

Simulation Acceleration User Guide

Version S-2021.09-2, October 2023



Copyright Notice and Proprietary Information

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

Destination Control Statement

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

Disclaimer

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

Trademarks

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

Free and Open-Source Software Licensing Notices

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

Third-Party Links

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

www.synopsys.com

Contents

Preface.....	13
About This Book	13
Intended Audience.....	13
Contents of This Book	14
Typographical Conventions	16
Synopsys Statement on Inclusivity and Diversity	17
 1. Introduction to Simulation Acceleration Cosimulation	19
1.1. Simulation Acceleration Flow	19
1.1.1. V2VX Mode.....	20
1.1.2. V2Z Mode.....	20
1.2. Simulation Acceleration Compile and Runtime Options	22
1.3. Simulation Acceleration Coding Guidelines.....	26
1.4. SystemC Support in Simulation Acceleration	28
1.5. Other Supported Features in Simulation Acceleration	29
1.5.1. Interface Appearing in Both SW and HW	29
1.5.2. Moving a HW Module/Interface/Task/Function/Block to Execute in SW	30
1.5.2.1. Using Compile-Time Options.....	30
1.5.2.2. Using an Embedded Source Code SystemVerilog Attribute, Comment Pragma, or UTF Command.....	31
1.5.2.3. Moving Initial Blocks From HW to SW Automatically	32
1.5.3. SVA Assertions are Supported in SW and HW That Is Not Mapped to IF FPGAs	35
1.5.4. VHDL Support and Limitations.....	36
1.5.5. Moving Glue Logic to Improve Performance.....	38
1.5.6. Support for Final Block in Hardware	40
1.5.7. Support for Concurrent Assertion in Hardware	41
1.5.8. Moving dumpvars/dumpports to the DUT Automatically	42
1.5.9. Miscellaneous Features.....	45
 2. Supported Syntax in Simulation Acceleration	51
2.1. Supported Syntax for Using Signals as the Communication Mechanism	51
2.2. Signal Communication Using Primitive Data Types.....	52

2.2.1. Signal Communication Using Aggregate Data Types.....	52
2.2.2. Signal Communication Using Interface Ports.....	53
2.2.3. Signal Communication Using Real Types	54
2.2.4. Signal Communication Using XMRs.....	55
2.2.5. Optimizing Performance for Clock XMRs From SW to HW (clkopt)	56
2.2.6. Optimizing Performance for XMRs With Force and Release Statements ..	57
2.2.7. SW Accessing HW Memory Using \$readmem and \$writemem System Tasks	58
2.2.8. Accessing and Forcing HW Signals From Within SW Using \$hdl_xmr and \$hdl_xmr_force	59
2.2.8.1. \$hdl_xmr: Access HW Signals From Within SW	59
2.2.8.2. \$hdl_xmr_force: Force HW Signals From Within SW	59
2.2.9. Signal Communication Through External Subroutines	60
2.2.10. Strength Analysis in Simulation Acceleration.....	61
2.2.11. Limitations With Signal-Based Communication	63
2.3. Supported Syntax for Using Subroutines as the Communication Mechanism	65
2.3.1. Interface Task/Function (Subroutines) Calls From SW to HW.....	66
2.3.2. Class Method Calls From HW Interfaces to SW	67
2.3.3. DPI Subroutine Call From HW Interfaces.....	69
2.3.4. Streaming Export Methods.....	69
2.3.5. UVM Messaging Support	69
2.3.6. Import Calling an Export.....	71
2.3.7. \$test\$plusargs, \$value\$plusargs Support.....	72
2.3.8. System Tasks.....	73
2.3.9. System Task Support for \$display	74
2.3.10. Enabling Concurrent Export Task/Function Calls	77
2.4. Supported and Unsupported Behavioral Compiler Syntax	79
2.4.1. Supported Behavioral Compiler Syntax in Simulation Acceleration	79
2.4.1.1. #delay Support	80
2.4.1.2. Clocking Edge in for Loops	81
2.4.1.3. Multiple Clocks or Edge Expressions in the Same Process	81
2.4.1.4. Complex Edge Combination	81
2.4.1.5. Combinations of Edge and Level Events	81
2.4.1.6. Bounded Loops.....	82
2.4.1.7. Unbounded Loops	82
2.4.1.8. Wait Statements	82
2.4.1.9. Named Events	82
2.4.1.10. Behavioral Compiler Supports all SystemVerilog Data Types	83
2.4.1.11. Clocking Blocks.....	83

