

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

---

#### 5.2.1.1 Obtaining a List of Clock Groups

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

The `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.

#### 5.2.1.2 Enabling Clocks for N Cycles

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

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

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

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

**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**.

### 5.2.1.3 Disabling Clocks

**zRci** supports the following command to disable clock:

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

### 5.2.1.4 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}
```

## 5.2.2 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*

### 5.2.2.1 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.

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

### 5.2.2.2 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***.

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

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

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

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

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

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

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

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

}

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

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

return(0);
}
```

## 5.4 Reading Signal Values During Runtime

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

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

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

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
```

---

## 5.5 Changing the Signal Values During Runtime

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

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

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*

### 5.5.1 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).

#### 5.5.1.1 C++ Example

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

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

### 5.5.1.2 zRci Example

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

### 5.5.2 Injecting a Value on a Signal

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

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

### 5.5.2.1 C++ Example

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

---

Changing the Signal Values During Runtime

### 5.5.2.2 zRci Example

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

---

### 5.5.3 Forcing a Value on a Signal

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

#### 5.5.3.1 C++ Example

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

---

#### 5.5.3.2 zRci Example

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

---

## 5.5.4 Releasing a Forced Signal

### 5.5.4.1 C++ Example

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

---

### 5.5.4.2 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 describes the following subtopics.

- *Enabling RunManager*
- *RunManager APIs*

## 6.1 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.

## 6.2 RunManager APIs

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

**TABLE 9** APIs Supported with RunManager

| API Name                                                                | Description                                                                                                                                                                                                           |
|-------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| RunManager<br>*Board::getMainRunManager()                               | Creates and returns a pointer to the main RunManager instance.<br>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<br>*Board::getNewRunManager(unsigned int instanceNumber = 0) | Creates and returns a pointer to the RunManager instance.<br>Returns NULL when all the instances of RunManager are used or the requested instance is not available.                                                   |

**TABLE 9** APIs Supported with RunManager

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

**TABLE 9** APIs Supported with RunManager

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

# 7 Using SystemVerilog Assertions

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

- [\*SVA Compilation\*](#)
- [\*Runtime Usage\*](#)
- [\*Post Processing SVA\*](#)
- [\*Supported SVA subset\*](#)

## 7.1 SVA Compilation

The compilation options for SVA are available using the `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.

## 7.2 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](#)

### 7.2.1 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\*](#)
- [\*Dealing with Assertion Failures\*](#)
- [\*Changing the Reported Severity Level\*](#)
- [\*Disabling/Enabling Assertions and Signaling SVA Failures\*](#)

### 7.2.1.1 Starting Assertion Processing

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

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

#### Post-Processing Mode

In this mode, all assertion failure messages during design run are dumped in a report file for post-emulation analysis. A post-processing tool, 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.

## 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:

## Runtime Usage

```
// 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**

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

### 7.2.1.2 Stopping SVA Processing

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

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

### 7.2.1.3 Resetting SVA Fail Counters

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

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

You can also reset the failure counters using the config sva\_reset\_counters command. For details, see the **ZeBu Unified Command-Line User Guide**.

## 7.2.1.4 Dealing with Assertion Failures

### Clock Stopping on Assertion Failure

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

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

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

where:

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

To disable Clock Stopping upon assertion failure:

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

### Interface of the OnStop Callback

The interface of the OnStop callback is as follows:

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

where:

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

### 7.2.1.5 Changing the Reported Severity Level

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

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

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

The available values for severity are defined by the ZEBU\_SVA\_Severity type in the ZEBU\_ROOT/include/Types.h header file:

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

#### Example

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

### 7.2.1.6 Disabling/Enabling Assertions and Signaling SVA Failures

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

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

The prototype is as follows:

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

where:

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

## 7.2.2 Controlling Assertions Using 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\*](#)

### 7.2.2.1 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.

### 7.2.2.2 Stopping SVA Processing

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

### 7.2.2.3 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.

#### 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**.

### 7.2.2.4 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**.

### 7.2.3 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

## Post Processing SVA

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

### Note

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

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

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

## 7.3 Post Processing SVA

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

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

For example,

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

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

## 7.4 Supported SVA subset

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

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

**TABLE 10** Supported SystemVerilog SVA Subset

| Supported SystemVerilog SVA Subset |                              | Status |
|------------------------------------|------------------------------|--------|
| SEQUENCES                          | cycle_delay_range            | S      |
|                                    | ## 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*     |
| Sequence operators                 | throughout                   | S      |
|                                    | intersect                    | S      |

Supported SVA subset

**TABLE 10** Supported SystemVerilog SVA Subset

| Supported SystemVerilog SVA Subset                              |                    | Status |
|-----------------------------------------------------------------|--------------------|--------|
| 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      |

**TABLE 10** Supported SystemVerilog SVA Subset

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

Supported SVA subset

**TABLE 10** Supported SystemVerilog SVA Subset

| Supported SystemVerilog SVA Subset                                                                                                                                                                                                               | Status |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------|
| ASSERTION<br>CONTROL TASKS<br>Feature supported through C++ API                                                                                                                                                                                  | S      |
| \$assertkill                                                                                                                                                                                                                                     | S      |
| \$asserton                                                                                                                                                                                                                                       | S      |
| \$assertoff                                                                                                                                                                                                                                      | U      |
| CONCURRENT ASSERTIONS                                                                                                                                                                                                                            | S      |
| assert                                                                                                                                                                                                                                           | S      |
| assume                                                                                                                                                                                                                                           | S      |
| cover                                                                                                                                                                                                                                            | S*     |
| Concurrent assertions outside procedural code                                                                                                                                                                                                    | S      |
| Concurrent assertions inside procedural block <ul style="list-style-type: none"> <li>● Assertions with clocking event different from clocking event of procedural block not supported</li> </ul>                                                 | S*     |
| Concurrent assertions inside procedural loop <ul style="list-style-type: none"> <li>● Assertions with clocking event different from clocking event of procedural loop not supported</li> <li>● Multi-clocked assertions not supported</li> </ul> | S*     |
| DEFERRED ASSERTIONS                                                                                                                                                                                                                              | S      |

**TABLE 10** Supported SystemVerilog SVA Subset

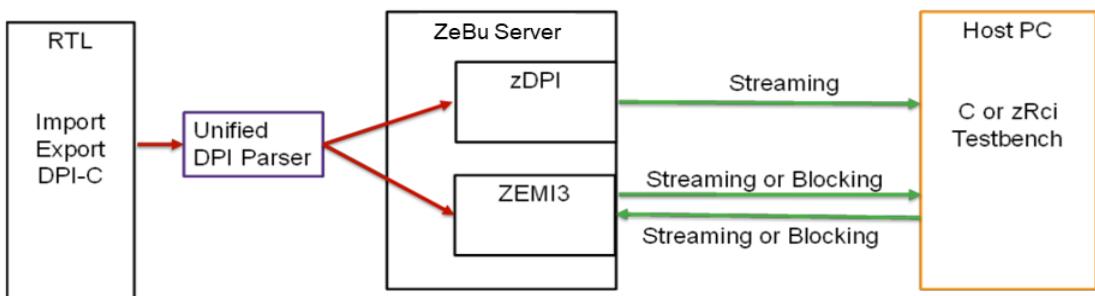
| Supported SystemVerilog SVA Subset | Status |
|------------------------------------|--------|
| IMMEDIATE ASSERTIONS               | S      |
| EXPECT                             | U      |

# 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 40.** 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.

**Note** *I*

---

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\*](#)

## 8.1 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 are not handled in a good way: always bud, some work/some not work

## 8.1.1 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

## 8.1.2 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 funtions, ideal dor "DPI Monitors"
  - Unified DPI handle stream from HW -> SW and from SW-> HW, it synthetize automatically all DPI calls as ZEMI3 or zDPI with unified parser
  - Regarding system task/function:
    - zDPI handle system task/function but not in a good way: always bud, some work/some not work
    - Unified DPI handle system task/function correctly

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

**TABLE 11** Unified DPI Versus zDPI - Configuration Setup

| Has Outputs | Context |                              |                    |                 |                  |
|-------------|---------|------------------------------|--------------------|-----------------|------------------|
| Yes         |         |                              | Ignored            | Blocking        | Blocking         |
| No          | Yes     | Initial                      | Ignored            | Blocking        | Blocking         |
| No          | No      | Initial:<br>Only 1 call      | Ignored            | Blocking*       | Blocking*        |
| No          | No      | Initial:<br>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_alloc_export |                 |                  |
|             |         | Export                       | Ignored            | Supported       | Supported        |

**Note**

1. 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.
2. Any function that is zDPI but that shares a process with ZEMI3 changes to ZEMI3-Streaming.

