

# **ZeBu® Getting Started Guide**

---

Version V-2024.03-1, July 2024



# Copyright and Proprietary Information Notice

© 2024 Synopsys, Inc. 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 <https://www.synopsys.com/company/legal/trademarks-brands.html>.

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

## Free and Open-Source Licensing Notices

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

## Third-Party Links

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

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

# Contents

---

About This Book .....	6
Contents of This Book .....	6
Hardware Documentation .....	7
Related Documentation .....	7
Other Useful References .....	8
Typographical Conventions .....	9
Synopsys Statement on Inclusivity and Diversity .....	9

---

<b>1. ZeBu Server .....</b>	<b>11</b>
Features of ZeBu Server .....	11
ZeBu Server 4 Hardware Overview .....	12
Design Capacity of ZeBu Server 4 .....	13
FPGAs in ZeBu Server 4 .....	13
ZeBu Emulation Flow .....	14
Introduction to UTF .....	15

---

<b>2. Assembling a Design for Emulation .....</b>	<b>17</b>
Mapping Your Design and Test Environment .....	17
Importing a Design From a Simulation Environment .....	17
Preparing a DUT-only Environment .....	18
Clock Modeling .....	19
zceiClockPort Primitive .....	19
Clock Delay Support (clock_delay) .....	19
clockDelayPort Primitive .....	20
Memory Modeling .....	20
Memory Inference From a Design .....	21
Guidelines for Memory Performance Improvement .....	21
Memory Model IP .....	23
Accessing Signals During Runtime .....	23
Reading Signals .....	23

Forcing and Injecting Values .....	24
Verilog System Tasks .....	24
Examples of Using System Tasks .....	28
<hr/>	
<b>3. Compilation .....</b>	<b>29</b>
Overall Unified Compile Flow .....	29
Basic UTF Commands .....	29
Example of a UTF File .....	30
Compilation Script for VCS Unified Use Model .....	30
Setting Up VCS .....	31
Setting up the <code>synopsys_sim.setup</code> File .....	31
Example of <code>synopsys_sym.setup</code> file .....	32
Setting up Compilation Commands Using UTF .....	32
VCS Options .....	32
Compilation Settings for DesignWare Blocks .....	34
Specifying the Settings in ZeBu Unified Compile Script .....	35
Using a Compute Farm .....	36
Configuring the Job Queuing System .....	36
Using Sun™ Grid Engine or Platform LSF® .....	37
Compiling a Project .....	37
Compiling With <code>zCui</code> in the Batch Mode .....	38
Compiling With <code>zCui</code> in the GUI Mode .....	38
Compilation Analysis .....	41
Important Log Files .....	42
Analyzing Compilation Results Using <code>zBatchExplorer</code> .....	43
Analyzing Compilation Health Using <code>zHealth</code> .....	44
<hr/>	
<b>4. Controlling Emulation Runtime .....</b>	<b>46</b>
Controlling Runtime Parameters .....	46
Setting the Emulation Speed .....	47
Setting Design Clock Parameters for Runtime .....	47
Example 1 .....	48
Example 2 .....	48
Memory Initialization for Runtime .....	48
Using <code>zRci</code> to Control Emulation Runtime .....	48

Common zRci Command-Line Options .....	49
Using zRci With a C++ Testbench .....	49
Starting zRci in Batch Mode .....	49
Controlling Clocks During Runtime Using zRci .....	50
Obtaining a List of Clock Groups .....	50
Enabling Clocks for N Cycles .....	50
Enabling Clocks in the Free-Running Mode .....	51
Disabling Clocks .....	51
Getting the Number Clock Cycles Executed .....	51
Managing Memories .....	52
Runtime Performance Analysis With zTune .....	52
<hr/>	
<b>5. Using SystemVerilog Assertions .....</b>	<b>54</b>
Enabling SVA Through assertion_synthesis UTF Command .....	54
Controlling Synthesized Assertions at Runtime .....	55
Controlling Assertions From the C++ Testbench .....	55
Starting Assertion Processing .....	55
Post-Processing Mode .....	55
Live Processing Mode .....	56
Stopping SVA Processing .....	56
Controlling Assertions Using zRci .....	56
Starting Assertion .....	56
Live Processing Mode .....	57
Post-Processing Mode .....	57
Stopping SVA Processing .....	57
Post-Processing SVA .....	57
<hr/>	
<b>6. DPI Synthesis Support .....</b>	<b>59</b>
Compilation UTF Commands for DPI Synthesis .....	59
Controlling DPI Function Calls at Runtime .....	60
Enabling DPI Using zRci .....	60
Enabling DPI Using the C++ API .....	60
Example: Importing Function Calls in C and SystemVerilog .....	60
Example: SystemVerilog Design .....	60
Example of a C Code Design .....	61

