ZeBu® Server

Getting Started Guide

Version O-2018.09-SP1, June 2019

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at *http://www.synopsys.com/company/legal/trademarks-brands.html*.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Software Licensing Notices

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

Third-Party Links

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

www.synopsys.com

Contents

Preface 11
. About This Book 11
Contents of This Book 11
Related Documentation 12

1. ZeBu Server 13
1.1. Overview Of ZeBu Server 14
1.1.1. Features 14
1.1.2. Hardware Overview 15
1.2. ZeBu Emulation Flow 21
1.3. Introduction to UTF 22

2. Assembling a Design for Emulation 25
2.1. Mapping Your Design and Test Environment 26
2.1.1. Importing a Design From a Simulation Environment 26
2.1.2. Preparing a DUT-only Environment 27
2.2. Clock Modeling 27
2.2.1. zceiClockPort Primitive 27
2.2.2. Clock Delay Support 28
2.2.3. clockDelayPort Primitive 29
2.3. Memory Modeling 29
2.3.1. Memory Inference From a Design 29
2.3.2. Memory Model IP 31
2.4. Accessing Signals During Runtime 32
2.4.1. Reading Signals 32
2.4.2. Forcing and Injecting Values 33
2.5. Verilog System Tasks 34

3. Compilation 39
3.1. Overall Unified Compile Flow 40
3.1.1. Basic UTF Commands 40
3.2. Compilation Script for VCS Unified Use Model 41
3.2.1. Setting Up VCS 42

3.2.2. Setting up the synopsys_sim.setup File 42
3.2.3. Setting up Compilation Commands Using UTF....43
3.2.4. VCS Options 43
3.2.5. Compiling With DesignWare Building Blocks 46
3.3. Specifying the Settings in ZeBu Unified Compile Script.... 47
3.4. Using a Compute Farm 48
3.4.1. Configuring the Job Queuing System 48
3.4.2. Using Sun™ Grid Engine or Platform LSF® 50
3.5. Compiling a Project 51
3.5.1. Compiling With zCui in the Batch Mode 51
3.5.2. Compiling With zCui in the GUI Mode 52
3.6. Compilation Analysis 56
3.6.1. Important Log Files....56
3.6.2. Analyzing Compilation Results Using zBatchExplorer....58
3.7. Analyzing Compilation Health Using zHealth 60

4. Controlling Emulation Runtime 63
4.1. Controlling Runtime Parameters 64
4.1.1. Emulation Speed 64
4.1.2. Setting Design Clock Parameters for Runtime 64
4.1.3. Memory Initialization for Runtime 65
4.2. Using zRun to Control Emulation Runtime 66
4.2.1. Common zRun Command-Line Options 66
4.2.2. Controlling Clocks During Runtime Using zRun....68
4.2.3. Managing Memories 74
4.3. Runtime Performance Analysis With zTune 78

5. Using SystemVerilog Assertions at Runtime....79
5.1. Enabling SVA Through assertion_synthesis UTF Command.... 79
5.2. Controlling Synthesized Assertions at Runtime 80
5.2.1. Controlling Assertions From the C++ Testbench....80
5.2.2. Controlling Assertions Using zRun 82
5.2.3. Post-Processing SVA 82

6. DPI Synthesis Support 85
6.1. Compilation UTF Commands for DPI Synthesis.... 85
6.2. Controlling DPI Function Calls at Runtime.... 86

6.3. Example: Importing Function Calls in C and SystemVerilog..... 87

List of Figures

ZeBu Server 3: 2-slot Front and Rear Panels 15
ZeBu Server 3: 5-slot Front and Rear Panels 16
Architecture of an FPGA Module in ZeBu Server 3 18
ZeBu Server 4: Front Panel 19
Architecture of an FPGA Module in ZeBu Server 4 20
ZeBu Emulation Flow 21
Importing a DUT and Test Environment From a Simulation Environment .. 26
Preparing a DUT-Only Environment 27
zCui Default View 52
zCui Options 53
zCui GUI 54
Project Messages Panel 54
Launching Compilation in zCui 55
Viewing the VCS Task in zCui 55
Opening a Directory using zBatchExplorer 58
Selecting Log File and zCui Global File 59
Example of HTML Report Generated by zHealth 61
System Verilog Assertion Panel 82

List of Tables

Design Capacity for ZeBu Server 3 16
FPGA Type and FPGA Module Types in ZeBu Server 3 17
FPGA Type and FPGA Module Types in ZeBu Server 4 20
UTF Commands Categories 22
UTF Help Commands 23
Memories and Memory Preferences 30
Supported Verilog System-Tasks 34
Basic UTF Mandatory Commands 40
VCS Options 44
Sun Grid Engine and Platform LSF Recommendation 51
zHealth HTML Report Sections 60

Preface

About This Book

The *ZeBu® Server Getting Started Guide* introduces the Synopsys emulator system, ZeBu Server. This guide helps you to perform the following:

- Preparing your design for emulation
- Compiling the design
- Controlling the emulation runtime

This guide also provides the information about the tools used to analyze the emulation performance.

Contents of This Book

The *ZeBu® Server Getting Started Guide* has the following chapters:

Chapter	Describes...
ZeBu Server	Introduces ZeBu Server and its features
Assembling a Design for Emulation	Describes how to prepare your design for emulation before compilation
Compilation	Describes how to compile your design in ZeBu
Controlling Emulation Runtime	Describes how to control ZeBu runtime
Runtime Performance Analysis With zTune	Introduces **`zTune`**, a tool to analyze emulation performance
Using SystemVerilog Assertions at Runtime	Describes SystemVerilog assertions for functional verification
DPI Synthesis Support	Describes DPI imported function calls supported in ZeBu

Related Documentation

Document Name	Description
ZeBu Server 4 Site Planning Guide	Describes planning for ZeBu Server 4 hardware installation.
ZeBu Server 3 Site Planning Guide	Describes panning for ZeBu Server 3 hardware installation.
ZeBu Server Site Administration Guide	Provides information on administration tasks for ZeBu Server 3 and ZeBu Server 4. It includes software installation.
ZeBu Server Getting Started Guide	Provides brief information on using ZeBu Server.
ZeBu Server User Guide	Provides detailed information on using ZeBu Server.
ZeBu Server Debug Guide	Provides information on tools you can use for debugging.
ZeBu Server Debug Methodology Guide	Provides debug methodologies that you can use for debugging.
ZeBu Server Unified Command-Line User Guide	Provides the usage of Unified Command-Line Interface (UCLI) for debugging your design.
ZeBu Server Functional Coverage User Guide	Describes collecting functional coverage in emulation. For VCS and Verdi, see the following: - Coverage Technology User Guide - Coverage Technology Reference Guide - Verification Planner User Guide - Verdi Coverage User Guide and Tutorial For SystemVerilog, see the following: - SystemVerilog LRM (2017)
ZeBu Server Power Estimation User Guide	Provides the power estimation flow and the tools required to estimate the power on a System on a Chip (SoC) in emulation. For SpyGlass, see the following: - SpyGlass Power Estimation and Rules Reference - SpyGlass Power Estimation Methodology Guide - SpyGlass GuideWare2018.09 - Early-Adopter User Guide
ZeBu Verdi Integration Guide	Provides Verdi features that you can use with ZeBu. This document is available in the Verdi documentation set.
ZeBu Server LCA Features Guide	Provides a list of LCA features available with ZeBu Server.
ZeBu Server Release Notes	Provides enhancements and limitations for a specific release.

1 ZeBu Server

This chapter presents an overview and the features of the Synopsys emulator, ZeBu Server. This chapter explains the ZeBu Server in the following topics:

- *Overview Of ZeBu Server*
- *ZeBu Emulation Flow*
- *Introduction to UTF*

1.1 Overview Of ZeBu Server

ZeBu Server is a very high-capacity emulator system with easy setup and debugging capabilities.

ZeBu Server 3 is available in two slots and five slots to handle designs up to 60– 300 million ASIC-equivalent gates and it allows you to connect two users in a two-slot system and up to five users in a five-slot system.

ZeBu Server 3 is also expandable in a multi-unit environment to accommodate up to 49 users (by combining 10 units of a five-slot system) to handle designs up to three billion ASIC-equivalent gates.

ZeBu Server 4 can handle designs up to 20 billion ASIC-equivalent gates.

1.1.1 Features

The salient features of ZeBu Server are as follows:

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

    NOTE: *The Direct ICE interface is supported only with ZeBu Server 3.*
  - The Smart Z-ICE interface provides support for standard software debuggers using JTAG cables.

1.1.2 Hardware Overview

As already mentioned, ZeBu Server 3 is available in both two-slot and five-slot systems.

Note *A two-slot ZeBu Server 3 is not intended to be part of a multi-unit environment.*

1.1.2.1 ZeBu Server 3

In this section, an overview of two types of ZeBu Server 3 is provided.