## 8.2 Compilation

### 8.2.1 zDPI (Default Technology)

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

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

```
dpi_synth -enable ALL
```

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

|                                          |                                                            |
|------------------------------------------|------------------------------------------------------------|
| <code>dpi &lt;instance name&gt;</code>   | Selects the given DPI function                             |
| <code>module &lt;module name&gt;</code>  | Selects all DPI calls in all instances of the given module |
| <code>tree &lt;hierarchy name&gt;</code> | Selects all DPI calls below the given hierarchy            |
| <code>path &lt;hierarchy name&gt;</code> | Selects all DPI calls inside the given hierarchy instance  |

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

For example:

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

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

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

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

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

Synthesis also generates the following files:

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

This file is named as: zcui.work/design.synth\_Default\_RTL\_Group/dpi\_log/<module>.log

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

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

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

This header is zcui.work/<backend\_compil\_dir>/grp0\_ccall.h

## 8.2.2 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 system 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 .
- 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.

### 8.2.2.1 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

```
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 41.** Example 1: 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
```

**FIGURE 42.** Example 2: Analyzing of Import DPI Called in the Initial Block

## DPI Exports Summary

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

## Compilation

| 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".

| 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)

**FIGURE 43.** Example 1: List of In and Out Ports of Import DPI Called in Initial Block



## Compilation

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)

**FIGURE 44.** Example 2: List of In and Out Ports of System Task (\$plusargs)

## **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
```

## **8.2.3 Compiling ZeBu DPI files**

After zebu.work is built, esnure 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
```

## **8.3 Writing DPI Import Functions**

This section consists of the following sub-sections:

- *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"
```

## 8.4 ZeBu API to Control DPI Functions at Runtime

The ZeBu API provides the following control capabilities:

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

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

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

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

## 8.4.1 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*

### 8.4.1.1 Clocking Mode

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

This section consists of the following sub-sections:

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

#### 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.

**Note**

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

---

### 8.4.1.2 Enabling Synchronization of DPI Calls

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

The syntax of this method is:

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

---

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

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

#### 8.4.1.3 Selecting Function Calls Only on Input Changes

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

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

---

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

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

---

#### 8.4.1.4 Loading Dynamic Libraries

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

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

---

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

If <path> is not specified, the **LD\_LIBRARY\_PATH** environment variable must

include the path to <library name>.so.

**Note**

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

### 8.4.1.5 Starting DPI Call Processing

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

To start the processing of import calls by name:

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

where,

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

**Note**

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

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

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

where,

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

**NOTE:** *The Start method can be called multiple times to enable multiple sets of function calls.*

#### 8.4.1.6 Stopping DPI Call Processing

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

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

To stop the processing of import calls by name:

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

---

where,

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

**Note**

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

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

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

where,

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

## 8.4.2 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_synchronization`
- `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](#).

### 8.4.3 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 plusargs from your DPI C code.

## 8.5 Processing DPI Function Calls Offline

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

Processing DPI calls offline prevents:

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

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

To use this feature, perform the following steps:

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

## 8.5.1 Identifying the DPI Functions

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

## 8.5.2 Enabling the DPI Offline Feature During Emulation

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

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

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

### 8.5.2.1 Using ZEBU\_OFFLINE\_DPI and its Offline DPI Specification File

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

- [\*Setting the ZEBU\\_OFFLINE\\_DPI Environment Variable\*](#)
- [\*Writing the Offline DPI Specification File\*](#)

#### Setting the ZEBU\_OFFLINE\_DPI Environment Variable

You set the ZEBU\_OFFLINE\_DPI environment variable to a file listing all the DPI function calls to execute offline as follows.

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

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

- EnableSynchronization
- DisableSynchronization
- Start
- SetOfflineDumpName

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

## Writing the Offline DPI Specification File

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

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

### Note

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

The offline DPI specification file contains the following information:

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

This setting in the offline DPI specification file overrides the

EnableSynchronization and DisableSynchronization methods used at runtime in the testbench.

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

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

- One DPI function pattern in one line.
- Blank lines and leading and trailing whitespaces are ignored.
- Comments must start with the pound character (#).
- Regular expressions for DPI function patterns must follow the syntax rules for Portable Operating System Interface (POSIX) Extended Regular Expression. To avoid partial matches, the circumflex (^) and dollar (\$) characters automatically wrap regular expressions if they are not already present.  
Example: f.\*o matches foo or foofoo, but does not match foobar
- If the pattern for DPI function name is (\*), it is replaced with (.\*)

### **Example 1**

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

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

---

where,

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

### **Example 2**

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

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

---

where,

- target defines the path to the ZTDB file.

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

### 8.5.2.2 Using the DPI Offline Feature With Methods Within the Testbench

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

#### Defining the ZTDB File

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

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

where,

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

For example,

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

## Activating DPI Function Calls in Offline Mode

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

To start the offline processing of DPI calls by name:

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

where,

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

**Note**

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

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

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

where,

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

**Note**

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

---

## Reading the Offline DPI Specification File

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

The syntax of this method is:

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

---

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

For example,

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

---

### 8.5.3 Creating a Shared Library

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

For example

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

---

### 8.5.4 Creating an Input File for the **zdpiReport** Tool

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

### 8.5.5 Using the **zdpiReport** Tool

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

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

---

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

**TABLE 12** zdpiReport Mandatory Parameters

| Parameters                             | Description                                                                                                                                                                                                                                                                                                                                                                   |
|----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-f &lt;flist filename&gt;</code> | <ul style="list-style-type: none"> <li>Specifies the filename containing the names of DPI functions to be executed by <b>zdpiReport</b>.</li> <li>Supports system tasks such as \$display.</li> <li>Selects system task in any scope to be enabled.</li> </ul> <p><b>Example:</b><br/>To enable system task only in top.d1 scope:<br/><code>\$display scope=top.d1</code></p> |
| <code>-i &lt;.ztdb filename&gt;</code> | Specifies the ZTDB filename where the offline calls were previously dumped.                                                                                                                                                                                                                                                                                                   |
| <code>-l &lt;.so filename&gt;</code>   | Specifies the shared library containing the DPI function implementation.                                                                                                                                                                                                                                                                                                      |

**TABLE 13** zdpiReport Options

| Parameters                        | Description                                                                                            |
|-----------------------------------|--------------------------------------------------------------------------------------------------------|
| <code>-synchronize</code>         | Synchronizes DPI calls.                                                                                |
| <code>-z &lt;zebu.work&gt;</code> | Specifies the compilation directory where the runtime database is available (default is . /zebu.work). |
| <b>To speed up decoding</b>       |                                                                                                        |
| <code>[-mtdecode]</code>          | Enables parallelized.ztdb decoding.                                                                    |
| <code>[-mtexec [N]]</code>        | Enables parallelized <b>zDPI</b> execution, in at most N threads.                                      |
| <code>[-mtread M[,T]]</code>      | Enables parallelized.ztdb reader: maximum M MB of memory and T threads.                                |
| <code>[-synchronize-scope]</code> | Synchronizes DPI calls within scopes.                                                                  |

**TABLE 13** zdpiReport Options

| Parameters                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | Description |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|
| <ul style="list-style-type: none"><li>• If the <code>[-mtexec [N]]</code> option is used, functions belonging to different synchronization groups must be thread safe.</li><li>• If the <code>-synchronize-scope</code> option is used, a separate synchronization group is created for <b>zDPI</b> calls in each scope.</li><li>• If <code>-f</code> is used multiple times with <code>-synchronize</code>, functions enabled by each <code>-f</code> file are placed in a separate synchronization group.</li><li>• If the <code>-synchronize</code> option is not used, all functions execute in a single thread.</li></ul> |             |

**TABLE 13** zdpiReport Options

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

For example,

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

**Note**

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

### 8.5.5.1 Speed-up Decoding

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

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

### 8.5.5.2 Diagnostics

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

#### Example

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

### 8.5.5.3 Profiling

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

#### Example

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

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

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

---

## 8.6 Optimization Guidelines

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

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

### 8.6.1 Enhancing Data Transfer

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

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

**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 14** 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.

## 8.6.2 Defining Sampling Clock Frequency

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

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

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

For example,

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

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

## 8.6.3 Investigating Emulation Slowdowns

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

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

### 8.6.3.1 Checking with zDPI Disabled

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

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

For example,

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

### Combining DPI Calls

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

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

For example,

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

### Limiting the Argument Lists

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

### 8.6.3.2 Checking With zDPI Enabled

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

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

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

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

**Note**

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

---

## 8.7 Limitations of the ZeBu DPI Technologies

### 8.7.1 zDPI

This section lists the limitation of the **zDPI** feature in the current version of the software. See the following subsections:

- [Multiple Clock Groups](#)
- [Performance](#)
- [Synthesis](#)

#### 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.

---

Example of DPI Function Calls to Print Counter Value

- DPI function calls in functions/tasks are not synthesized.
- System tasks are not supported in this version.

## 8.7.2 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.

## 8.8 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

### 8.8.1 Design

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

- *System Verilog Code*
- *Synthesis*

### 8.8.1.1 System Verilog Code

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

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

---

The DPI function is called from the design on each rising edge of the design clock clk as displayed in the following code snippet:

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

---

The Following is the complete SystemVerilog test:

## Example of DPI Function Calls to Print Counter Value