# Preface

---

This section has the following topics:

- [About This Book](#)
- [Contents of This Book](#)
- [Hardware Documentation](#)
- [Related Documentation](#)
- [Other Useful References](#)
- [Typographical Conventions](#)
- [Synopsys Statement on Inclusivity and Diversity](#)

---

## About This Book

The **ZeBu® Getting Started Guide** introduces the Synopsys emulator system. 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® Getting Started Guide* has the following chapters:

Chapter	Describes...
<a href="#">ZeBu Server</a>	Introduces ZeBu Server and its features
<a href="#">Assembling a Design for Emulation</a>	Describes how to prepare your design for emulation before compilation

Chapter	Describes...
<a href="#">Compilation</a>	Describes how to compile your design in ZeBu
<a href="#">Controlling Emulation Runtime</a>	Describes how to control ZeBu runtime
<a href="#">Using SystemVerilog Assertions</a>	Describes SystemVerilog assertions for functional verification
<a href="#">DPI Synthesis Support</a>	Describes DPI imported function calls supported in ZeBu

---

## Hardware Documentation

Document Name	Description
ZeBu Server 5 Site Planning Guide	Describes planning for ZeBu Server 5 hardware installation.
ZeBu Server 5 Site Administration Guide	Provides information on administration tasks for ZeBu Server 5 hardware. It includes software installation.
ZeBu Server 5 Getting Started Guide	Provides brief information about Synopsys' ZeBu Server 5.
ZeBu Server 5 Hardware Performance Technologies User Guide	Provides information on the performance technologies available with the ZeBu Server 5 hardware architecture.
ZeBu Server 5 Release Notes	Provides enhancements and limitations for a new ZeBu Server 5 release.

---

## 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.

Document Name	Description
<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 Synthesis Verification User Guide</i>	Provides a description of zFmCheck.
<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.

## Other Useful References

Document Name	Description
VCS User Guide	Provides information on using VCS to simulate a design.
Verdi User Guide and Tutorial P	Provides information on using Verdi.
Verdi Coverage Technology User Guide, Coverage Technology Reference Guide, Verification Planner User Guide, and Tutorial	Provides information on using Verdi to debug coverage data.



## Typographical Conventions

This document uses the following typographical conventions:

To indicate	Convention Used
Program code	<code>OUT &lt;= IN;</code>
Object names	<code>OUT</code>
Variables representing objects names	<code>&lt;sig-name&gt;</code>
Message	Active low signal name ' <code>&lt;sig-name&gt;</code> ' must end with <code>_X</code>
Message location	<code>OUT &lt;= IN;</code>
Reworked example with message removed	<code>OUT_X &lt;= IN;</code>
Important Information	<b>NOTE:</b> This rule...

The following table describes the syntax used in this document:

Syntax	Description
<code>[ ]</code> (Square brackets)	An optional entry
<code>{ }</code> (Curly braces)	An entry that can be specified once or multiple times
<code> </code> (Vertical bar)	A list of choices out of which you can choose one
<code>...</code> (Horizontal ellipsis)	Other options that you can specify

## 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

## ZeBu Server

---

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

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

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

- [Features of ZeBu Server](#)
- [ZeBu Server 4 Hardware Overview](#)
- [ZeBu Emulation Flow](#)
- [Introduction to UTF](#)

---

## Features of ZeBu Server

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 Smart Z-ICE interface provides support for standard software debuggers using JTAG cables.

**Note:**

A two-slot ZeBu Server 3 is not intended to be part of a multiunit environment.

---

## ZeBu Server 4 Hardware Overview

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 1 ZeBu Server 4: Front Panel



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

---

## Design Capacity of ZeBu Server 4

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.