2.4.1.12. Fork/Join	84
2.4.1.13. SystemVerilog Strings, \$sformatf, %m are Supported	84
2.4.1.14. Usage of Memory in Behavioral Code	86
2.4.2. Unsupported Behavioral Compiler Syntax in Simulation Acceleration..	88
3. Scheduling Semantics and Race Conditions.....	91
3.1. Initialization Between SW and HW	92
3.2. Signal Handshake Between SW and HW	92
3.3. Resolving Clock/Data Race Conditions Between SW and HW	92
3.4. Subroutine Handshake Between SW and HW	93
4. Incremental Compile.....	95
4.1. Modes for Performing Incremental Compile with zCui.....	95
4.2. Support for \$hw_force ECO	96
4.3. Key Messages in Simulation Acceleration Incremental Compile	98
5. SVA Assertion and Coverage Support	101
6. Generating the Profiler Report	105
6.1. Enabling Profiler	106
6.2. Profiler Report Sections	107
6.3. Example: Communication Profiler Report	108
7. Debugging and Reports	115
7.1. Debug Options for Communication Mechanisms	115
7.2. Functional Debug in Simulation Acceleration	117
7.2.1. Enabling UCLI Signal Access for ZeBu Compile	117
7.2.2. Enabling UCLI Waveform Output for ZeBu Compile.....	118
7.2.3. Supported UCLI commands in Simulation Acceleration	119
7.2.3.1. get UCLI Command.....	119
7.2.3.2. memory UCLI Command.....	120
7.2.3.3. stop UCLI Command	121
7.2.3.4. run UCLI Command	124
7.2.3.5. force, release, show UCLI Commands	125
7.2.3.6. restart UCLI Command	126
7.2.3.7. System Information Commands.....	126

7.2.3.8. dump Commands.....	127
7.2.3.9. sniffer Command	128
7.2.4. Verdi Interactive Debug for Simulation Acceleration.....	130
7.2.5. Waveform Outputting Using UCLI.....	130
7.2.5.1. Dynamic-Probe.....	131
7.2.5.2. FWC Probe.....	132
7.2.5.3. QiWC Probe.....	132
7.2.5.4. Outputting Signals of HW Module Instances That Were Moved to SW	133
7.2.5.5. Multi-ZTDB Output	133
7.2.5.6. Output FWC/QiWC Waveform Only on Time Advance.....	134
7.2.5.7. Using Verdi for Debug.....	135
7.2.5.8. Output ZeBu XTOR Internal Signals	135
7.2.5.9. Waveform Expansion in Simulation Acceleration	136
7.2.5.10. Waveform Outputting config slice for zSimzilla	137
7.3. Using simprofile with Simulation Acceleration.....	138
8. Timing Analysis.....	143
8.1. UTF File Changes	143
8.2. Viewing the zTime Report for Critical Output Routing Data Paths	144
8.3. Analyzing a Critical Path Using zTime.....	147
9. Using Time Decoupled Mode for Performance Improvements.....	149
9.1. Consideration for Using Time Decoupling	149
9.2. Use Model for the Time Decouple Mode	150
9.2.1. Compile-Time Use Model.....	150
9.2.2. Runtime Use Model	151
9.3. Limitations of Time Decouple Mode	152
10. Enabling zDPI in Simulation Acceleration.....	153
10.1. Use Model for Enabling zDPI in the V2Z Flow	153
10.2. Use Model for Enabling zDPI in the V2VX Flow	154
10.3. Limitations of zDPI Support in Simulation Acceleration.....	154

11. Estimating DUT Power Requirements Using Power Profiling 155

11.1. Enabling Universal IP (UIP) Power Profiling155

11.2. Use Model of the uip Command155

List of Tables

Supported Elements for Signal Communication Using XMRS.....	55
Pragma Support for \$display Optimizations	76
Runtime Options to Enable Profiler	106
Runtime Options for Logging Communication Transactions Between SW and HW	115
Dynamic Trigger Options for the stop Command	122
UCLI Commands to Retrieve System Information	126
zTime Report: Critical Output Routing Data Paths Column Descriptions....	146
zTime Report: Column Description of Segments in Critical Path	148

List of Figures

Simulation Acceleration Flow 21

Communication Profiler Report108

Profiling Report: Counter Time Construct View140

Profiling Report: Construct Information140

Profiling Report: Source Code Information.....141

zTime Report: Critical Output Routing Data Paths145

zTime Report: Analyzing Critical Output Routing Data Paths.....147

zTime Report: Segments in Critical Path147

About This Book

The **Simulation Acceleration User Guide** describes how to use Simulation Acceleration to enable cosimulating SystemVerilog testbenches with the DUT without any changes to the code. Simulation Acceleration enables communicating between software (SW) and hardware (HW) parts using SystemVerilog mechanisms, such as ports on a module or interface and calling of subroutines implemented in an interface.