Two-Slot ZeBu Server 3

The two-slot ZeBu Server 3 consists of the following elements:

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

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

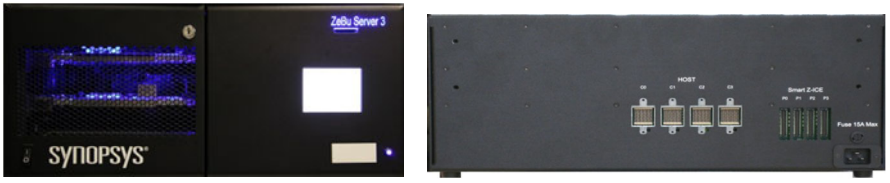

FIGURE 1. ZeBu Server 3: 2-slot Front and Rear Panels

Five-Slot ZeBu Server 3

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

- Five slots to plug-in up to five FPGA modules.
- A backplane equipped with the following components:

- ❒ PCIe interface (10 connectors to link up to five PCs, four units, and a hub).
- ❒ Smart Z-ICE interface (Five connectors with 80 data pins and five clock pins).

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

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

Note *Single-unit systems are not compatible with multi-unit systems.*

Design Capacity

The following table lists the design capacity for each ZeBu Server 3 unit.

TABLE 1 Design Capacity for ZeBu Server 3

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

FPGAs in ZeBu Server 3

ZeBu Server 3 is a scalable system that maps a DUT into Xilinx Virtex-7 LX2000 FPGA.

The ZeBu testbench is mapped to dedicated Interface FPGAs (IF) that implements Reconfigurable Testbench (RTB). It is a patented interface technology that incorporates proprietary hardware, firmware, and software layers.

The following table lists FPGA module types supported in ZeBu Server 3.

TABLE 2 FPGA Type and FPGA Module Types in ZeBu Server 3

FPGAs Type	FPGA Module	Description
Xilinx Virtex-7 LX2000	9F	• Four FPGAs with 512 MB DDR3 memory for each FPGA • One FPGA with 2x 8 GB DDR3 memory • Four FPGAs with no additional memory • One FPGA dedicated to the RTB
	9F/ICE	• Four FPGAs with 512 MB DDR3 memory for each FPGA • One FPGA with 2x 8 GB DDR3 memory • Four FPGAs with no additional memory • One FPGA dedicated to the RTB • Direct In-Circuit Emulation (ICE) interface (568 data pins and dedicated clock pins) to connect a target system

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

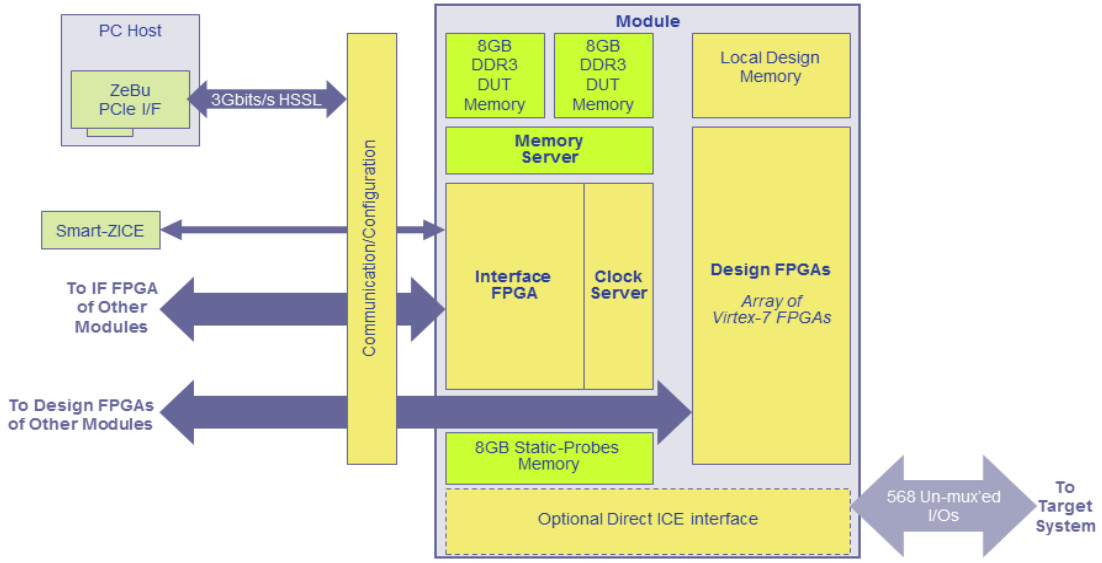

FIGURE 3. Architecture of an FPGA Module in ZeBu Server 3

1.1.2.2 ZeBu Server 4

ZeBu-Server 4 extends the scaling capabilities of ZeBu Server and it can be a standalone or a part of a multi-unit environment. The new architecture is designed to ease the connection of additional emulation resources and additional Host PCs.

ZeBu Server 4 provides two times the emulation performance to enable System-on-Chip (SoC) verification and software bring-up, and to address the exploding verification requirements of automotive, 5G, networking, Artificial Intelligence (AI), and data center SoCs.

ZeBu Server 4 also provides five times lower power consumption with half data center footprint. In addition, ZeBu Server 4 delivers software innovation for faster compile, advanced debug, power analysis, simulation acceleration, and hybrid emulation.

The ZeBu Server 4 hardware is designed to be deployed into 27U or 42U rack cabinets.

For example:

- Two ZeBu Server 4 units can be fitted into a 27U cabinet.
- Up to four ZeBu Server 4 units can be fitted into a 42U cabinet.

The following figure displays the front panel of ZeBu Server 4.

FIGURE 4. ZeBu Server 4: Front Panel

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

Design Capacity

For details on design capacity of ZeBu Server 4, see the *ZeBu Server 4 Site Planning Guide*.

FPGAs in ZeBu Server 4

ZeBu Server 4 is a scalable system that maps a DUT into Xilinx Ultrascale XCVU440 FPGA.

The following table lists FPGA module types supported in ZeBu Server 4.

TABLE 3 FPGA Type and FPGA Module Types in ZeBu Server 4

FPGAs Type	FPGA Module	Description
Xilinx Ultrascale XCVU440	12F	• 2 FPGAs with 8GB DDR3 memory to map the design • 6 FPGAs with 2x1GB of DDR3 memory to map the design • 4 FPGAs with no additional memory

The following figure displays the architecture of an FPGA module within ZeBu Server 4.

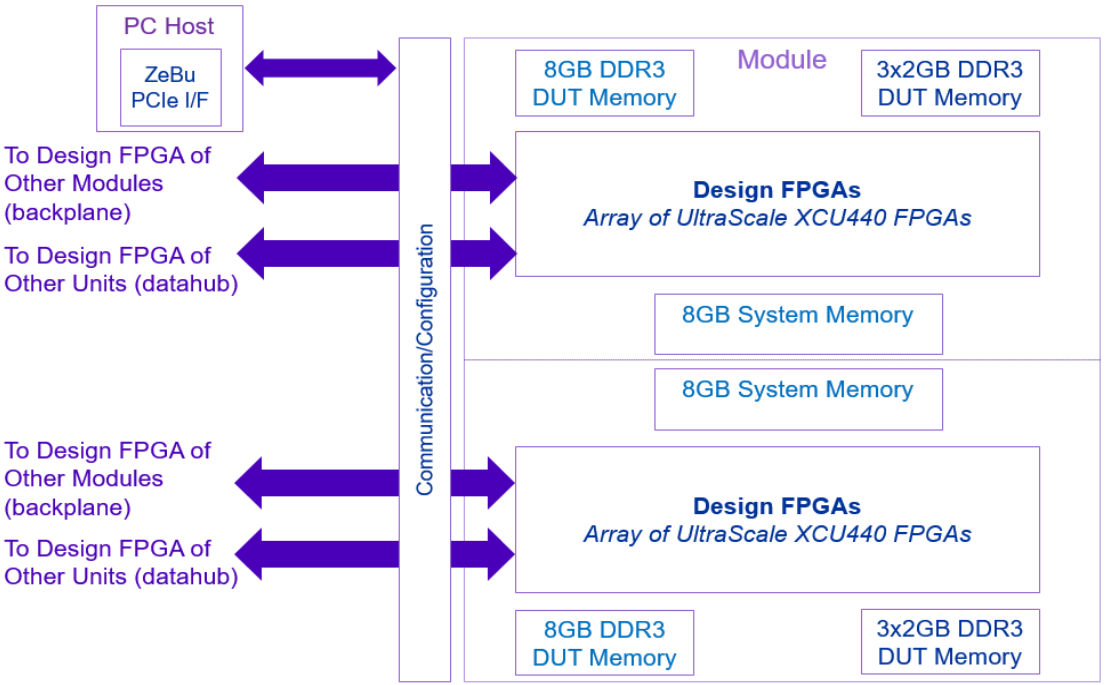

FIGURE 5. Architecture of an FPGA Module in ZeBu Server 4

1.2 ZeBu Emulation Flow

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

The Unified Compile ZeBu project file, that is a UTF file, contains compilation parameters. ZeBu compilation is managed by the zCui application that can be launched in graphical or batch mode.

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.