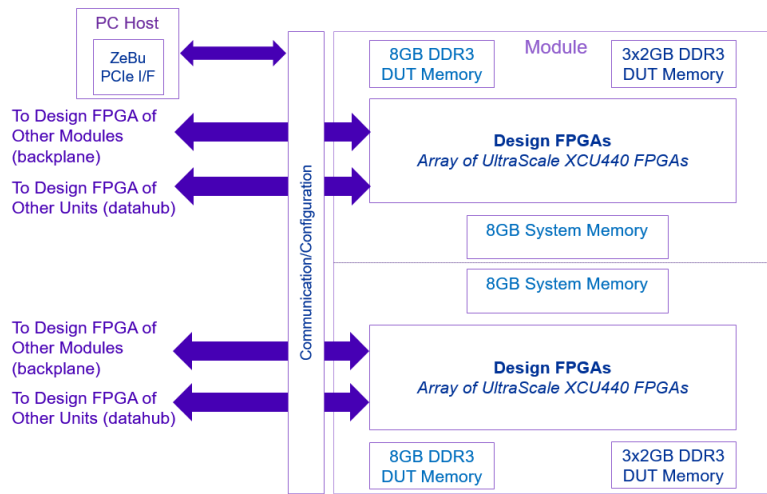
The following table lists FPGA module types supported in ZeBu Server 4.

Table 1 FPGA Type and FPGA Module Types in ZeBu Server 4

FPGAs Type	FPGA Module	Description
XilinxUltrascaleXCVU440	12F	2 FPGAs with 8GB DDR3 memory to map the design6 FPGAs with 2x1GB of DDR3 memory to map the design4 FPGAs with no additional memory

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

Figure 2 Architecture of an FPGA Module in ZeBu Server 4



## ZeBu Emulation Flow

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

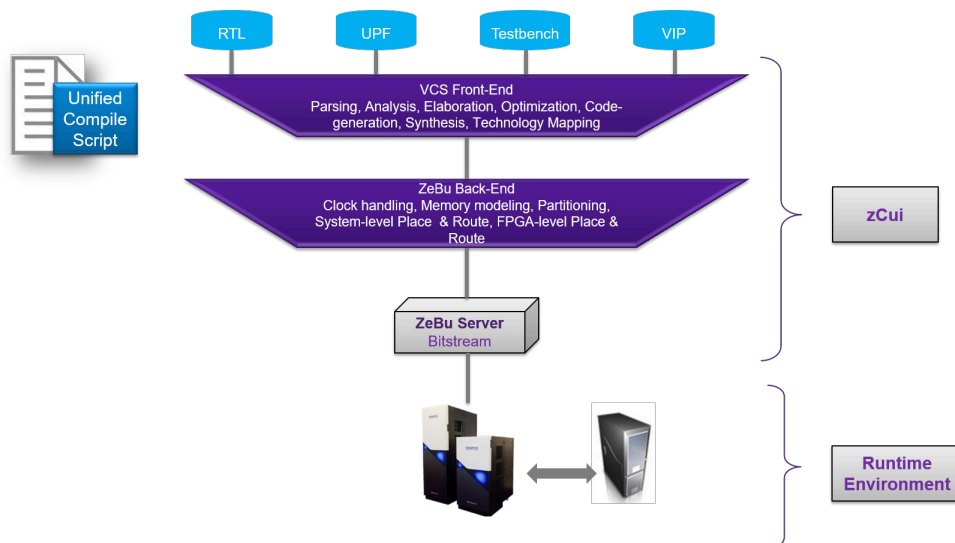
The Unified Compile ZeBu project file, that is a UTF file, contains compilation parameters. ZeBu compilation is managed by the zCui application that can be launched in graphical or batch mode.

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 3 ZeBu Emulation Flow



The ZeBu emulation flow consists of the following steps:

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

## 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 coexist 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 2 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

**Table 2** UTF Commands Categories (Continued)

Settings	Description	Requirement
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
<b>zCui</b> 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 3** UTF Help Commands

Commands	Description
<code>vcs -help utf+all</code>	Displays the list of commands.
<code>vcs -help utf+&lt;command&gt;</code>	Displays the detailed help for <command>.
<code>vcs -help utf+*</code>	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. See the following sections:

- [Mapping Your Design and Test Environment](#)
- [Clock Modeling](#)
- [Memory Modeling](#)
- [Accessing Signals During Runtime](#)
- [Verilog System Tasks](#)

---

### 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](#).

---

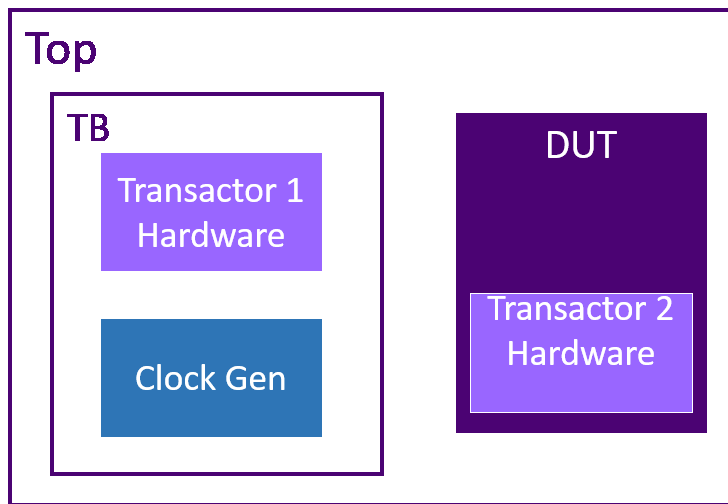
### Importing a Design From a Simulation Environment

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.