```
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
```

### 8.8.1.2 Synthesis

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

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

---

### Example of DPI Function Calls to Print Counter Value

```
zcui.work/design/synth_Default_RTL_Group/dpi_log/counter.log
```

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

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

## 8.8.2 Implementation of DPI Functions

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

The grp0\_ccall.h file is included to check that the prototypes of the user DPI functions are compliant with the import declarations in SystemVerilog.

The print\_count DPI function is implemented in the dpi.cc file:

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

extern "C" void print_count(const svBitVecVal *id, const svBit ci,
const svBitVecVal *count)
{
    svScope s = svGetScope();
    printf("Counter '%s', id=%u, count=%u\n", svGetNameFromScope(s),
id[0], count[0]);
}
```

In this example, a dpi.so dynamic library is created from dpi.cc:

```
$ g++ -fPIC -shared -o dpi.so dpi.cc -I$ZEBU_ROOT/include
-Izcui.work/zebu.work
```

## 8.8.3 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**.

---

### 8.8.3.1 Implementation of the Testbench

In this example, the C++ testbench controls the co-simulation driver and the DPI calls.

The following are the specific calls that control the DPI calls:

- To load the `dpi.so` dynamic library:

```
CCall::LoadDynamicLibrary(z, "dpi.so");
```

---

- To sample the DPI calls on both edges of clock `clk`:

```
CCall::SelectSamplingClocks(z, "clk");
```

---

- To start all the DPI calls:

```
CCall::Start(z);
```

---

The source code of the testbench (`tb.cc`) is as follows:

```
#include <libZebu.hh>
using namespace ZEBU;
using namespace std;
```

---

## Example of DPI Function Calls to Print Counter Value

```
int main(){
    try {
        // Board init
        Board *z = Board::open("zcui.work/zebu.work");
        Driver *d      = z->getDriver("counter_cosim");
        Signal *reset = d->getSignal("reset");
        Signal *load  = d->getSignal("d");
        Signal *data  = d->getSignal("data");
        Signal *ci    = d->getSignal("ci");
        Signal *pce   = d->getSignal("print_count_enable");

        d->connect();

        // enabling the DPI calls
        CCall::LoadDynamicLibrary(z, "dpi.so");
        CCall::SelectSamplingClocks(z, "clk");
        CCall::Start(z);

        z->init();
        // counter reset
        *reset = 1;
        d->run(2);
        // counter load
        *reset = 0;
        *load = 1;
        *data = 0xBABEFACE;
        d->run(1);
        // counter enable for 200000 cycles
        *ci = 1;
    }
}
```

```
d->run(200000);
// Stopping the DPI calls
CCall::Stop(z);

d->disconnect();

z->close();

} catch (exception &x) {
    cerr << "Got exception " << x.what() << endl;
}
}/
```

The testbench is compiled with the g++ compiler into an executable file (tb):

```
$ g++ -o tb tb.cc -I$ZEBU_ROOT/include -L$ZEBU_ROOT/lib -lZebu
```

---

### 8.8.3.2 Proceeding With Runtime

Ensure the LD\_LIBRARY\_PATH environment variable is correctly set, and launch the testbench as displayed in the following code snippet:

```
$ export LD_LIBRARY_PATH=<path to dpi.so>:$LD_LIBRARY_PATH
$ ./tb
```

---

### 8.8.4 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
```

---

### Example of DPI Function Calls to Print Counter Value

```
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 chapter discusses the following topics.

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

## 9.1 Save and Restore - Recommendations

- Only the state of the ZeBu hardware is saved.
  - Restore is done by a separate script.
- Before saving the state of the hardware,
  - Flush and close waveform files (if any), and
  - Flush and close DPI files (if any)

## 9.2 C++ Method for Save and Restore

To save the hardware state, use the `FastHardwareState` class as follows:

```
FastHardwareState::FastHardwareState();
```

This method is the constructor of the `FastHardwareState` object.

The `FastHardwareState` class provides the following methods:

- To initialize the `FastHardwareState` object, use the following method:

```
void FastHardwareState::initialize(Board *board)
```

where,

□ `board` is a pointer to the current `Board` object.

- To initialize and filter the `FastHardwareState` object, use the following method:

```
void FastHardwareState::initialize(Board *board, const Filter  
*filter)
```

## C++ Method for Save and Restore

where,

- board is a pointer to the current Board object.
- filter is a pointer to a Filter object and allows to filter the following types of components to save:
  - ◆ internal signals
  - ◆ driver signals
  - ◆ internal
  - ◆ external memories
  - ◆ clocks

- To capture the hardware state into memory, use the following method:

```
void FastHardwareState::capture()
```

- To write the hardware state previously captured on the disk, use the following method:

```
void FastHardwareState::save(const char *filename, bool inParallel
= false)
```

Where,

- filename is the name of the directory in which the hardware state must be saved.
- inParallel specifies if the hardware state must be saved using parallel tasks.

**Note**

*The FastHardwareState::save method cannot be called after calling Board::close.*

*The FastHardwareState::save method does not release the memory allocated by the call to the capture method.*

The following C++ method is available for save and restore:

```
static Board *Board::restoreHardwareState  
    const char *saveDirectory,  
    const char *zebuWorkPath = "./zebu.work",  
    const char *designFile = 0,  
    const char *processName = "default_process")  
    throw(std::exception);  
  
static Board *Board::restoreHardwareStateWithSelectionFile  
    const char *saveDirectory,  
    const char *zebuWorkPath = "./zebu.work",  
    const char *selectionDBPath = 0,  
    const char *designFile = 0,  
    const char *processName = "default_process")  
    throw(std::exception);
```

where,

- `saveDirectory` is the name of the file from which the state of the ZeBu hardware is to be restored.
- `zebuWorkPath` (default "../zebu.work") is the path to the `zebu.work` directory.
- `designFile` (default "designFeatures") is the name of the `designFeatures` file.
- `processName` (default "default\_process") is the name identifying this process in the case of a multi-process testbench.
- The `Board::restoreHardwareState()` and `Board::restoreHardwareStateWithSelectionFile()` methods return a pointer to the `Board` object or `NULL` if the open operation failed.