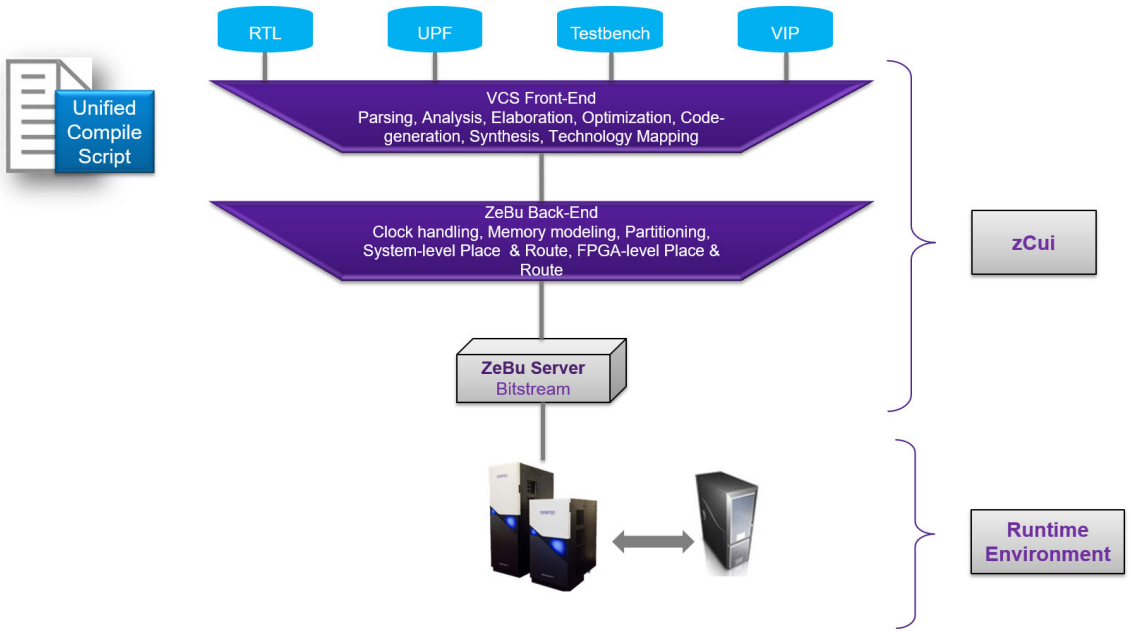

FIGURE 6. ZeBu Emulation Flow

The ZeBu emulation flow consists of the following steps:

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

1.3 Introduction to UTF

The Unified Tcl Format 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.

TABLE 4 UTF Commands Categories

Settings	Description	Requirement
VCS compile commands	• Launches the VCS compiler. • Declares the hardware top. • Sets the remote command for VCS. • Optionally declares any instance of a design that should not be synthesized.	Mandatory
HW configuration	Specifies the .tcl file for the hardware architecture configuration (This .tcl file is generated during ZeBu installation).	Mandatory
Front-end compilation	Executes memory inference options and forces black box.	Optional –uses ZeBu default settings
Back-end compilation	Executes clustering parameters and clock handling options.	Optional –uses ZeBu default settings
`zCui` compilation	Provides additional remote commands and number of jobs allowed in parallel.	Optional –uses ZeBu default settings
Debug options	Provides Combinational Signals Accessibility (CSA) and post-run debug activation.	Optional

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

TABLE 5 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.

For an example of the commands, see *Specifying the Settings in ZeBu Unified Compile Script*.

2 Assembling a Design for Emulation

This chapter describes how to assemble your design for emulation before compilation. It contains the following sections:

- *Mapping Your Design and Test Environment*
- *Clock Modeling*
- *Memory Modeling*
- *Accessing Signals During Runtime*
- *Verilog System Tasks*

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 the 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 a transactor instantiated in a top module.

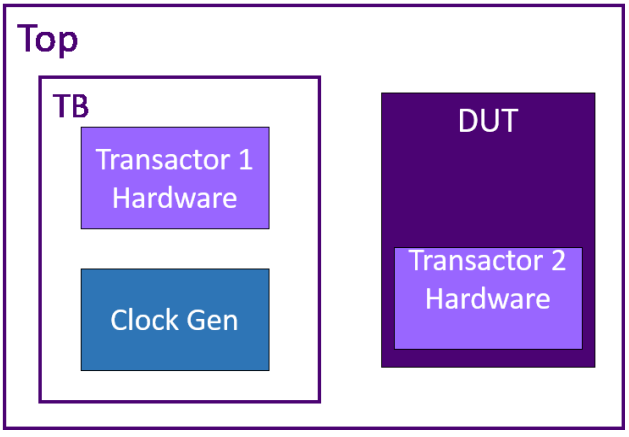

FIGURE 7. Importing a DUT and Test Environment From a Simulation Environment

2.1.2 Preparing a DUT-only Environment

When a DUT and the testbench are available as separate elements, you must create a test environment for the DUT, include necessary transactors, clock generators and connect elements.

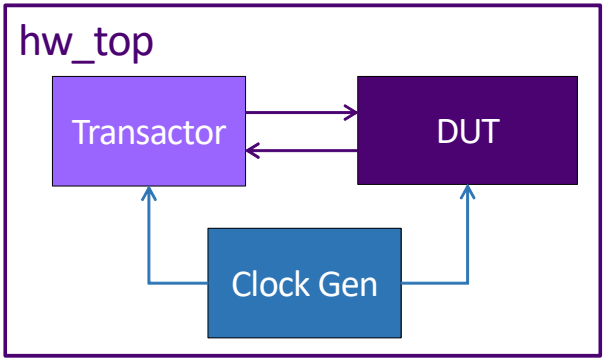

FIGURE 8. Preparing a DUT-Only Environment

This necessary enclosing scope is called the hardware top module, which is a SystemVerilog/Verilog module without any ports.

2.2 Clock Modeling

Primary clocks can be modeled with predefined clock generation primitives or reusing clock generation schemes present in the design environment.

2.2.1 zceiClockPort Primitive

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 the `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*.

2.2.2 Clock Delay Support

You can enable the synthesis of `verilog #delay` in procedural blocks to reuse existing clock generation schemes in DUT or in transactors codes using the following UTF command:

```
clock_delay [-module {list_of_rtl_clock_module}]
```

ZeBu can handle such procedural blocks as follows:

```
always #5 clk=~clk;
```

```
initial begin
  resetn = 0;
  #mydelay;
  resetn = 1;
end
```

The `verilog #delay` synthesis is compatible with the usage of `zceiClockPort` primitive.

Note *There is no limitation in terms of number of clocks that can be synthesized.*

2.2.3 clockDelayPort Primitive

Primary clock can be modeled using `clockDelayPort` primitive.

The `clockDelayPort` primitive is based on clock delay support and is introduced to use like `zceiClockPort` primitive and eliminates the limitation on number of clocks.

A clock generation can be modeled using the `clockDelayPort` primitive as follows:

```
clockDelayPort ClockPort (.clock(clk), .reset(rstn));
```

Many parameters are available to control the clock signal shape (duty cycle, phase) and the reset signal shape (initial value, duration).

2.3 Memory Modeling

ZeBu Server instantiates SRAM-type memory models, which are generated by the memory-generator of the ZeBu compiler. These generated memory 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 subsections:

- *Memory Inference From a Design*
- *Memory Model IP*

2.3.1 Memory Inference From a Design

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

- Small-size memory implemented using registers (up to 1 KB) or Lookup Table (LUT) RAMs (also known as Distributed RAMs) (1KB - 10 KB) in Xilinx Virtex FPGAs.

- Medium-size memory (10 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` UTF commands.

TABLE 6 Memories and Memory Preferences

memories	
`-flops -instance #instances`	Specifies memories to be implemented as flops
`-zmem -instance #instances`	Specify memories to implement as zMems
`-zmem_size_threshold <int> (default int=2048)`	Specifies size threshold for flop versus zMem memories for automatic memory inference
`-zmem_port_threshold <int> (default int=128)`	Specifies port threshold for flop versus zMem memories for automatic memory inference
`-drop_write_only <bool>`	Drop write-only memories
memory_preferences	
`-ramlut_to_bram_threshold|-lutram_to_bram_threshold <integer> (default integer=11)`	Specifies ratio (percentage) of RAM LUTs to BRAMs or restore to default the previously set value

2.3.1.1 Guidelines for Performance

Memory performance improvement 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 a high number of ports. Therefore, the code must be reviewed to understand the source of the ports and

change the 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 needed by 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 must stop on every memory active clock edge, which impacts 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 using a synchronous port or use following synthesis switch:
    - In a Tcl file, add `-syncMemPortRetime`
    - In the UTF file, add `synthesis -wls_option_file {path_to_tcl_file}`
- **Write ports:** Recommended coding style
  - Asynchronous write ports

    Avoid when possible as they may be slower.
  - Read-modify-Write ports

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

2.3.2 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 transactors).
- 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 obtain access to the memory interface. In addition, you must use the `load_edif` command to load the EDIF description of the memory model.

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

2.4 Accessing Signals During Runtime

This section describes how to access signals during runtime.

2.4.1 Reading Signals

Any sequential signal can be read at runtime if it is not optimized during compilation.

Combinational signals can be made readable at runtime by adding dynamic-probes.

The following UTF command adds a dynamic-probe to ensure that the signal can be readable at runtime:

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

When this command is applied to a sequential signal, it ensures that the signal is not optimized.