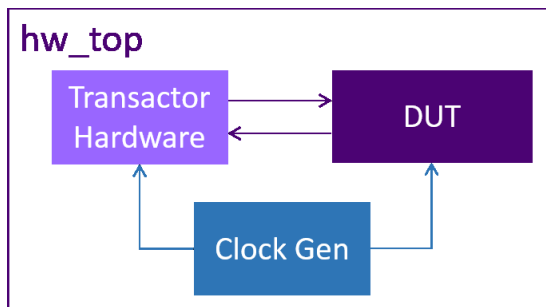
Figure 4 Importing a DUT and Test Environment From a Simulation Environment



## 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 5 Preparing a DUT-Only Environment



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

---

## Clock Modeling

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

- [zceiClockPort Primitive](#)
- [Clock Delay Support \(clock\\_delay\)](#)
- [clockDelayPort Primitive](#)

---

### 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](#).

---

### Clock Delay Support (clock\_delay)

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.

---

## 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).

---

## 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](#)
- [Guidelines for Memory Performance Improvement](#)
- [Memory Model IP](#)

## 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 4**      *Memories and Memory Preferences*

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

## Guidelines for Memory Performance Improvement

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.

---

## 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.

---

## Accessing Signals During Runtime

This section describes how to access signals during runtime. See the following subsections:

- [Reading Signals](#)
- [Forcing and Injecting Values](#)

---

### 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 **zRci** or C++ API described in the `Board.hh` file or the `ZEBU_Board.h` header files.

## 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, combinatorial 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 **zRci** or C++ APIs.

## Verilog System Tasks

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

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

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

**Table 5** Supported Verilog System Tasks

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



**Table 5**      *Supported Verilog System Tasks (Continued)*

Task	Supported in ZEMI-3	Supported in Design	Default
\$time	Yes	Yes	No (clock_delay)
<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
\$isunknown(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 5**      *Supported Verilog System Tasks (Continued)*

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

**Table 5**      *Supported Verilog System Tasks (Continued)*

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

Table 5 Supported Verilog System Tasks (Continued)

Task	Supported in ZEMI-3	Supported in Design	Default
\$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, \$dumpports	Limited Support	Limited Support	Yes
\$dumpfile, \$dumpoff, \$dumpon, \$dumpall, \$dumplimit, \$dumpflash, \$dumpportsoff, \$dumpportson, \$dumpportsall, \$dumpportslimit, \$dumpportsflush	No	No	-
<b>Note:</b> dumpvars/dumpports are used for QIwc and FWC waveform capture. For details on usage, see the <i>ZeBu Debug Guide</i> .			

## Examples of Using System Tasks

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.

Use the \$display task for error handling. However, new verification methodologies are more assertion based. For more information about assertions, see Using SystemVerilog Assertions.

# 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](#)

---

### Overall Unified Compile Flow

With the Unified Compile flow, the design and its test environment are compiled together by the VCS<sup>TM</sup> 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** for ZeBu compilation. **zCui** manages the VCS command execution and the ZeBu back-end compilation.

---

### Basic UTF Commands

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

Table 6 Basic UTF Mandatory Commands

Commands	Description
<code>architecture_file -filename {&lt;path_to_zse_configuration.tcl&gt;}</code>	Specifies the hardware architecture file.
<code>vcs_exec_command{&lt;script_name_with_pa th&gt;}</code>	Specifies the VCS command for a design.
<code>set_hwtop -module &lt;design_top_module_name&gt;</code>	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
```

## 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](#)
- [Compilation Settings for DesignWare Blocks](#)

---

## 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
```

- 4.