Intended Audience

This guide is written for design engineers to assist them with using Simulation Acceleration in ZeBu.

These engineers should have knowledge of the following Synopsys tools:

- ZeBu Server
- VCS
- Verdi

Contents of This Book

This document has the following sections.

Section	Describes
<i>Introduction to Simulation Acceleration Cosimulation</i>	Steps to deploy Simulation Acceleration cosimulation flow (V2VX and V2Z modes), Simulation Acceleration Compile and Runtime Options, Simulation Acceleration Coding Guidelines, and SystemC Support in Simulation Acceleration
<i>Supported Syntax in Simulation Acceleration</i>	Supported syntax for using signals as the communication mechanism, supported syntax for using subroutines as the communication mechanism, and supported and unsupported behavioral compiler syntax
<i>Scheduling Semantics and Race Conditions</i>	Describes various topics, such as resolving Clock/Data race conditions between SW and HW
<i>Incremental Compile</i>	How to enable incremental compile on a given build
<i>SVA Assertion and Coverage Support</i>	How to enable functional coverage and assertions at compile time; Also, describes controlling SVAs at runtime
<i>Generating the Profiler Report</i>	The SimXL Profiler report generated by the Simulation Acceleration [Communication] Profiler tool for Simulation Acceleration.
<i>Debugging and Reports</i>	Debug options for communication mechanisms, functional debug in Simulation Acceleration, and using simprofile with Simulation Acceleration
<i>Timing Analysis</i>	Viewing the zTime report for critical output routing data paths and analyzing a critical path using zTime
<i>Using Time Decoupled Mode for Performance Improvements</i>	The use model to mark functions, which only have inputs, so that these functions are not monitored or controlled by Simulation Acceleration transactors. This is known as time decoupling of functions, which improves performance.
<i>Enabling zDPI in Simulation Acceleration</i>	Use model for enabling zDPI in V2Z and V2VX flows
<i>Estimating DUT Power Requirements Using Power Profiling</i>	The Power Profiling feature in Simulation Acceleration enables you to estimate the power requirements of the DUT, which resides in the hardware; this section describes how to enable UIP profiling

Related Documentation

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

Typographical Conventions

This document uses the following typographical conventions:

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

The following table describes the syntax used in this document:

Syntax	Description
[] (Square brackets)	An optional entry
{ } (Curly braces)	An entry that can be specified one time or multiple times
(Vertical bar)	A list of choices out of which you can choose one
. . . (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 Introduction to Simulation Acceleration Cosimulation

Simulation Acceleration enables cosimulating SystemVerilog testbenches with the DUT without any changes to the code.

The usage of Simulation Acceleration in ZeBu has the following advantages:

- It provides a mechanism for easy creation of a generic simulation testbench setup that can be easily ported to the ZeBu emulation platform.
- It provides a well-structured approach as recommended by standard and advanced SystemVerilog based verification methodologies, such as UVM.
- It provides an incremental approach to bringing up a working simulation environment into an emulation environment. The approach starts with a signal-based communication mechanism between SW (VCS) and HW (ZeBu). Then, progresses with working through the different testbench interfaces toward a transaction-based communication mechanism.

Simulation Acceleration enables communicating between software (SW) and hardware (HW) parts using SystemVerilog mechanisms, such as ports on a module or interface and calling of subroutines implemented in an interface.

For more information on Simulation Acceleration, see the following topics:

- [Simulation Acceleration Flow](#)
- [Simulation Acceleration Compile and Runtime Options](#)
- [Simulation Acceleration Coding Guidelines](#)
- [SystemC Support in Simulation Acceleration](#)
- [Other Supported Features in Simulation Acceleration](#)
- [VHDL DUT wrapped in a SystemVerilog module is supported in HW](#)[Other Simulation Acceleration Limitations and Temporary Known Limitations](#)

1.1 Simulation Acceleration Flow

You can start to deploy the Simulation Acceleration cosimulation flow on a working VCS test environment. The following modes are used:

- [V2VX Mode](#)

■ *V2Z Mode*

1.1.1 V2VX Mode

The V2VX mode is used to compile and run a test case in the environment using VCS only to validate the readiness of environment for Simulation Acceleration.