Sequential signals or combinatorial signals on which a dynamic-probe is added can be read using **zRun**, **zRci** or C/C++ API described in the Board.hh file or the ZEBU_Board.h header files.

2.4.2 Forcing and Injecting Values

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

Injecting a signal means setting a user-defined value at runtime. However, the value is overwritten by the value defined by the design as soon as it changes.

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

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

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

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

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

The runtime control of the signals designated by `zforce` and `zinject` commands is possible using **zRun**, **zRci** or C/C++ APIs.

2.5 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 and ZEMI-4 transactors.

For information about ZEMI-3 and ZEMI-4, see the *ZeBu Server User Guide*.

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

TABLE 7 Supported Verilog System-Tasks

Task	Supported in ZEMI3/ZEMI4	Supported in Design	Default
Simulation Tasks			
`$finish (always block)`	No	Yes	No
Simulation Time Functions			
`$realtime`	No	No	-
`$time`	Yes	Yes	No (clock_delay)
Timescale Tasks			
`$timeformat`	No	No	-
Conversion Functions			
`$bitstoreal, $realtobits`	No	No	-
`$signed, $unsigned`	Yes	Yes	Yes
`$cast (statically determined)`	Partial	Partial	Yes
Array Query Functions			
`$unpacked_dimensions, $dimensions, $left, $right, $low, $high, $size, $increment`	Yes	Yes	Yes
Math Functions			

TABLE 7 Supported Verilog System-Tasks

Task	Supported in ZEMI3/ZEMI4	Supported in Design	Default
`$clog2, $asi, $ln, $acos $log10, $atan, $atan2, $tanh $exp, $sqrt, $hypot, $pow $sinh, $floor, $cosh, $ceil $sin, $asinh, $cos, $acosh $tan, $atanh`	No	No	-
Bit Vector System Functions			
`$countbits`	No	No	-
`$countones, $onehot, $onehot0`	Yes	Yes	Yes
`$isunknown(in SVA)`	Partial	Partial	-
Severity Tasks			
`$fatal, $error, $warning, $info`	No	No	-
Elaboration Tasks			
`$fatal, $error, $warning, $info`	Yes	Yes	Yes
Assertion Control Tasks			
`$asserton, $assertoff, $assertkill`	Yes	Yes	Yes
Sampled Value System Functions			
`$rose, $fell, $stable, $changed, $past`	Yes	Yes	Yes
Coverage Control Functions			
`$coverage_control,$coverage_get_max, $coverage_get, $coverage_merge, $coverage_save, $get_coverage, $set_coverage_db_name, $load_coverage_db`	No	No	-

TABLE 7 Supported Verilog System-Tasks

Task	Supported in ZEMI3/ZEMI4	Supported in Design	Default
Probabilistic Distribution Functions			
`$random($memset)`	Partial	Partial	-
`$dist_chi_square, $dist_erlang, $dist_t, $dist_exponential, $dist_normal, $dist_poisson, $dist_uniform`	No	No	-
Stochastic Analysis Tasks and Functions			
`$q_initialize, $q_add $q_full, $q_exam, $q_remove`	No	No	-
PLA Modeling Tasks			
`$async$and$array, $async$and$plane, $async$nand$array, $async$or$array, $async$nand$plane, $sync$or$array, $async$or$plane, $async$nor$array, $async$nor$plane, $sync$and$array, $sync$and$plane, $sync$nand$array, $sync$nand$plane, $sync$or$plane, $sync$nor$array, $sync$nor$plane`	No	No	-
Miscellaneous Tasks and Functions			
`$system`	No	No	-
Display Tasks			

TABLE 7 Supported Verilog System-Tasks

Task	Supported in ZEMI3/ZEMI4	Supported in Design	Default
`$display, $displayb, $displayh, $displayo, $write, $writeb, $writeh, $writeo`	Yes	Yes	No
`$strobe, $monitorh`	No	No	-
`$monitor (initial block)`	No	Yes	Yes
File I/O Tasks and Functions			
`$fopen, $fclose, fdisplay, $fdisplayb, $fdisplayh, $fdisplayo, $fwrite, $fwriteb, $fwriteh, $fwriteo`	Yes	Yes	No
`$sformat`	No	No	-
Memory Load Tasks			
`$readmemb, $readmemh`	Yes	Yes	Yes
Memory Dump Tasks			
`$writememb, $writememh`	No	No	-
`$memset`	Yes	Yes	Yes
Command Line Input			
`$value$plusargs (always block), $test$plusargs(always block)`	No	Yes	No
VCD Tasks			
`$dumpvars, $dumpports`	Limited Support	Limited Support	Yes

TABLE 7 Supported Verilog System-Tasks

Task	Supported in ZEMI3/ZEMI4	Supported in Design	Default
`$dumpfile, $dumpoff, $dumpon, $dumpall, $dumplimit, $dumpflash, $dumpportsoff, $dumpportson, $dumpportsall, $dumpportslimit, $dumpportsflush`	No	No	-
Note: `dumpvars/dumports` are used for Qiwc and FWC waveform capture. For details on usage, see the *ZeBu Server Debug Guide*.			

The following example explains how to use these system tasks:

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


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

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

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

Some of the Verilog System tasks can be enabled by other UTF 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 at Runtime*.

3 Compilation

This chapter describes how to compile your design for ZeBu. It discusses the following sections:

- *Overall Unified Compile Flow*
- *Compilation Script for VCS Unified Use Model*
- *Specifying the Settings in ZeBu Unified Compile Script*
- *Using a Compute Farm*
- *Compiling a Project*
- *Compilation Analysis*

3.1 Overall Unified Compile Flow

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

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

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

The entire compilation flow is controlled by **zCui**, a the orchestration tool, for ZeBu compilation. **zCui** manages the VCS command execution and the ZeBu back-end compilation.

3.1.1 Basic UTF Commands

The following table lists some of the basic mandatory UTF commands.

TABLE 8 Basic UTF Mandatory Commands

Commands	Description
`architecture_file -filename {<path_to_zse_configuration.tcl>}`	Specifies the hardware architecture file.
`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.

Example of a UTF File

```
architecture_file -filename /path/to/zse_configuration.tcl
vcs_exec_command /path/to/vcs_script.sh
set_hwtop -module hwtop
```

3.2 Compilation Script for VCS Unified Use Model

For compilation, the design is provided through the VCS command script to:

- Specify RTL source files
- Source the language to be used
- Specify the compilation library
- Define the top module for the design
- Top Level VHDL generics, Verilog parameters
- Add include paths
- Add Top level Verilog macros
- Add Library paths
- Add Verilog library file name extension

In emulation, you can reuse the VCS script used in the simulation environment to reduce the design bring-up time.

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

- *Setting Up VCS*
- *Setting up the synopsys_sim.setup File*
- *Setting up Compilation Commands Using UTF*
- *VCS Options*
- *Compiling With DesignWare Building Blocks*

3.2.1 Setting Up VCS

To setup VCS, perform the following steps:

1. Point the `$VCS_HOME` environment variable to the VCS installation path. Also, the `$PATH` environment variable must 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 platform:

```
% vcs –full64 -id
```

3.2.2 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 precedence to the lowest):

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`)

Example of `synopsys_sym.setup` file

```
WORK > MYLIB
MYLIB : /path/to/mylib
```

3.2.3 Setting up Compilation Commands Using UTF

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

The script must 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 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).

    NOTE: *The design analysis units can be done outside of the ZeBu compilation and be reused. The mandatory command is the VCS elaboration command that must be managed by* **zCui** *exclusively.*

3.2.4 VCS Options

There are five categories of VCS options:

- Compilation
- Elaboration

- Object-generation
- Linking
- Simulation

In the emulation (ZeBu) flow, only compilation and elaboration options are used.

VCS supports two compile options as follows:

- Two-Step flow: In this flow, only Verilog or SystemVerilog is used and the compilation options provided to the `vcs` command.
- Three-Step flow: In this flow, VHDL, Verilog, SystemVerilog, and SystemC are used and the compilation options provided to the `vlogan` or `vhdlan` command.

The following table lists example options for compilation and elaboration.