---

## 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 SYNOPSISYS_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
```

---

## 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.

---

## 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 7**      *VCS Options*

Examples for VCS Options	Description
Examples for VCS Compilation Options	
Verilog Analyzer (vlogan)	<code>+define+&lt;macro&gt;</code> : Defines a macro in the Verilog source <code>-f file</code> : Specifies files and command options <code>-l &lt;logfile&gt;</code> : Generates the log file <code>-q</code> - Quiet (no internal messages and banner) <code>-v &lt;lib_file&gt;</code> : Specifies a Verilog library file <code>-y &lt;libdir&gt;</code> : Specifies a directory of Verilog library files <code>+libext+&lt;ext&gt;</code> : Specifies library file extensions (used with <code>-y</code> ) <code>-work &lt;libdir&gt;</code> : Analyzes into a specified logical library <code>+v2k</code> : Enables Verilog 2001 constructs <code>-sverilog</code> : Enables SystemVerilog constructs <code>-timescale=1ns/1ps</code> : Specifies a default timescale <code>+incdir+&lt;dir&gt;</code> : Specifies search directory for included files <code>+librescan</code> : Searches unresolved module starting first library in the <code>vlogan</code> command
VHDL Analyzer (vhdlan)	<code>-work &lt;logical_lib&gt;</code> : Analyzes the specified logical library <code>-q</code> - Quiet (no internal messages and banner) <code>-nc</code> : Suppresses the copyright header <code>-vhdl87</code> : Enables VHDL-87 syntax (VHDL-93 is default) <code>-f &lt;options file&gt;</code> : Specifies source files and switches <code>-xlrn</code> : Allows a relaxed/non-LRM compliant code <code>-smart_order</code> : Identifies the file order dependencies
Examples for VCS Elaboration Options	

Table 7 VCS Options (Continued)

Examples for VCS Options	Description
Provided to vcs command	<p><code>-l &lt;logfile&gt;</code>: Creates the runtime log file</p> <p><code>-P pli.tab</code>: Compiles the user-defined PLI table</p> <p><code>&lt;.c .o files&gt;</code>: Adds C or object files to compile</p> <p><code>-xlm</code>: Allows a relaxed/non-LRM compliant code</p> <p><code>-ignore_driver_checks</code>: Suppresses multiple driver checks</p> <p><code>-liblist</code>: Specifies the library search order for unresolved module or entity instantiated in Verilog</p>

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**.

## Compilation Settings for DesignWare Blocks

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

1. Set the `$SYNOPTSYS` environment variable to the Synopsys synthesis (DesignCompiler) installation.
2. Map the logical libraries to appropriate files in the `synopsys_sim.setup` file.
3. In your VCS script, add the commands for the following steps:

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

```
mkdir dware
```

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

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

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

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

---

## 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
```

---

## 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*.

---

## 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`
- `Zebulse`
- `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.