---

C++ Example

## 9.3 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" );

```

---

## 9.4 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.

## 9.5 zRci Example

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

```
checkpoint -save myHwState0  
[...]  
checkpoint -restore myHwState0
```

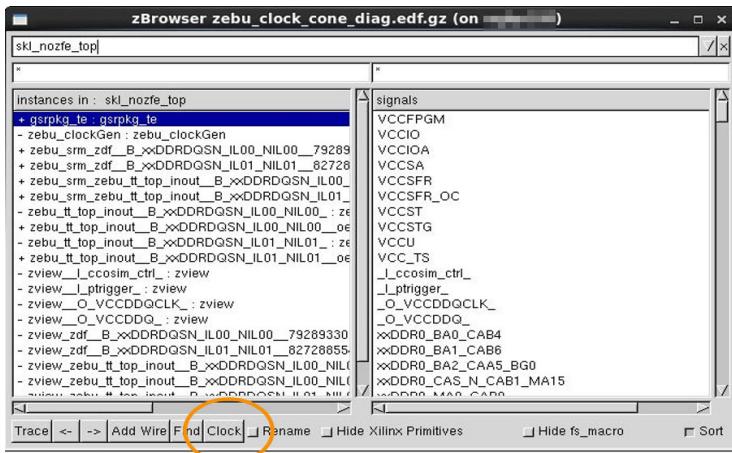
---

# 10 Clock Cone Visualization in zBrowser

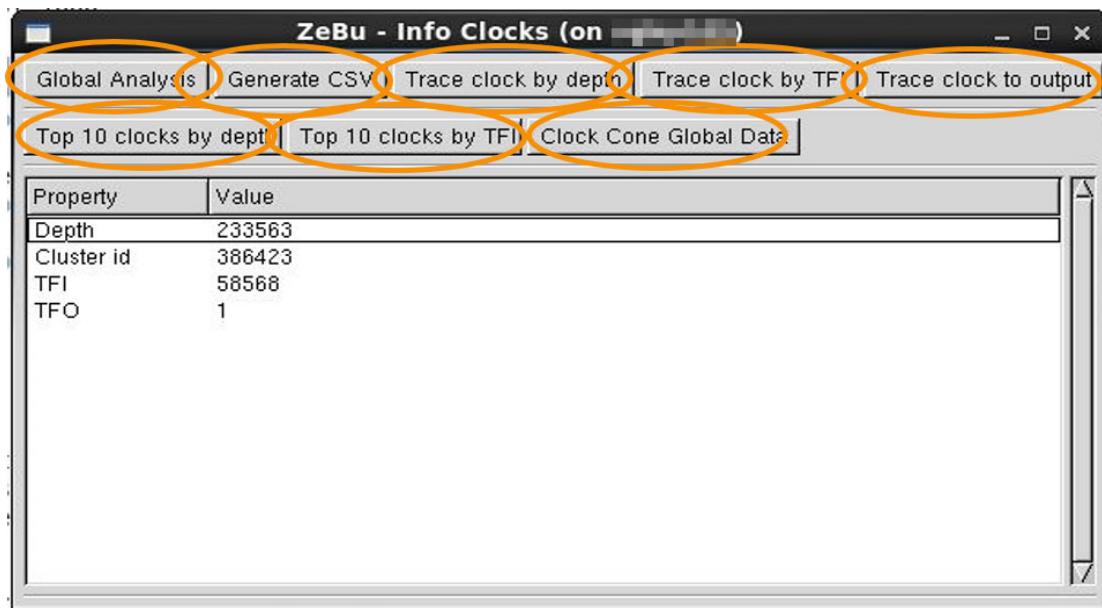
A **Clock** panel is introduced in **zBrowser** to support the following:

- Precomputes 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<sup>+</sup>) deepest clocks by logic levels
- Finds the top-10 (or top-N<sup>+</sup>) 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**.



**FIGURE 45.** Clock Panel in zBrowser



**FIGURE 46.** Attributes of Clock

This section describes the following subsection(s):

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

## 10.1 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
```

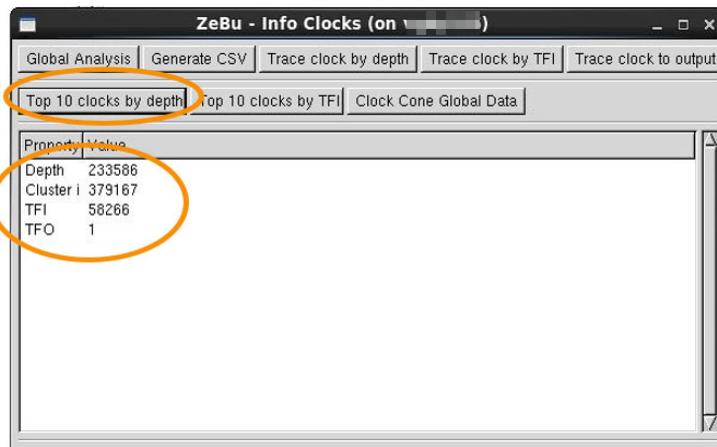
This makes it easier to correlate with RTL register names.

## 10.2 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 47.** 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.



The screenshot shows a window titled "ZeBu - Deepest Clocks (on [REDACTED])". The table has three columns: "Clock Name", "Depth", and "Number of inputs (TFI)". There are 14 rows of data, each representing a different clock signal with its corresponding depth and TFI count. The rows are as follows:

| Clock Name                                                                                          | Depth  | Number of inputs (TFI) |
|-----------------------------------------------------------------------------------------------------|--------|------------------------|
| skl_noze_top_gsrpkq_te_gsrpkq_gsrdie.core0_fe.bpu.bptblqm.lsd_fsms_1_bptblsdfsm.zfast_e391ygrw2gnc1 | 233586 | 58266                  |
| skl_noze_top_gsrpkq_te_gsrpkq_gsrdie.core0_fe.bpu.bptblqm.lsd_fsms_1_bptblsdfsm.zfast_pmcczyry33hc2 | 233586 | 58266                  |
| skl_noze_top_gsrpkq_te_gsrpkq_gsrdie.core1_fe.bpu.bptblqm.lsd_fsms_1_bptblsdfsm.zfast_e391ygrw2gnc1 | 233586 | 58266                  |
| skl_noze_top_gsrpkq_te_gsrpkq_gsrdie.core1_fe.bpu.bptblqm.lsd_fsms_1_bptblsdfsm.zfast_pmcczyry33hc2 | 233586 | 58266                  |
| skl_noze_top_gsrpkq_te_gsrpkq_gsrdie.core0_fe.bpu.bptblqm.lsd_fsms_0_bptblsdfsm.zfast_e391ygrw2gnc1 | 233586 | 58266                  |
| skl_noze_top_gsrpkq_te_gsrpkq_gsrdie.core0_fe.bpu.bptblqm.lsd_fsms_0_bptblsdfsm.zfast_pmcczyry33hc2 | 233586 | 58266                  |
| skl_noze_top_gsrpkq_te_gsrpkq_gsrdie.core1_fe.bpu.bptblqm.lsd_fsms_0_bptblsdfsm.zfast_e391ygrw2gnc1 | 233586 | 58266                  |
| skl_noze_top_gsrpkq_te_gsrpkq_gsrdie.core1_fe.bpu.bptblqm.lsd_fsms_0_bptblsdfsm.zfast_pmcczyry33hc2 | 233586 | 58266                  |
| skl_noze_top_gsrpkq_te_gsrpkq_gsrdie.core0_fe.bpu.bnplpd.zfast_evjsd25lm6y1                         | 233580 | 58193                  |
| skl_noze_top_gsrpkq_te_gsrpkq_gsrdie.core0_fe.bpu.bnplpd.zfast_o2gxnitgp811                         | 233580 | 58193                  |

**FIGURE 48.** Clock Signals

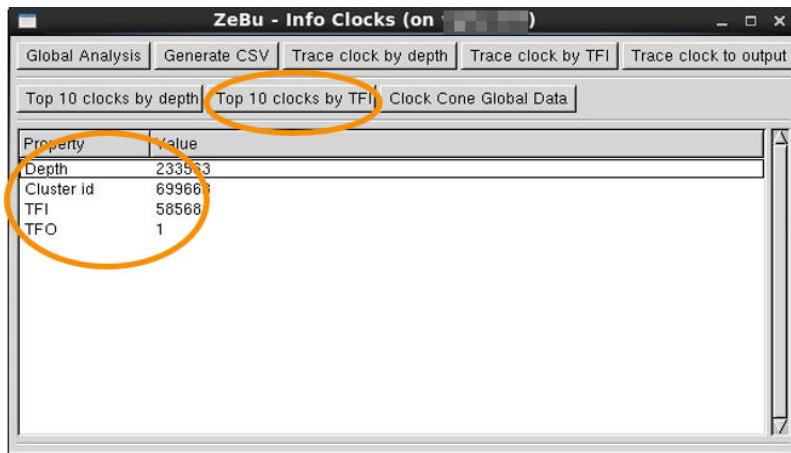
## 10.3 Finding Clocks with Maximum TFI

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**.