TABLE 9 VCS Options

Examples for VCS Options	Description
Examples for VCS Compilation Options	
Verilog Analyzer (vlogan)	• `+define+<macro>`: Defines a macro in the Verilog source • `-f file`: Specifies files and command options • `-l <logfile>`: Generates the log file • `-q - Quiet` (no internal messages and banner) • `-v <lib_file>`: Specifies a Verilog library file • `-y <libdir>`: Specifies a directory of Verilog library files • `+libext+<ext>`: Specifies library file extensions (used with `-y`) • `-work <libdir>`: Analyzes into a specified logical library • `+v2k`: Enables Verilog 2001 constructs • `-sverilog`: Enables SystemVerilog constructs • `-timescale=1ns/1ps`: Specifies a default timescale • `+incdir+<dir>`: Specifies search directory for included files • `+librescan`: Searches unresolved module starting first library in the `vlogan` command

TABLE 9 VCS Options

Examples for VCS Options	Description
VHDL Analyzer (vhdlan)	• `-work <logical_lib>`: Analyzes the specified logical library • `-q - Quiet` (no internal messages and banner) • `-nc`: Suppresses the copyright header • `-vhdl87`: Enables VHDL-87 syntax (VHDL-93 is default) • `-f <options file>`: Specifies source files and switches • `-xlrm`: Allows a relaxed/non-LRM compliant code • `-smart_order`: Identifies the file order dependencies
Examples for VCS Elaboration Options	
Provided to vcs command	• `-l <logfile>`: Creates the runtime log file • `-P pli.tab`: Compiles the user-defined PLI table • `<.c|.o files>`: Adds C or object files to compile • `-xlrm`: Allows a relaxed/non-LRM compliant code • `-ignore driver_checks`: Suppresses multiple driver checks • `-liblist`: Specifies the library search order for unresolved • `module` or `entity` instantiated in Verilog

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

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

Or

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

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

3.2.5 Compiling With DesignWare Building Blocks

For DesignWare Building Blocks (DWBB) compile-automation in ZeBu, ensure that `synopsys_sim.setup` is available (as it is a must in mixed-signal flow) and then perform the following steps:

1. Point the `$SYNOPSYS` environment variable to the Synopsys synthesis (DesignCompiler) installation directory. For example,

```
% setenv SYNOPSYS /global/apps3/syn_2015.06-SP4
```

2. Add the `-syn_dw` option to VCS analyze commands (`vlogan`/`vhdlan`).

   For example,

```
% vlogan -syn_dw <other vlogan options>
% vhdlan -syn_dw <other vhdlan options>
```

3. Add the `-syn_dw` option to the VCS elaboration command.

   For example,

```
% vcs -syn_dw <other elab options>
```

4. Add `+dump_dir=<directory_full_path>`

   This analyze and elaboration option allows you to locate auto-parsed content in a custom location. By default, automation generates DesignWare libraries at `vlogan`/`vhdlan` launch area.

   For example:

```
% vlogan -syn_dw+dump_dir=/path/dw_libs <other vlogan opts>
% vcs -syn_dw+dump_dir=/path/dw_libs <other elab opts>
```

The following is an another option available in DWBB compile-automation:

- `+allow_override`: This elaboration option allows you to override higher priority of auto-compiled cells and follow regular design resolution rules. By default, automation selects its auto-compiled cells while instances resolution during VCS elaboration.

- `+minpower`: This analysis and elaboration option allows the usage of cells from DesignWare `minPower` library components instead of regular DWBB. The DesignWare `minPower` is an additional set of cells with the basic DWBB targeting for reducing power consumption.

For example:

```
% vlogan -syn_dw+minpower
% vcs -syn_dw+minpower
```

3.3 Specifying the Settings in ZeBu Unified Compile Script

For compilation, to specify settings in ZeBu Unified Compile Script, the recommended settings are follows:

- Declare the H/W configuration file

```
architecture_file -filename {<path_to_zse_configuration.tcl>}
```

- Specify VCS command script for the design

```
vcs_exec_command {<script_name_with_path>}
```

- Specify the top level of the design

```
set_hwtop -module <design_top_module_name>
```

- Reconstruct Combinational Signals in the waveforms using CSA

```
debug -waveform_reconstruction true  -> csa and simzilla
```

- Perform Verdi compilation simultaneously

```
debug -verdi_db true
```

- Compile for Post Run Debug

```
debug -use_offline_debug true
```

- Set the design size

```
design_size -mode auto
```

- Enable Automatic Generation of zCores (for designs >1 board)

```
clustering -system_auto_core_generation true
```

- Set FPGA filling rates

```
clustering -resource_usage MEDIUM
```

- Set parameters for Xilinx FPGA P&R

```
fpga -enable_parff MULTI_STAGE
```

- Enable generation of a compilation profiler

```
profile -compile true
```

3.4 Using a Compute Farm

It is recommended to compile on a compute farm. If the remote command is not declared, the ZeBu compiler is launched on the system from which **zCui** is launched.

For more details on the compute farm, see the *Compilation PCs* section in the *ZeBu Server 4 Site Planning Guide*.

3.4.1 Configuring the Job Queuing System

Each ZeBu compilation task has a name and can be associated to a queue.

There are pre-defined queues but you can define additional queues and associate tasks to them. There is also a default association of tasks to the predefined queues.

To setup the job queues, use the following command:

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

where,

- `njobs` (number of jobs: It must consider the number of processors in the farm (which is the maximum number of jobs for efficiency purposes) and the acceptable load on the computer that runs the load sharing tool.

- `<given_queue_name>`: It is a predefined value or a user-defined value.
  The predefined values are:
  - DEFAULT_QUEUE
  - Zebu
  - ZebuAlternateSynthesis
  - ZebuHeavy
  - ZebuIse
  - ZebuLight
  - ZebuRelaunchedIse
  - ZebuRelaunchedIseLevel2
  - ZebuSuperHeavy
  - ZebuSynthesis

For example, use the following command as a minimal setting to run all jobs on the compute farm:

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

With this command, all predefined queues use the same `submit` command with an unlimited numbers of jobs.

3.4.2 Using Sun™ Grid Engine or Platform LSF®

It is recommended to use an external task scheduler such as Sun™ Grid Engine or Platform LSF® to perform the load balancing of the compute farm.

The following table lists recommendations for Sun™ Grid Engine or Platform LSF®:

TABLE 10 Sun Grid Engine and Platform LSF Recommendation

External Task Scheduler	Typical Remote Command	Typical Kill Command
Sun Grid Engine	`qrsh`	`qdel`
Platform LSF with blocking jobs and a non-interactive queue	`bsub -K`	`bkill`
Platform LSF with blocking jobs and an interactive queue	`bsub -Is`	`bkill`

Additional options may be necessary for these commands to match the compilation constraints and your IT environment. In particular, you need to target only compilation hosts with 64-bit operating systems and to manage priorities.

The typical submit command with `qrsh` is as follows:

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

3.5 Compiling a Project

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

- Batch mode
- GUI mode

3.5.1 Compiling With zCui in the Batch Mode

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

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

where,

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

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

3.5.2 Compiling With zCui in the GUI Mode

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**.

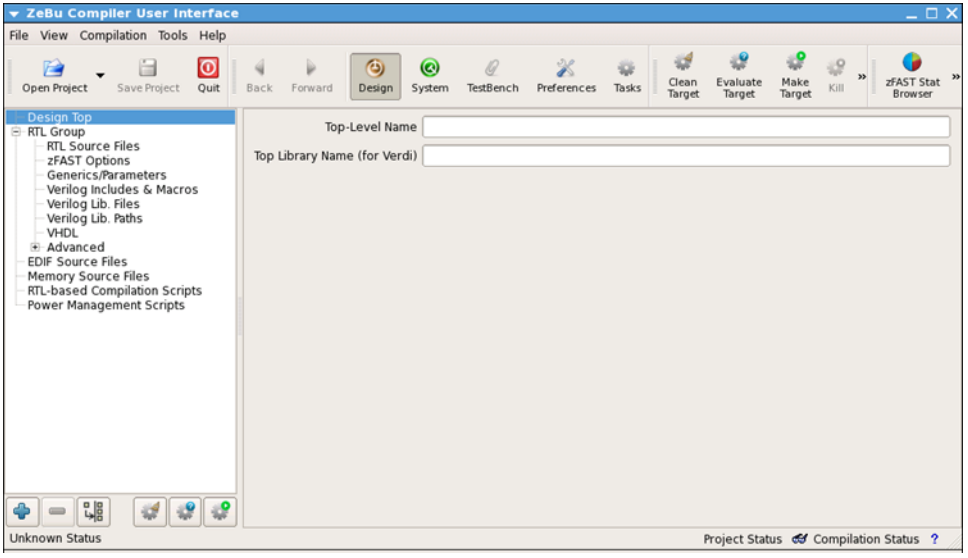

FIGURE 9. `zCui` Default View

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

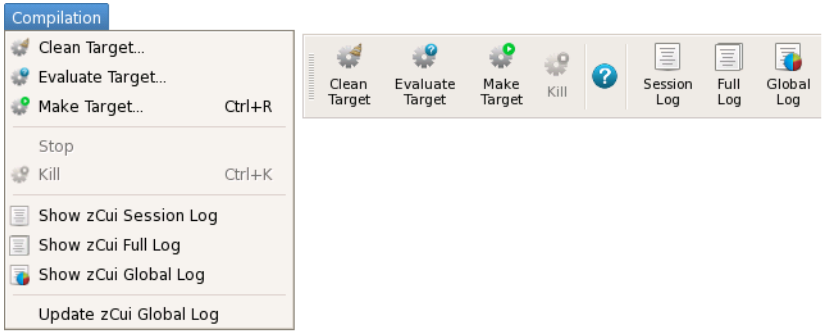

FIGURE 10. `zCui` Options

The following operations are available in the GUI mode:

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

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

```
zCui -u <project_name>.utf [-w <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:

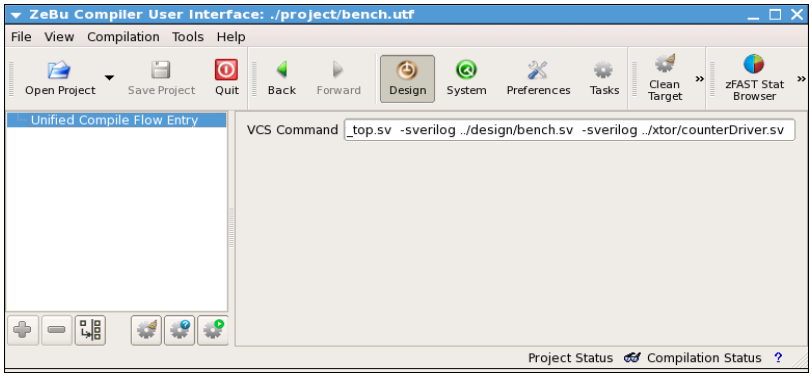

FIGURE 11. `zCui` GUI

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

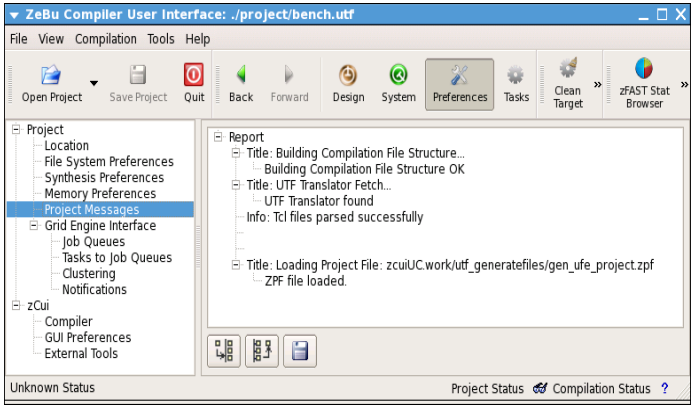

FIGURE 12. Project Messages Panel

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

the next compilation.

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

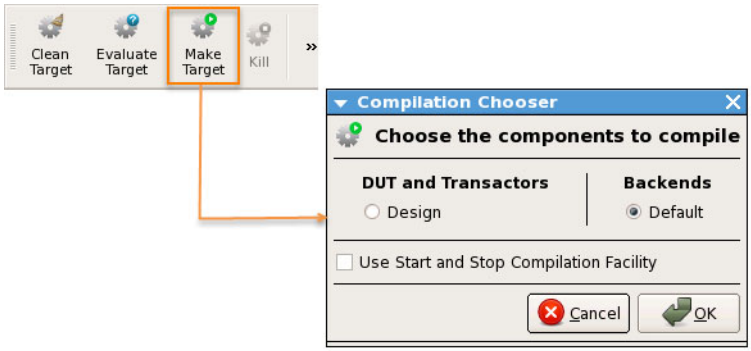

FIGURE 13. Launching Compilation in zCui

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

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

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

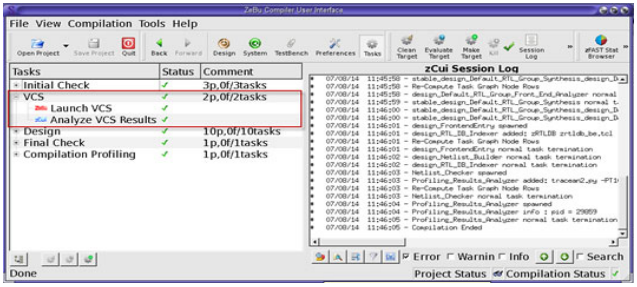

FIGURE 14. Viewing the VCS Task in zCui

3.6 Compilation Analysis

This section consists of the following sub-sections:

- *Important Log Files*
- *Analyzing Compilation Results Using zBatchExplorer*

3.6.1 Important Log Files

There are two important log files as follows:

- The full compilation flow is logged by **zCui** in `<zcui.work>/zCui/log/zCui.log`.
- The design size estimation can be found in `<zcui.work>/zebu.work/zTopBuild.log`.

```
#   step DESIGN SIZE ESTIMATION : Computed a design size of 3
module(s) for given user fill rates
#   step DESIGN SIZE ESTIMATION :
# +-------------------------------+------+------+--------+-------
+---------+-----+-------+-----+
# |Resource usage                 |LUTs  |Regs  |Fwc bits|Fwc
ips|Qiwc bits|BRAMs|RAMLUTs|DSPs |
# +-------------------------------+------+------+--------+-------+
# |Estimated size of the design   |13M   |7,167K|1,936   |0      |0
|1,888|64K     |4,862|
# |Board size with user fill-rates|5,497K|4,838K|2,604K  |2,604K
|4,719K   |5,805|1,241K |9,720|
# +-------------------------------+------+------+--------+-------+
# |               For 1 boards     |241.7%|148.2%|0.1%    |0.0%
|0.0%     |32.5%|5.2%   |50.0%|
# |               For 2 boards     |120.9%|74.1% |0.0%    |0.0%
|0.0%     |16.3%|2.6%   |25.0%|
```

```
# |Computed  ->  For 3 boards     |80.6% |49.4% |0.0%    |0.0%
|0.0%     |10.8%|1.7%   |16.7%|
# |              For 4 boards     |60.4% |37.0% |0.0%    |0.0%
|0.0%     |8.1% |1.3%   |12.5%|
# |              For 5 boards     |48.3% |29.6% |0.0%    |0.0%
|0.0%     |6.5% |1.0%   |10.0%|
# |              For 6 boards     |40.3% |24.7% |0.0%    |0.0%
|0.0%     |5.4% |0.9%   |8.3% |
# |              For 7 boards     |34.5% |21.2% |0.0%    |0.0%
|0.0%     |4.6% |0.7%   |7.1% |
# +-------------------------------+------+------+--------+-------+
```

The theoretical performance of the design can be found in `<zcui.work>/zebu.work/zTime.log` (more conservative value in `zTime_fpga.log` from the post FPGA Timing Analysis).

```
#   step REPORT : +-----------------------------------------------
--------------------------------+
#   step REPORT :   Longest inter-fpga filter path delay is : 89 ns
#   step REPORT :   Critical routing data path delay : 290 ns
#   step REPORT :   . Constant part    : 50 ns
#   step REPORT :   . Multiplexed part : 241 ns
#   step REPORT :   Xclock frequency is : 450 MHz
#   step REPORT :   Longest memory period is : 820 ns
#   step REPORT :   Fast Waveform Captures detected, if use at run-
time the driverClk frequency might be limited.
#   step REPORT :   Driver clock frequency is limited by memories
#   step REPORT :   The theoretical frequency using default
settings and ignoring clock skew is 1219 Khz
#   step REPORT : +-----------------------------------------------+
```

3.6.2 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:

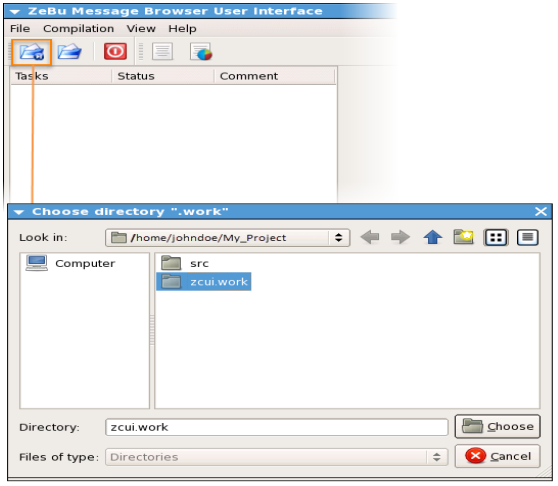

FIGURE 15. Opening a Directory using `zBatchExplorer`

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:

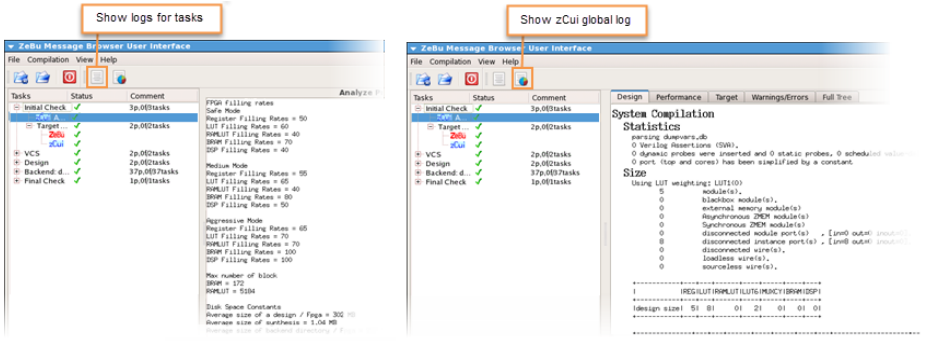

FIGURE 16. Selecting Log File and `zCui` Global File

3.7 Analyzing Compilation Health Using zHealth

In most of the cases, the compilation issues observed in FPGA P&Rs have a root cause upstream in the compilation flow.

The zHealth tool helps you to list the potential compilation health issues in the compilation flow.

zHealth script generates an HTML report about the health of the current the compilation that is in-progress or finished.

The compilation health is modeled by scoring the computation at different stages of the compilation flow versus the defined thresholds.

Each compilation stages sections are decorated using the health labels (GREEN, YELLOW, RED). Only general and non-null scores sections are reported. The HTML report has customizable thresholds. A single page HTML document is reported with collapsible and nested sections.

- Interface