---

## 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 8 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
```

---

## 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

---

## 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 <zcuu_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.
- **<zcuu\_uc\_work\_dir>** is the optional working directory for compilation; by default, this is **./zcuu.work**.

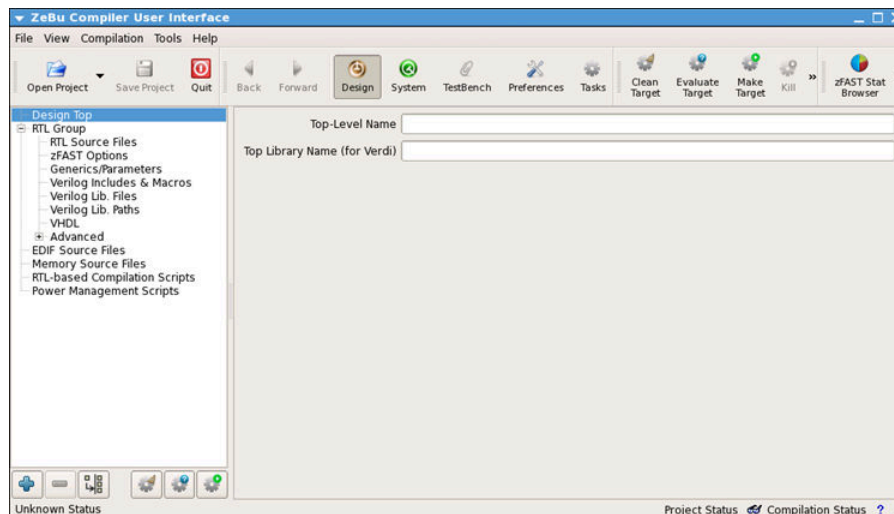
After compilation, you can explore the compilation status with **zBatchExplorer**, a graphical tool similar to the Tasks workspace in **zCui** GUI. For more information, see [Analyzing Compilation Results Using zBatchExplorer](#).

---

## Compiling With zCui in the GUI Mode

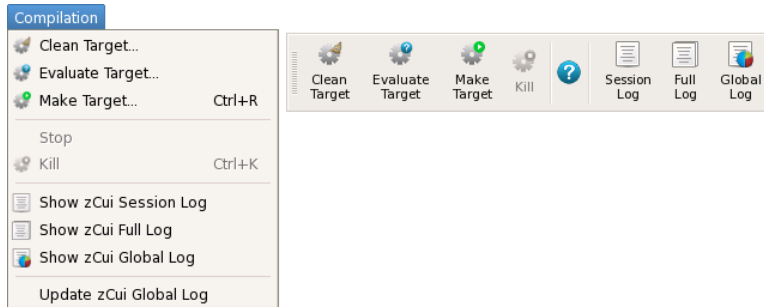
To compile a design, ZeBu offers a graphical interface, **zCui**, to configure the compiler and analyze the compilation results. The following figure displays the default view of **zCui**.

Figure 6      *zCui Default View*






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 7      *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:

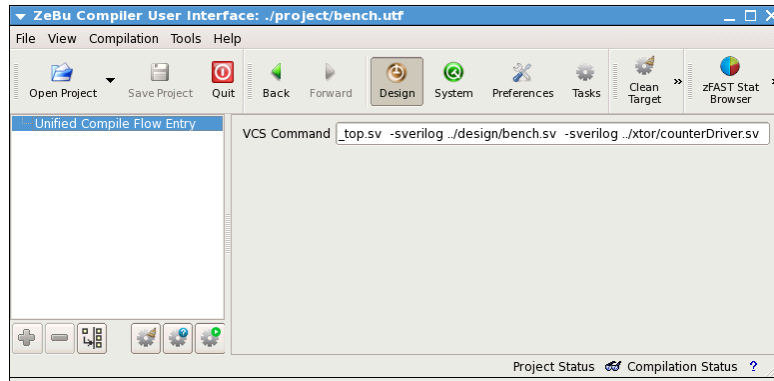
```
zCui -u <project_name>.utf [-w <zcu_uc_work_dir>]
```

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.

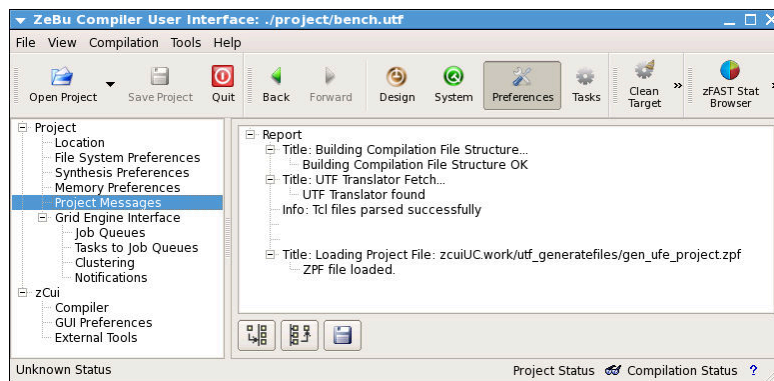
The GUI displays the **Design** workspace, with only the **Unified Compile Flow Entry** panel, which displays the VCS command line from your UTF file as displayed in the following figure:

Figure 8 *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 9 *Project Messages Panel*

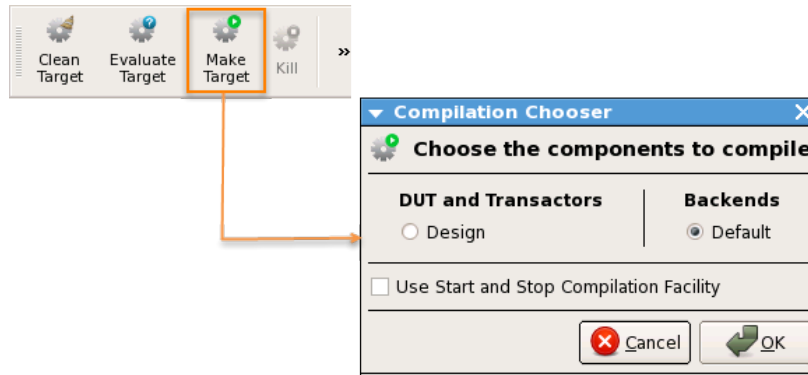


If you modify some compilation settings in **zCui**, the modified settings are applied on the next compilation.



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

Figure 10 Launching Compilation in zCui

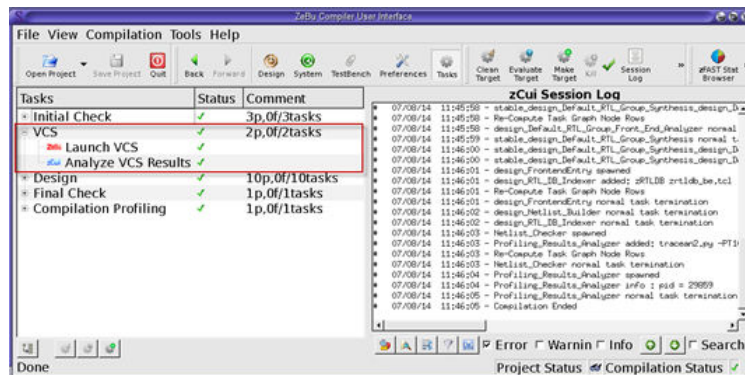


In the **Compilation Chooser** dialog (see Figure 10), you must select one of the following:

- 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 11 Viewing the VCS Task in zCui



## Compilation Analysis

This section consists of the following sub-sections:

- [Important Log Files](#)
- [Analyzing Compilation Results Using zBatchExplorer](#)

## Important Log Files

There are two important log files as follows:

- The full compilation flow is logged by **zCui** in <zcu.work>/zCui/log/zCui.log.
- The design size estimation can be found in <zcu.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|Qiw
# |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 <zcu.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 : +-----+
```