## Tracing a Known Clock Signal



**FIGURE 49.** Clocks by TFI

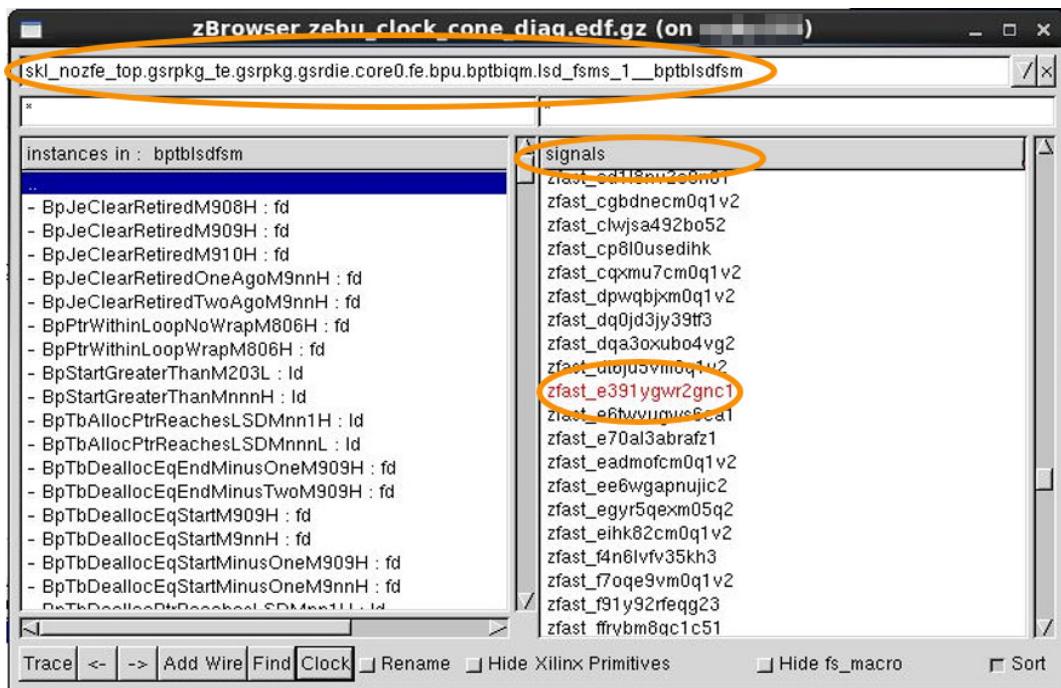
## 10.4 Tracing a Known Clock Signal

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.*

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

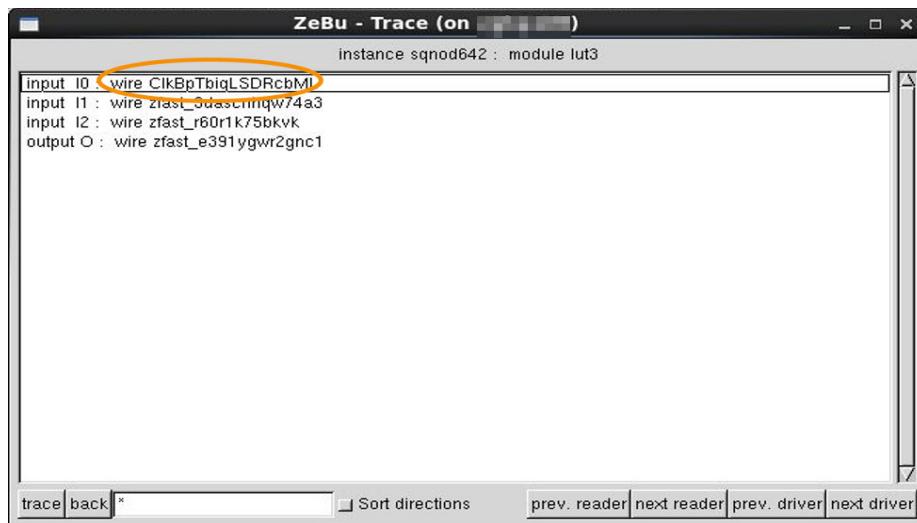


2. Click the “ports/signals” pane to “signals”.
3. Double-click the clock signal to open the trace window.

## Example

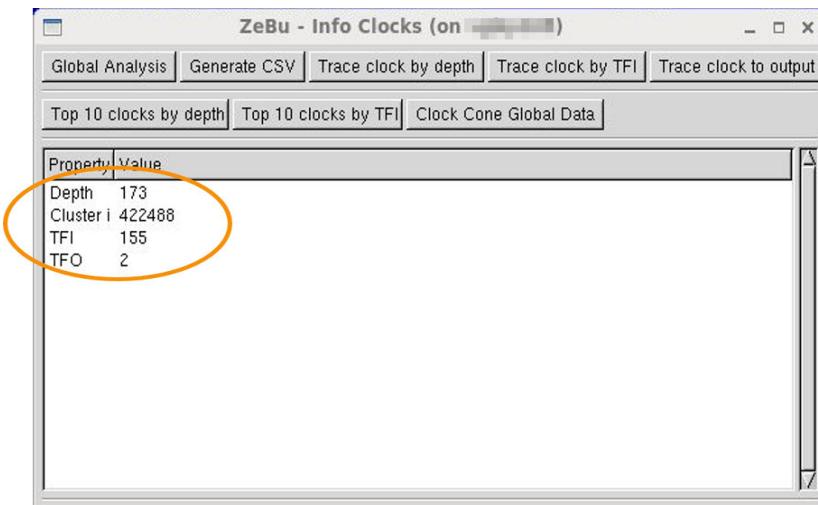
The following figure shows the list of signals available within a module.