```
zHealth [<backend folder path>] [-T "<report title>"] [-o
<generated report name>.html]
```

By default, zHealth looks for a `zcui.work/zebu.work` that is the backend compilation directory and produces a `zHealth.html` file.

The following table lists the section available in the HTML report.

TABLE 11 zHealth HTML Report Sections

Header • Customizable Title • Timestamp • Absolute path of the zebu.work	System Level • Required and used boards • High cut after partitioning * • Board resources overflow * • Cut grow after post partitioning steps (RLC, etc.) *
Configuration Management • Used tools paths and versions	Board Level • High FPGA cuts after partitioning * • FPGA resources overflow * • High FPGA cut increase after partitioning (RLC, etc.) *

TABLE 11 zHealth HTML Report Sections

Front-end • Risky modules * • Memories with high number of ports * • zForce commands with high number of affected wires *	System Routing • High post routing FPGA cut * • High FPGA cut difference after routing * • High FPGA max XDR *
* Displayed only on non-null score	FPGA Level • FPGA going into PARFF * • Top 10 concerning FPGAs * • FPGA statistics table

The following figure displays an example HTML page generated by zHealth.

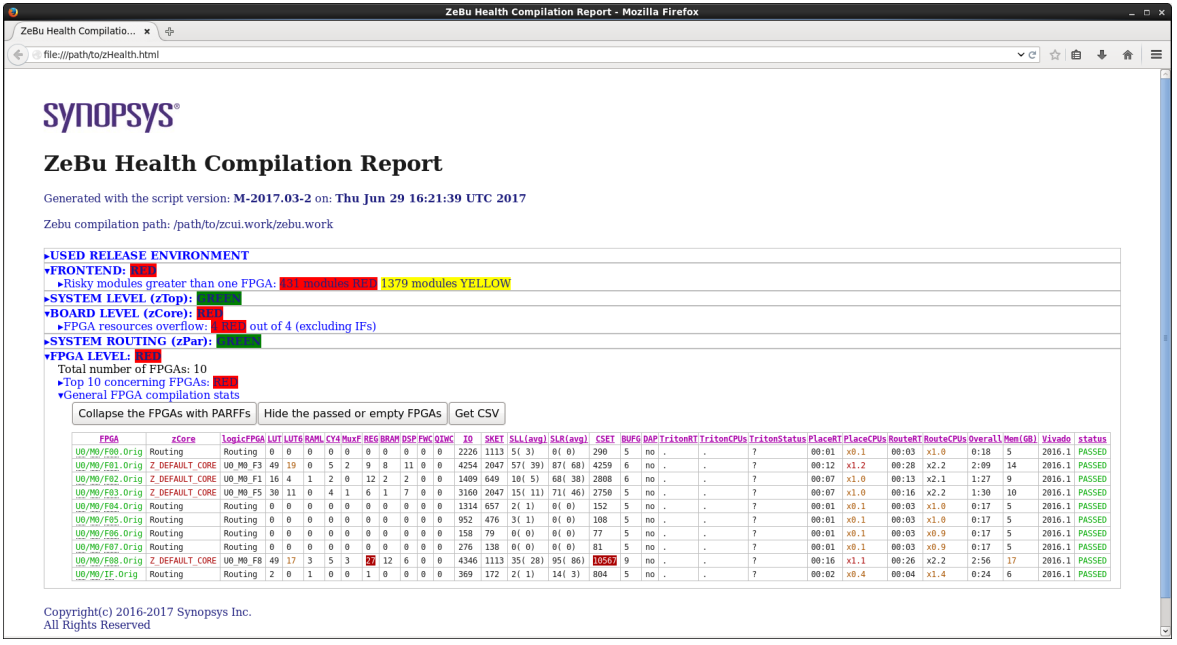

FIGURE 17. Example of HTML Report Generated by zHealth

4 Controlling Emulation 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
- **`zRun`**: A Tcl-based program that controls the ZeBu system emulation runtime. **`zRun`** can be used as standalone or with any of the preceding options. **`zRun`** has a graphical interface; but can also be used in the batch mode.

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

This chapter discusses the following topics:

- *Controlling Runtime Parameters*
- *Using zRun to Control Emulation Runtime*
- *Runtime Performance Analysis With zTune*

4.1 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*
- *Setting Design Clock Parameters for Runtime*
- *Memory Initialization for Runtime*

4.1.1 Emulation Speed

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

The time estimation input can be found in `zTime_fpga.log` if the post-FPGA timing analysis is activated.

4.1.2 Setting Design Clock Parameters for Runtime

To specify parameters for design clocks declared as outputs of `zceiClockPort` during compilation, 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 as a relative value using the `VirtualFrequency` parameter or a clock waveform using the `Waveform` parameter.

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

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

The following example displays how to declare a frequency ratio of two 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"
```

4.1.3 Memory Initialization for Runtime

To initialize a memory for runtime, 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.

4.2 Using zRun to Control Emulation Runtime

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

You must set the `$DISPLAY` environment variable correctly before starting **zRun**, especially if you use a remote terminal.

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

```
zRun -functionList
```

You can use the zRun tool with a Tcl script. The `-do <tclsript>` option specifies a Tcl script that is to be executed when **zRun** is launched. The Tcl script can contain any Tcl command and specific **zRun** commands. You can initialize memories and automate the entire run using Tcl commands. This is also useful when an initialization phase is executed before the **zRun** GUI launches. For example:

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

For information on using the **zRun** tool, see the following subsections:

- *Common zRun Command-Line Options*
- *Controlling Clocks During Runtime Using zRun*
- *Managing Memories*

4.2.1 Common zRun Command-Line Options

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

- *Using zRun With a C/C++ Testbench*

- *Starting zRun in Batch Mode*
- *Specific Recommendation for zRun Batch Mode*

Using `zRun` With a C/C++ Testbench

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

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

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

Starting `zRun` in Batch Mode

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

For example,

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

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

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

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

Specific Recommendation for `zRun` Batch Mode

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

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

```
zRun -synchroClose
```

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

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

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

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

Instead, use the following code snippet:

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

For example,

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

4.2.2 Controlling Clocks During Runtime Using zRun

This section describes the commands to control the ZeBu clocks. For more information, see the following subsections:

- *Obtaining a List of Clock Groups*
- *Enabling Clocks for N Cycles*
- *Enabling Clocks in the Free-Running Mode*
- *Disabling Clocks*

- *Getting the Clock Status*
- *Getting the Number Clock Cycles Executed*

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

4.2.2.1 Obtaining a List of Clock Groups

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

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

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

4.2.2.2 Enabling Clocks for N Cycles

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

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

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

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

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

Syntax:

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

where,

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

For example:

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

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

4.2.2.3 Enabling Clocks in the Free-Running Mode

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

Syntax:

```
ZEBU_Clock_enableForever <clk_name>
```

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

Note

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

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

For example,

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

It works only if `-debugDriverClk` is used in the **zRun** command line.

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

4.2.2.4 Disabling Clocks

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

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

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

Syntax:

```
ZEBU_Clock_disable <clk_name>
```

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

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

4.2.2.5 Getting the Clock Status

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

Syntax:

```
ZEBU_Clock_getStatus <clk_name>
```

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

It returns running or stopped.

4.2.2.6 Getting the Number Clock Cycles Executed

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

Syntax:

```
ZEBU_Clock_getCounter <clk_name>
```

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

Example

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

```
set clkGroupList [ZEBU_Clock_getGroupNameList]
# get the first clock group
set firstClkGrp [lindex $clkGroupList 0]
# get the list of clocks in the group
set clkList [ZEBU_Clock_getNameList $firstClkGrp]
# get the first clock of the group
set firstClk [lindex $clkList 0]
# Start 1000000 cycles on firstClk
ZEBU_Clock_enable $firstClk 1000000

# Wait until end of run

# Note : use ZEBU_getStatus because a testbench (if used)
# could close the session during the run
while {    [ZEBU_getStatus] == "open" \
        && [ZEBU_Clock_getStatus $firstClk] == "running" } {
  # Display number of cycles executed
  set nbCycles [ZEBU_Clock_getCounter $firstClk]
  puts "$firstClk still running : $nbCycles has been executed"
  after 500
}
```

4.2.3 Managing Memories

The external **zrm** memories, namely, BRAMs and LUTRAMs can be controlled from **zRun** Tcl scripts. You can save the content of a memory in a file using `ZEBU_Memory_storeToFile` or in a buffer using `ZEBU_Memory_storeToBuffer`. In addition, you can load content into a memory either from a memory file using `ZEBU_Memory_loadFromFile` or from a buffer using `ZEBU_Memory_loadFromBuffer`.

This section consists of the following subsections:

- *Saving Memory Contents*
- *Loading Content From a File*
- *Loading Content From a Buffer*

Saving Memory Contents

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

- `ZEBU_Memory_storeToFile`: This command stores the memory content into a file. This command is dynamic, that is, it is available while emulation is running.
- `ZEBU_Memory_storeToBuffer`: This command stores the memory content and flushes it in the buffer passed as parameter.