## Analyzing Compilation Results Using zBatchExplorer

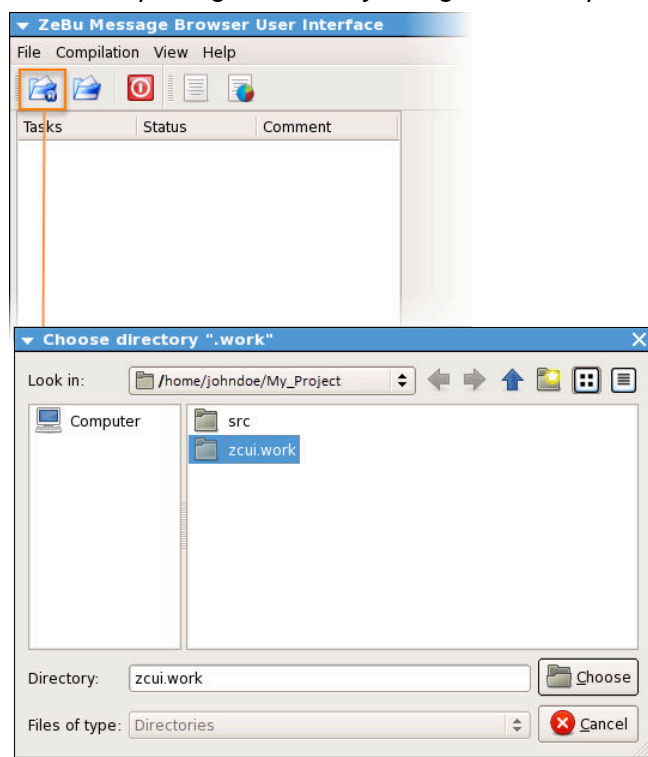
**zBatchExplorer** is a GUI that displays the task tree (similar to the **Tasks** workspace in **zCui**) after compilation. It analyzes log files of all the tasks, **zCui** full log, and global log.

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 12 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 13 Selecting Log File and zCui Global File



## 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.



# 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
- **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](#)
- [Using zRci to Control Emulation Runtime](#)
- [Runtime Performance Analysis With zTune](#)

---

## 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 is located 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:

- [Setting the Emulation Speed](#)
- [Setting Design Clock Parameters for Runtime](#)
- [Memory Initialization for Runtime](#)

---

## Setting the 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. You can fine-tune these settings by using the `designFeatures` file. This file is located in the directory where the runtime software is launched or in the compilation output directory (typically `zebu.work`).