## Tracing a Known Clock Signal



**FIGURE 50.** Clock Signals in a Module

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

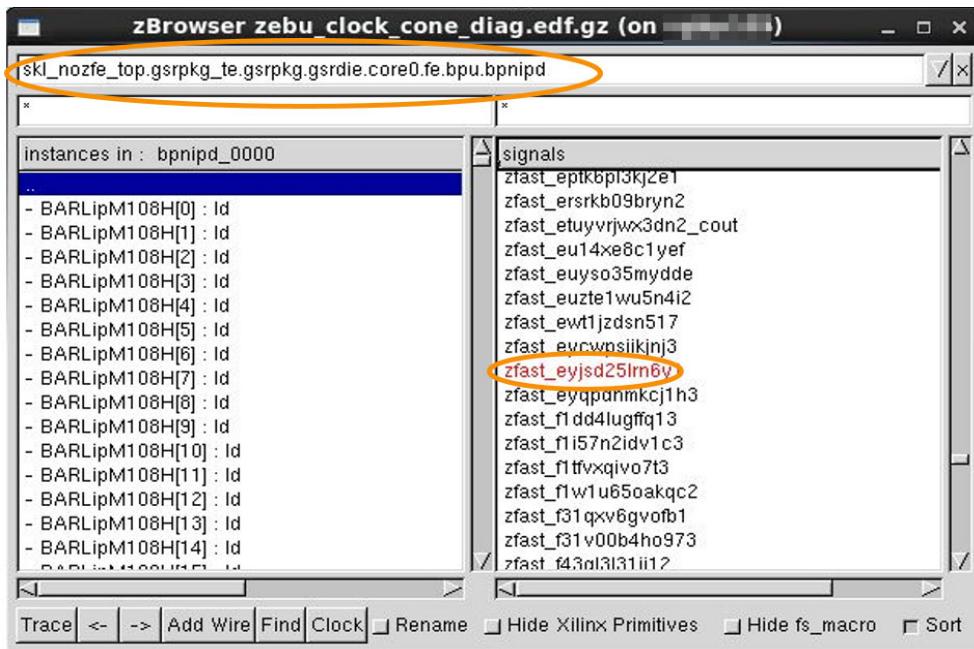


**FIGURE 51.** Clock Attributes

## 10.5 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.

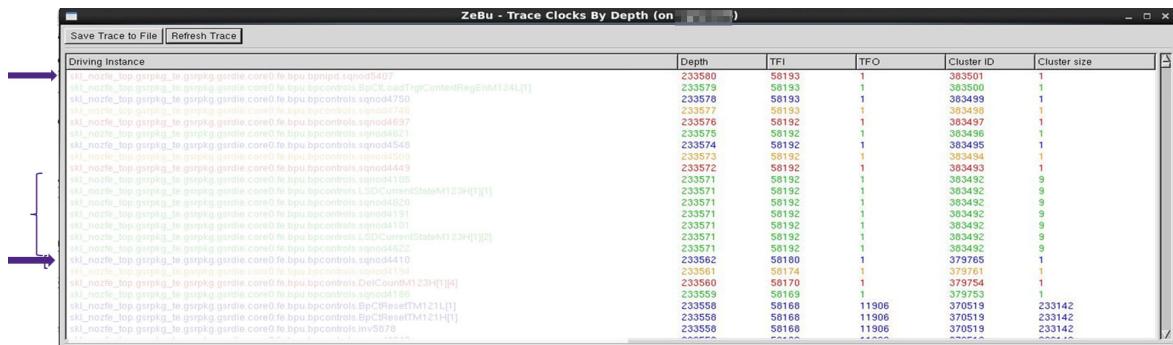


**FIGURE 52.** Clock Signal Contributing to Maximum Logic Level

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

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

## Tracing the Clock Path Contributing to Maximum Logic Levels

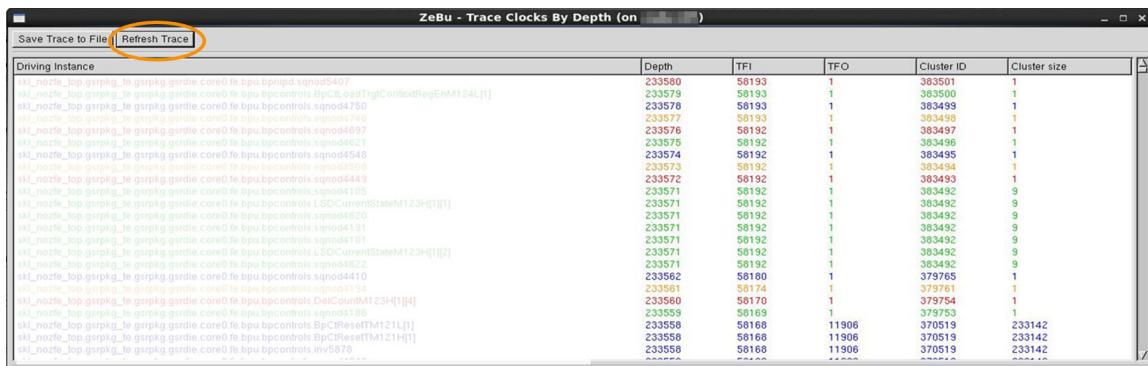


**FIGURE 53.** Maximum Logic Level Path

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 54.** Tracing an Intermediate Point

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



| Driving Instance                                                                     | Depth  | TFI   | TFO   | Cluster ID | Cluster size |
|--------------------------------------------------------------------------------------|--------|-------|-------|------------|--------------|
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.sqnode5407                 | 235580 | 58193 | 1     | 383501     | 1            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.sqnode4750                 | 233579 | 58183 | 1     | 383500     | 1            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.sqnode4746                 | 233578 | 58193 | 1     | 383499     | 1            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.sqnode4697                 | 233577 | 58193 | 1     | 383498     | 1            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.sqnode4621                 | 233576 | 58192 | 1     | 383497     | 1            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.sqnode4548                 | 233575 | 58192 | 1     | 383496     | 1            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.sqnode4547                 | 233574 | 58192 | 1     | 383495     | 1            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.sqnode4449                 | 233573 | 58192 | 1     | 383494     | 1            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.sqnode4105                 | 233572 | 58192 | 1     | 383493     | 1            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.LSDCurrentState1123H[1][1] | 233571 | 58192 | 1     | 383492     | 9            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.sqnode4191                 | 233571 | 58192 | 1     | 383492     | 9            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.sqnode44101                | 233571 | 58192 | 1     | 383492     | 9            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.LSDCurrentState1123H[1][2] | 233571 | 58192 | 1     | 383492     | 9            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.sqnode44522                | 233571 | 58192 | 1     | 383492     | 9            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.sqnode44110                | 233562 | 58192 | 1     | 379756     | 1            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.sqnode44111                | 233561 | 58174 | 1     | 373761     | 1            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.CelCountM123H[1][4]        | 233560 | 58170 | 1     | 379754     | 1            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.sqnode4188                 | 233559 | 58169 | 1     | 379753     | 1            |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.BpCResetTM121H[1]          | 233558 | 58168 | 11906 | 370519     | 233142       |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.BpCResetTM121H[1]          | 233558 | 58168 | 11906 | 370518     | 233142       |
| sd_nozle_top_gprkg_be_gprkg_gndie_cored0_to_bpu_bpcontrol.invS076                    | 233558 | 58168 | 11906 | 370519     | 233142       |

**FIGURE 55.** Saving Tracing to a File**Note**

This file helps you to find instances based on names, e.g. "\*scan\*", "\*dft\*", "\*rst\_sync\*", and so on.

## 10.6 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 56.** Tracing Clock Path Contributing to Maximum IO Cut

Tracing an Intermediate Clock Signal to its Final Output

## 10.7 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 57.** 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.

## 10.8 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.*

## 10.8.1 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
```

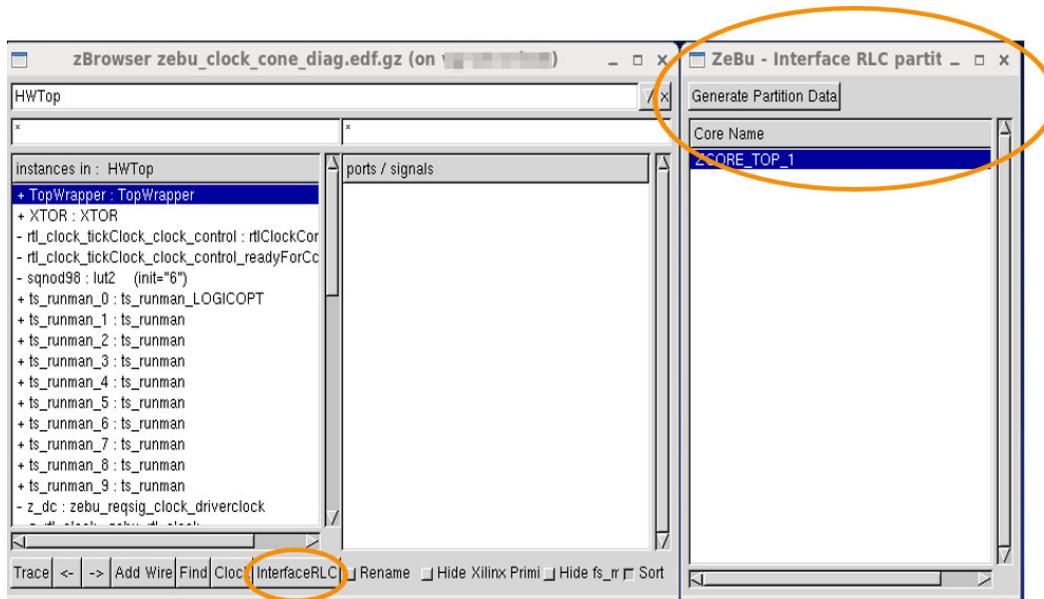
This option helps to study the extent of clock-cone across COREs. The **zBrowser** window refreshes after setting this option.

## 10.8.2 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

## Analyzing Clock Localization

5. Click **Interface RLC** in the **zBrowser** window.  
The **ZeBu Interface RLC** Partition window opens.



**FIGURE 58.** 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 discusses the following topics.

- [\*designFeatures File\*](#)
- [\*Runtime Clock file\*](#)
- [\*Memory Content File\*](#)

## 11.1 designFeatures File

The designFeatures file is an optional file for the ZeBu runtime software. This file contains user settings that override the default settings for emulation runtime.

The ZeBu runtime uses the designFeatures file if it is present in either the directory where the runtime software is launched.

The path to the designFeatures file can also be provided as an argument to the open method, which is called in the testbench.

If a designFeatures file is not found, the default values are retained for emulation runtime.

The designFeatures file contains the following elements:

- *Location of the Calibration Files*
- *Declaring the Process Name*
- *Parameters for Design Clocks*
- *Parameters for Transactors*
- *Initializing Memories*
- *Programming the driverClk Reset Signal*
- *Programming the DUT Reset*
- *Declaration for Smart Z-ICE*
- *Symbolic Parameters for Timing Settings*
- *Reducing Disk Usage With Clock Delay Feature*

### 11.1.1 Syntax Rules

The syntax of the designFeatures file is a set of lines, each containing a parameter setting of the form: <parameter> = <value> ;

The syntax rules for the designFeatures file are as follows:

- Parameter names are case insensitive.
- File names and hierarchical paths are case sensitive.
- Comment lines start with a # character (only at the start of the line).
- <value> can be a decimal number or a string enclosed in double quotes (").

---

## designFeatures File

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`

## 11.1.2 Location of the Calibration Files

The default calibration files reside in the default directory (`$ZEBU_SYSTEM_DIR/calibration`). The path to the calibration files can also be specified using the `$calibrationPath` parameter as follows: `$calibrationPath = ?<path>?;`

## 11.1.3 Declaring the Process Name

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

### 11.1.3.1 Single-process Environment

When the verification environment has a single Linux process, there is no need to declare the process name in the `designFeatures` file; the `default_process` name is used.

This default configuration can be viewed in the `designFeatures.help` template file:

```
$nbProcess = 1;
$process_0 = "default_process";
```

---

### 11.1.3.2 Multi-process Environment

When the verification environment has multiple Linux processes, the process names must be declared.

```
$nbProcess = <nb_process>;  
$process_<ID> = "<process_name>";
```

where,

- <nb\_process> is an integer between 1 and 16 specifying the number of processes.
- <ID> is an integer index between 0 and 15. Processes must be declared with contiguous indexes starting from zero.
- <process\_name> is an arbitrary string used to identify a process in the open method and log files.

For example

```
$nbProcess = 1;  
$process_0 = "myProcess";
```

In the testbench:

```
// open session  
  
Board *board = Board::open ("myZebuWork", "designFeatures",  
"myProcess");  
  
if (board == NULL)  
    exit (1);
```

```
-- ZeBu : cpp_test_bench : Looking for a connection (pid 28200 at  
Tue 3 8 2010 - 17:46:11)  
-- ZeBu : cpp_test_bench : "my_process" is a full-capability  
process working on          ".../zebu.work".
```

---

designFeatures File

### 11.1.3.3 Unnamed Control Processes

A C/C++ testbench that does not control any transactor can execute without declaring its process name; it is known as a control process. The name of a control process may be designated in the open method even though the name is not declared in the designFeatures file.

The control process is used to analyze or control clocks, memories, logic analyzers, and so on. However, the control processes cannot control top-level I/O. The clocks generated by zceiClockPort must be driven by named processes.

For example

```
$nbProcess = 2;
$process_0 = "bench_0";
$process_1 = "bench_1";
```

---

In the testbench:

```
// open session
Board *board = Board::open ("myZebuWork", "designFeatures",
"bench_2");
if (board == NULL)
    exit (1);
```

---

```
-- ZeBu : tb_dut : WARNING : A process name has been specified
"bench_2" at open time, but it is not specified in the "./
designFeatures" file.

-- ZeBu : tb_dut : WARNING : The list of specified process names
is:

-- ZeBu : tb_dut : WARNING : "bench_0"
```

```
-- ZeBu : tb_dut : WARNING : "bench_1"
-- ZeBu : tb_dut : Looking for a connection (pid 11307 at Wed 9 3
2011 - 09:20:51) ...
-- ZeBu : tb_dut : "bench_2" is a control-only process working on
"../zcui.work/zebu.work".
```

### 11.1.3.4 Initialization Phases Not Executed on Memory

In a control-only C/C++ testbench, the memory initialization phase is not executed, which may cause the design might malfunction even if there is no error. To avoid this issue, you must identify the testbench in the `designFeatures` file.

```
$nbProcess = 1;
$process_0 = "<process_name>";
```

### 11.1.4 Parameters for Design Clocks

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

#### 11.1.4.1 Parameters for Primary Clocks

Each design clock is described in the `designFeatures` file by its own parameters, and it can be linked to other clocks in a group.

The syntax for clock parameters is as follows:

## designFeatures File

```

$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";
```

---

 designFeatures File

To declare a frequency ratio of two using the `Waveform` parameter:

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

---

### **Example of DUT Clock Description in free-running Mode**

These clocks are provided by the ZeBu clock generator, and the frequency is fixed.

```
$U0.M0.my_clock.Mode = "free-running";
$U0.M0.my_clock.Frequency = 5000;
```

---

### **Example of DUT Clock Description in `in-situ` Mode**

These clocks are provided externally, and the frequency cannot be altered.

```
$U0.M0.my_clock.Mode = "in-situ";
```

---

### **Syntax for Tolerance**

This parameter can be used to improve runtime performance when multiple primary clocks belong to the same clock group. The performance improvement is more noticeable when:

- The number of primary clocks is higher.
- The declared clock waveforms and their ratios are heterogeneous.
- The clock edges that are very close to each other are processed simultaneously, group-by-group or clock-by-clock.

To activate the `Tolerance` parameter, the following lines must be added for each

group/clock, which needs to be modified:

```
$U0.M0.<group>.Tolerance=<value>;  
$U0.M0.<clock>.Tolerance=<value>;
```

where <value> is a string, enclosed by double quotes, with the following values:

- "no": Default.
- "yes": Clock edges that are very close to one another are merged.

**Note**

- ❑ *Tolerance can be used only with controlled clocks.*
- ❑ *At least two clocks in the same group are required for an effective calculation.*
- ❑ *Only yes/no attributes are accepted.*

All clocks in a clock group have the same tolerance. However, the tolerance of a single clock in the group can be overwritten, as displayed in the following example:

```
#Group setting for group1  
$U0.M0.group1.Tolerance = "yes";  
  
#CLK0 uses the group setting for tolerance  
$U0.M0.CLK0.VirtualFrequency = 8;  
$U0.M0.CLK0.GroupName = "group1";  
  
# CLK1 overwrites the group setting to disable tolerance  
$U0.M0.CLK1.VirtualFrequency = 5;  
$U0.M0.CLK1.GroupName = "group1";  
$U0.M0.CLK1.Tolerance = "no";  
  
# CLK2 uses the group setting for tolerance.  
$U0.M0.CLK2.VirtualFrequency = 2;  
$U0.M0.CLK2.GroupName = "group1";
```

---

## designFeatures File

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.*

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).

-- 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` +  $\Delta$ , where  $\Delta$  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,

---

designFeatures File

- <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.

## 11.1.5 Parameters for Transactors

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

### 11.1.5.1 Global Transactor Settings

By default, the communication infrastructure for transaction-based verification uses a burst mode that is applied to all the ports of a transactor.

The following declaration can be used to modify the settings for the burst mode:

```
$xTor.burstMode = "high | medium | disable";
```

---

By default, the high burst mode is active, if the host computer supports it.

If the host computer does not support burst mode, the ZeBu runtime (**zServer**) automatically disables it (equivalent to disable value).

In such a case, try the medium value since it may improve the communication performance for large messages. For small-size messages, the disable value is recommended.

## 11.1.5.2 Specific Transactor Settings

The performance of transactor ports does not depend only on the port sizes. It is because big ports are used rarely while small ports are used more often. For this reason, the size of the buffer that receives port data from the hardware can be modified according the number of messages received in the buffer.

Performance is optimized when a frequently active port has multiple port values (messages) in its buffer. The current `$<driverName>::<portName>.nbMessMax` is given in the `designFeatures.<hostname>.help` file. The `nbMessMax` parameter specifies the maximum number of port-values (messages) to be stored in the buffer.

The `nbMessMax` setting is specific to each transactor port and it must consider the transactor's memory requirements:

```
| $<driverName>::<portName>.nbMessMax = <value>;
```

---

## 11.1.6 Initializing Memories

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

### 11.1.6.1 List of Memory-Content Files

A file containing a list of design memory instances that must be initialized with a specific content can be specified using the `$memoryInitDB` parameter. Each memory is initialized with its corresponding content-file. This initialization is applicable to any memory, irrespective of its physical implementation (LUTs, registers, BRAM, and so on).

This specification is done as follows:

```
| $memoryInitDB = "path_to_memory_init_list";
```

---

The path can be absolute or relative to the current directory. The memory load occurs in the `Board::open()` function.

For example

Declaration of the file for memory initialization:

```
| $memoryInitDB = "~/mydesign/init_mem1.mem";
```

---

---

designFeatures File

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;
```

---

**Note**

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

---

### 11.1.7 Programming the driverClk Reset Signal

A programmable reset signal driven by driverClk is available for transaction-based verification. The duration of the reset state is defined as the number of driverClock cycles. It should only be set when a transactor BFM requires multi-cycle assertions of reset.

```
$driverClk.Reset = <value>;
```

---

where <value> is the number of driverClk cycles for which reset is active (Default: 0).

**Note**

*It is recommended not to specify the parameter (comment with #). But, for transactors developed with ZEMI-3, you must set it to a non-zero duration: \$driverClk.Reset = 1;*

---

### 11.1.8 Programming the DUT Reset

You can define a reference clock to program the DUT reset. The active level of the

reset is derived from the selected output of the `zceiClockPort` instantiation, that is, reset (active high) or reset (active low).

The waveform of the reset signal can be programmed using the following parameters:

```
$tgClk.ResetInactive = <nb_inactive_cycles>;  
$tgClk.ResetActive = <nb_active_cycles>;  
$tgClk.SelectResetClkName = "U0.M0.<my_clock>";
```

## 11.1.9 Declaration for Smart Z-ICE

For a design using the Smart Z-ICE interface, the following line should be added to relocate a Z-ICE connector at runtime:

```
$smartZICE.connectorRemap_<i> = <j>;
```

where,

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

Only the connectors of the unit connected to the host computer can be used with the Smart Z-ICE interface.

For example

When the design is compiled for connectors 0 and 1 but the connectors actually used are connectors two and three, add the following in the `designFeatures` file:

```
$smartZICE.connectorRemap_0 = 2;  
$smartZICE.connectorRemap_1 = 3;
```

The following declarations are present in the `designFeatures.<hostname>.help` file:

```
$smartZIce.data  
$smartZIce.vcc  
$smartZIce.clock
```

Comments are automatically set from the compilation and should not be modified in the actual `designFeatures` file.

## 11.1.10 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.
- 

### 11.1.11 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 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.

## 11.2 Runtime Clock file

The clock file is an optional input to the Zebu runtime. It is an alternative to specifying the clocks inside the designFeatures file. For information about specifying the clock parameters in a separate file, see [Using a Separate Clock File](#).

## 11.2.1 Clock File Content

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

## 11.2.2 Clock File Syntax

The clock file is a text file whose syntax is described by the following Backus Naur Form (BNF):

```

file_txt :
    line_txt
    | file_txt line_txt

line_txt :
    '\n'
    | mode '\n'
    | waveform '\n'
    | virtual_frequency '\n'
    | frequency '\n'
    | group_name '\n'
    | Tolerance '\n'
    | delayClkEdge_<x> '\n'

mode :
    '$'CLOCK_NAME.mode '=' """mode_type""";'

waveform :
    '$'CLOCK_NAME.Waveform '=' """WAVEFORMSET""";'

virtual_frequency :
    '$'CLOCK_NAME.VirtualFrequency '=' NUMBER ';'

frequency :

```

```
'$'CLOCK_NAME.Frequency '=' NUMBER ';' '
group_name :
    '$'CLOCK_NAME.GroupName '=' """STRING"""; '
mode_type :
    controlled
    | free-running
    | in-situ
Tolerance_declaratiion :
    '$'GroupName.Tolerance '=' """toleranceSpecifier"""; '
    | '$'CLOCK_NAME.Tolerance '=' """toleranceSpecifier"""; '
toleranceSpecifier :
    yes
    | no
delayClkEdge_<x> :
    "NbDriverClk" NUMBER "FROM" '{' list_of_edges '}'
                                "TO" '{' list_of_edges '}'; '
list_of_edges :
    edge_declaration
    | list_of_edges, edge_declaration
edge_declaration :
    """edgeSpecifier"" "( '"""clockIdentifier"" ') "
    | """edgeSpecifier"" "( '*' ') '
edgeSpecifier :
    RISE
    | FALL
    | BOTH
```

where,

---

Memory Content File

- 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]`

## 11.3 Memory Content File

Memory content files are used to initialize, upload, or download memories. Memory content files work with any memory, irrespective of its physical implementation (LUTs, registers, BRAM, and so on).

### 11.3.1 ZeBu-Proprietary Binary Format

The ZeBu-proprietary binary format for memory content files is not readable. It can be created through a dedicated interface, `hex2bin`.

It is possible to have a 24-character header at the beginning of the binary memory content file to describe the initialization address range. This file is commonly named with the `.bin` extension (you must not use `.raw` extension, which is reserved for the binary file with no header).

When a binary file has no header (binary file generated out of emulation, or an executable file to be downloaded into a ROM), the file name must be named with the `.raw` extension for automatic detection at runtime.

### 11.3.2 Memory Binary Format for Shorter Load Time

A new binary sparse memory file format enables you to reduce the time taken for loading the memories.

The new compact files can be generated with the `hex2bin` tool and can be loaded using `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.

## Example

```
hex2bin -i hex.0.0.0 -o bin0.0.0 -m top.dut.mem00 -p ../zcui.work/
zebu.work -s -f 1000 -a 64
```

---

In addition, **hex2bin** proposes a new option to accept when the input file content goes beyond the last address of the memory.

- -t: Do not generate an error if the content of the input file goes beyond the last address of the memory

### 11.3.3 Memory Content File Text Format

The text format of a memory content file is a description of the memory content for each address or address range. Each line starts with the address or the address range and lists the corresponding data as follows:

```
@<address> : <data>
```

---

Or

```
@<start_addr>,<stop_addr> : <data>
```

---

where,

- The colon that separates address and <data> is optional.
- The comma between <start\_addr> and <stop\_addr> is mandatory.
- The values for <address> and <data> are specified in hexadecimal radix ('h, 'H or 0x before the value), binary radix ('b or 'B before the value) or decimal radix ('d or 'D before the value).
- The default radix for address and data values is hexadecimal.

Memory Content File

---

- 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.

### 11.3.3.1 Example

The following file initializes a 128-byte memory (16-bit words):

```
// Initialize all the memory with value 0x44
@0, FFFF : 44

// Load address 0 with value 0xBA00
@0 : ba00

// Load address 2 with 0xBB02
@2 : BB02

//Load address 3 with 0xCC03 and address 4 with 0xDD04
CC03
DD04

// Switch to binary mode by default for data
'b

// Load address 5 with 0x1234 in binary with legibility separator
@5: 16'b0001_0010_0011_0100

// Load address 6 with a decimal value (will be 0x10 in the memory)
16

//Switch to hexadecimal by default
'h

//Load address 7 with some undetermined bits (z)
@7 0x01z7

// Load range from address FF00 to FF03 with value 0xA
@FF00,FF03: A
```

---

---

Memory Content File

The actual content of a memory initialized with this example file is described in the following table:

| <b>Address</b> | <b>Value</b> |
|----------------|--------------|
| 0000           | BA00         |
| 0001           | 0044         |
| 0002           | BB02         |
| 0003           | CC03         |
| 0004           | DD04         |
| 0005           | 1234         |
| 0006           | 0010         |
| 0007           | 0107         |
| .....          | .....        |
| FEFF           | 0044         |
| FF00           | 000A         |
| FF01           | 000A         |
| FF02           | 000A         |
| FF03           | 000A         |
| FF04           | 0044         |