Syntax:

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

ZEBU_Memory_storeToBuffer <mem_name> [<first_addr> <last_addr>]
[format]
```

Where:

- `<mem_name>` is the memory name returned by `ZEBU_Memory_getNameList`.
- `<file_name>` is the name of the memory file.
- `[<first_addr> <last_addr>]` are the decimal values of the first and last memory addresses to be written in the memory file or buffer.
- You must specify both `<first_addr>` and `<last_addr>` or none of them. If you specify none, the entire memory content is stored to the file.
- `[mode]` sets the format of the output file (optional parameter):

- `t` = text (default mode).
- `b` = binary.

- `[format]` is the format for operation (optional parameter):
  - `%b` for binary.
  - `%o` for octal.
  - `%h` for hexadecimal.

For example,

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

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

Loading Content From a File

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

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

Syntax:

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

where,

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

For example,

The following command lines initialize every memory with different files:

```
# get the memories list with '.' as hierarchy separator
set memoryList [ZEBU_Memory_getNameList]
# initialize it
set i 0
foreach name $memoryList {

  puts "Initialize memory $name with file mem_$i"
  ZEBU_Memory_loadFromFile $name mem_$i
  incr i
}
```

Loading Content From a Buffer

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

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

Syntax:

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

where,

- `<mem_name>`: Specifies the name of the memory to fill as returned by `ZEBU_Memory_getNameList`.
- `<file_name>`: Specifies the name of the memory file.
- `<buf_name>`: Specifies the name of the buffer.
- `[<first_addr> <last_addr>]`: Specifies the decimal values of the first and last memory addresses to be written in the memory file or buffer.
- You must either specify both `<first_addr>` and `<last_addr>` or none of them. If you specify none, the entire memory content is loaded.
- `[format]` is the format for operation (optional parameter):
  - `%b` for binary.
  - `%o` for octal.
  - `%h` for hexadecimal.

4.3 Runtime Performance Analysis With zTune

zTune is a tool to analyze emulation performance. It presents in the same view both the real time software profile and the hardware profile. It helps you to identify emulation performance bottlenecks. The following are the important features of **zTune**:

- Monitors clock frequencies.
- Monitors clock stopping activity.
- Monitors software activity.
- Interactive GUI analyzes **zTune** profiling data.

The analysis does not affect the overall system performance. To generate **zTune** profiling data, you must compile the design by adding the profile `-xtors true` option in the UTF file and launch emulation runtime with **zTune** runtime options (see ***ZeBu Server User Guide***).

When performance monitoring is enabled, a directory is generated during the emulation. This directory is local to the first process that enables the performance monitoring. After emulation, this directory is post-processed by **zTune** to create a GUI that allows you to analyze the emulation performance.

For more details about **zTune**, see ***ZeBu Server User Guide.***

5 Using SystemVerilog Assertions at Runtime

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 time and runtime.

To use SVA in ZeBu, perform the following steps:

1. Enable SVA during design compile using UTF commands (see *Enabling SVA Through assertion_synthesis UTF Command*)
2. Start SVA using the Start method (see *Starting Assertion Processing*)
3. Generate reports for SVA post-emulation (see *Post-Processing SVA*)

5.1 Enabling SVA Through assertion_synthesis UTF Command

The compilation options for SystemVerilog Assertions (SVA) are available in the UTF command, `assertion_synthesis`. This command is mandatory to enable SVA.

Commonly used options to control 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 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 (legacy `zFast:"sva:fullModeAll"`).

5.2 Controlling Synthesized Assertions at Runtime

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

This section consists of the following subsections:

- *Controlling Assertions From the C++ Testbench*
- *Controlling Assertions Using zRun*
- *Post-Processing SVA*

5.2.1 Controlling Assertions From the C++ Testbench

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

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. Assertions can be enabled in the following two modes:

- *Post-Processing Mode*
- *Live Processing Mode*
- *Stopping SVA Processing*

Post-Processing Mode

In this mode, all assertion failure messages during design run are dumped into a report file for post-emulation analysis. A post-processing tool, **zsvaReport** (see *Post-Processing SVA*), reads this report file and generates an 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.
- `SVA::ENABLE_REPORT` enables the report only (default).

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:

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

Stopping SVA Processing

To stop SVA processing, call the `SVA::Stop` method. The `SVA::Start` method must be called before the `SVA::Stop` method. 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);
```

5.2.2 Controlling Assertions Using zRun

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

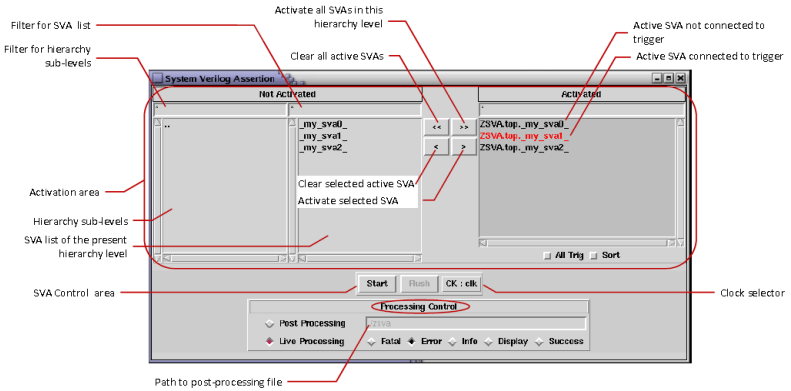

FIGURE 18. System Verilog Assertion Panel

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

5.2.3 Post-Processing SVA

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

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

For example,

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

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

6 DPI Synthesis Support

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

This section describes the following topics:

- *Compilation UTF Commands for DPI Synthesis*
- *Controlling DPI Function Calls at Runtime*
- *Example: Importing Function Calls in C and SystemVerilog*

6.1 Compilation UTF Commands for DPI Synthesis

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

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

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

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

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

```
dpi_synthesis -enable ALL
```

For more details about DPI compilation and synthesis, see the *ZeBu Server User Guide*.

6.2 Controlling DPI Function Calls at Runtime

The implementation of the DPI functions is provided as a dynamic runtime library. The DPI function calls may be controlled from the ZeBu runtime environment using **zRun** (Tcl commands) or using a C/C++ API.

This section describes the following subsections:

- *Formulating C Functions for DPI*
- *Enabling DPI Using zRun*
- *Enabling DPI Using the C++ API*

Formulating C Functions for DPI

The C code for the DPI function must include the `grp0_ccall.h` header using the following command:

```
#include <grp0_ccall.h>
```

The `grp0_ccall.h` provides the necessary definitions for ZeBu functions and is stored in the `zebu.work` compilation directory.

The C/C++ code must be compiled as a dynamic library as follows:

```
g++ -m64 -O2 -I$ZEBU_ROOT/include -I$ZEBU_WORK -Wall  -rdynamic -
fPIC -c source.c -o source.o

g++ -shared -o libdpi.so -m64 -lpthread -L$ZEBU_ROOT/lib source.o
```

Enabling DPI Using `zRun`

Before enabling the DPI, load the dynamic library containing the imported C functions.

This can be done using the following command:

```
ZEBU_CCall_loadDynamicLibrary <library Name>
```

Once loaded, it is possible to enable the DPI calls using the following command:

```
ZEBU_CCall_start
```

If you want to enable a specific DPI call, pass its hierarchical path as an argument to the `ZEBU_CCall_start` command.

The DPI calls can be disabled using the `ZEBU_CCall_stop` command.

Enabling DPI Using the C++ API

The following is the C++ API can be used to control DPI calls at run.

```
CCall::LoadDynamicLibrary(board, <library name>);
CCall:Start(board);
```

6.3 Example: Importing Function Calls in C and SystemVerilog

The following code snippets explain how to use imported function calls in C and SystemVerilog.

Example: SystemVerilog design

```
module fifo_usage_spy #(
  parameter WIDTH=5,
  parameter DEPTH=32
)(
  input  clk,
```

```
  input  [WIDTH-1:0] remain
);
  reg [WIDTH-1:0] min;
  import "DPI-C" context function void fifo_usage_spy_notify (
    input bit[WIDTH-1:0] min
  );
  always @(posedge clk or negedge rstn)
    if (!rstn)
      min <= DEPTH;
    else
      if (remain < min) begin
        min <= remain;
     fifo_usage_spy_notify(min);
      end

endmodule
```

Example of a C code design

```
#include <grp0_ccall.h>
…

extern void fifo_usage_spy_notify (const svBitVecVal* _arg_min)
{
  // Retrieve the scope of the function
  svScope scope = svGetScope ();
  // Retrieve user data
```

```
  void *ctx = svGetUserData(scope,
(void*)(fifo_usage_spy_notify));
  if (ctx == NULL)
  {
    // first call
    const char *i_name = svGetNameFromScope (scope);
    ctx = new MyObject(i_name);
    svPutUserData(scope, (void*)fifo_usage_spy_notify, ctx);
  }
  // Print fifo space
  cout << "Free space in FIFO:" << _arg_min[0] << endl;
}
```