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

## Example of `zTime.log` File

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

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

---

## 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.

## Example 1

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"
```

## Example 2

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"
```

---

## Memory Initialization for Runtime

To initialize a memory for runtime, specify 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
```

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

---

## Using zRci to Control Emulation Runtime

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.



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

```
zRci \
[-h|--help]
[<setup.ucli>|--do <setup.ucli>] \
[--zebu-work <zebu.work>] \
[--default-clock <clock>] \
[--default-clock-edge <phase>] \
[--so-testbench <lib>] \
[--c-call-library <lib>] \
[--attach <pid>|--testbench <cmd>] \
[-- tcl_args ...]
```

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

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

- [Common zRci Command-Line Options](#)
- [Controlling Clocks During Runtime Using zRci](#)
- [Managing Memories](#)

---

## Common zRci Command-Line Options

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

- [Using zRci With a C++ Testbench](#)
- [Starting zRci in Batch Mode](#)

### Using zRci With a C++ Testbench

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

```
zRci [<UCLI script>] \
--tesbench "<executable testbench command>"
```

**zRci** controls the emulation, including the start of the clocks.

When using **zRci**, the C++ testbench must be compiled with the `-pthread` option given to `g++`.

### Starting zRci in Batch Mode

You must provide a Tcl script to control emulation. This script must include all clock initialization.

To launch **zRci** in the batch mode, use the following:

```
zRci [<UCLI script>] \  
--zebu-work <zebu.work> \  
--default-clock <hw_top.clock_name> \  
--default-clock-edge <both|posedge|negedge>
```

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

---

## Controlling Clocks During Runtime Using zRci

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 Number Clock Cycles Executed](#)

### Note:

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

## Obtaining a List of Clock Groups

In most cases, only one clock group (default group) is used. But, it is possible to define several groups in the `designFeatures` file using the following `zRci` command:

```
get -clock_groups
```

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

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

## Enabling Clocks for N Cycles

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

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

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

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

**Syntax:**

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

where,

- `default_clock` returns the default clock name
- `primary_clocks` returns all clocks that belong to default clock group
- `secondary_clocks` returns all clocks that do not belong to the default clock group
- `all` returns all clocks
- `-cycles` returns the current cycle count for the returned clock.

## Enabling Clocks in the Free-Running Mode

**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 Command-Line User Guide*.

## Disabling Clocks

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

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

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

## Getting the Number Clock Cycles Executed

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

**Syntax:**

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

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

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

### Example

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

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

---

## Managing Memories

The external **zrm** memories, namely, BRAMs and LUTRAMs can be controlled from **zRci** Tcl scripts. You can save the content of a memory in a file using `memory -store <memory> -file <filename>` or in a buffer using `memory -store <memory> -buffer <buffer>`. In addition, 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, see the **memory commands** section in the *ZeBu Unified Command-Line User Guide*.

---

## 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 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 User Guide.

# 5

## 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 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](#))

---

### 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.

---

## Controlling Synthesized Assertions at Runtime

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.

This section consists of the following subsections:

- [Controlling Assertions From the C++ Testbench](#)
- [Controlling Assertions Using zRci](#)
- [Post-Processing SVA](#)

---

### 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.

#### 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](#)

See also [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);
```

---

## 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.

## Starting Assertion

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](#)

See also [Stopping SVA Processing](#).



## 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 dumped into a a ZTDB for post-emulation analysis while using:

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

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

## Stopping SVA Processing

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

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

---

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

where,

- `-l [ --log ] <filename>`: Specifies the log file name (default: `zsvaReport.log`)
- `-z [ --zebu-work ] <zebu.work>`: Specifies the ZeBu work directory.
- `-i [ --ztdb ] <.zsva filename>`: Specifies the ZTDB input directory.
- `-o [ --output ] <filename>`: Specifies the output file to save the report.
- `-s [ --severity ] <severity>`: Specifies the minimum severity level that is reported. Values can be `fatal|error|info|display|success`. Default is `error`.
- `-n [ --no-error ]`: Displays only a report and does not display the message report.

- `-m [ --max-errors-sva ] <number>`: Specifies the maximum number of error messages that are reported for each SVA.
- `--max-errors-all <number>`: Specifies the maximum number of error messages displayed for all SVA.
- `-b [ --begin ] <time>`: Specifies the begin time of the conversion (default: first captured time).
- `-e [ --end ] <time>`: Specifies the end time of the conversion (default: last captured time).
- `-t [ --timescale ] <timescale>`: Specifies time per cycle (for example, 10ns).
- `--force`: Forces conversion on unfinished ZTDB.
- `-h [ --help ]`: Displays the help.

For example,

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

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

# 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](#)

---

### 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 User Guide*.

---

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

This section describes the following subsections:

- [Enabling DPI Using zRci](#)
- [Enabling DPI Using the C++ API](#)

---

### Enabling DPI Using zRci

Before enabling the DPI, load the dynamic library containing the imported functions. This can be done using the following command:

```
ccall -load <so>
```

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

```
ccall -enable
```

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

The DPI calls can be disabled using the `ccall -flush` 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);
```

---

## 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;
}
```