- If the compile fails, there could be unsupported syntax used. In this case the compiler provides a compilation Not Yet Implemented (NYI) error message, indicating the specific syntax that is not supported.
- If running the test fails, there could be multiple reasons for failure, such as the following:
 - ❑ Race conditions between HW and SW that were not exposed in VCS.
 - ❑ Incorrect Simulation Acceleration Transformations and Mapping.
 - ❑ Usage of unsupported features.

If the test fails, it is recommended to do a waveform comparison between a regular VCS run and a V2VX VCS run, using Verdi nCompare capabilities.

1.1.2 V2Z Mode

The V2Z mode is used to compile and run a test case in a cosimulation environment. SW is compiled with VCS and HW is compiled using ZeBu.

- If the compile fails, there could be multiple reasons for failure, such as the following:
 - ❑ Incomplete port connectivity to HW
 - ❑ Synthesis issues caused by usage of non-synthesizable syntax
 - ❑ Place-and-Route issues
- If running the test fails, there could be multiple reasons for failure, such as the following:
 - ❑ Race conditions between HW and SW that were not exposed in VCS
 - ❑ Incorrect Simulation Acceleration Transformations and Mapping
 - ❑ Incorrect RTL code generated by Behavioral Compiler.
 - ❑ Usage of unsupported features.

Simulation Acceleration Flow

- ❑ Non congruence between simulation and emulation, for example emulator does not support 'z' and 'x' values.
- If the test fails, it is recommended to do a waveform comparison between a regular VCS run and a V2Z run, using Verdi nCompare capabilities

The `simv` executable is used to run the Simulation Acceleration environment to generate waveforms and log files.

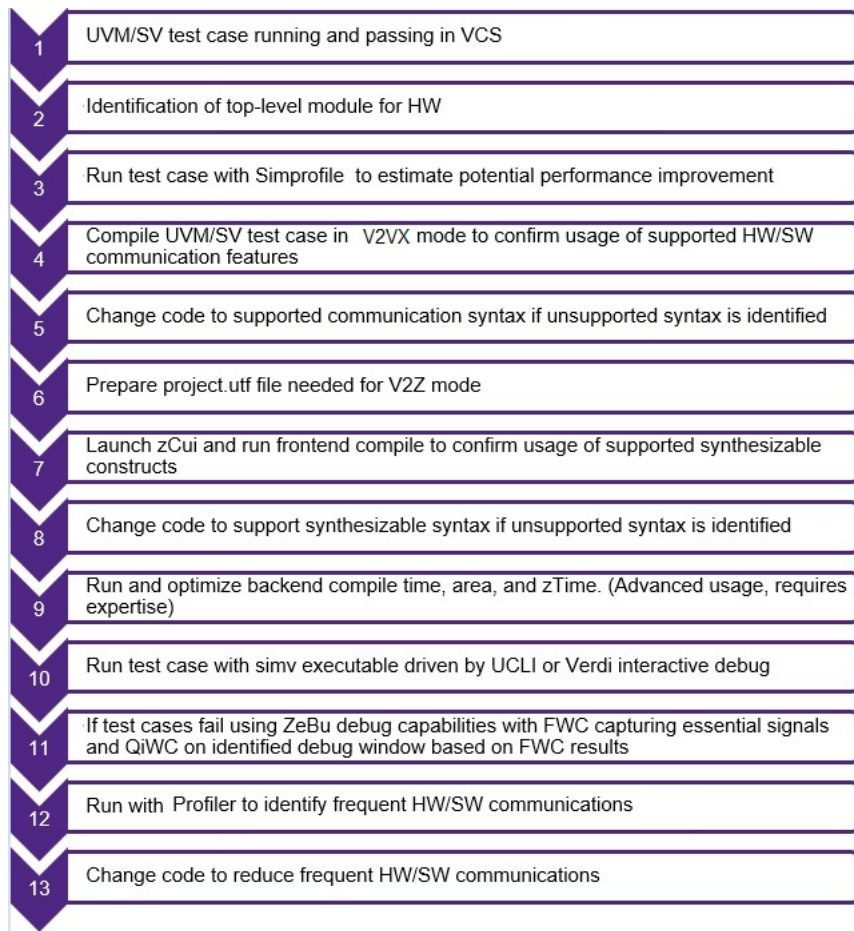


FIGURE 1. Simulation Acceleration Flow

1.2 Simulation Acceleration Compile and Runtime Options

The options for compilation and runtime depend on the mode. For more information, see the following subsections:

- [V2VX Mode Compilation and Runtime Options](#)
- [V2Z Mode Compilation and Runtime Options](#)
- [VCS Compile-Time Option to Generate Additional Diagnostic Data](#)
- [On-Screen Messages](#)

V2VX Mode Compilation and Runtime Options

- Uses only VCS for both HW and SW compilation. Used for pipe cleaning before moving to ZeBu compile.
- To enable the V2VX mode, use the following VCS Compile option:
`-simxlttest=v2vx`
- To specify HW top-level instance, use the following VCS Compile option:
`-Xhdl_cosim_dut <hierarchical path to the DUT instance>`
If the module is compiled as a top-level module, it is the name of the module.
- To specify that all instances of a module are HW top-level instance, use the following VCS Compile option:
`-Xhdl_cosim_dut_module <module-name>`
- Run using existing `simv` options.

V2Z Mode Compilation and Runtime Options

- Uses ZeBu for HW compilation and VCS for SW compilation.
- The ZeBu `zCu` tool does the full compile with a UTF file as input and the UTF file pointing to the VCS compile command using the `vcs_exec_command` UTF command.
- For compile, add the following UTF options and VCS compile options:
 - ❑ UTF command to enable Simulation Acceleration:
`simxl -enable true`

- ❑ UTF command to specify HW top-level instance when an instance path is required to select a specific instance subtree:
`simxl_set_hwtop -instance <hierarchical path to the DUT instance>`
 All logic from that instance and below goes into HW. If the module is compiled as a top-level module, the `-module` option below is required.

- ❑ The UTF command to specify that all instances of a module are HW top-level instances is as follows:

```
simxl_set_hwtop -module <module-name>
```

Use this command if the HW top is a single instance module.

Multiple `simxl_set_hwtop` (respectively for `v2vx`, `-Xhdl_cosim_dut`, `-Xhdl_cosim_dut_module`) commands can be used to specify more than one top, the following restrictions are applicable to both V2VX and V2Z modes:

- ❑ All HW top levels need to be instantiated in the same module.
- ❑ For example, specifying "top1.dut1" and "top2.dut2" as dut-instances would give an NYI error because dut1 and dut2 are in two different top modules, top1 and top2.
- ❑ A HW top module cannot be instantiated in another HW top module.
 For example, suppose Top has instances of module "A" and "B", named instA and instB. Module "A" has again an instance of module "B". In this case, you cannot specify both "top.instA" and "top.instB" as HW top instances.
- ❑ A HW top instance cannot be within a hierarchy that has been moved to SW.
- ❑ A HW top instance cannot be an instance-array of size more than 1. For example:

```
HWTOP dut_inst[1:0]();
```

If you specify "HWTOP" as hwtop module, an NYI error is reported.

Usage of `$dumpvars` for specifying waveform dumping directives can be used in any of the HW top modules. It is also possible to have a `$dumpvars` HW top module (For example, `dumpvars_module.sv`) to specify waveform dumping directives for any of the signals/instances in the instance subtrees of the other HW top modules.

- To execute the ZeBu compile, perform the following:
 - ❑ Open the **zCui** compilation GUI using the following:
`zCui -u project.utf`
 - ❑ Click the **Make Target** button in the **zCui** GUI to start the compile process.

- In the case where compilation of the **zCui** front end needs to be exported to enable **zCui** back-end compile at a different location, perform the following:

- ❑ Prior to the front-end compile, run the following command to set the `ZCUI_EXPERT` environment variable:

```
setenv ZCUI_EXPERT exportFrontEnd
```

- ❑ Execute the front-end compile only using **zCui**.
- ❑ After front-end compile, move the tar or zip file, `frontend.tgz`, to the different location and extract.
- ❑ Run the back-end compile using the following command and options:

```
zCui --rf feBeSplit/frontendRestore.xcui
```

VCS Compile-Time Option to Generate Additional Diagnostic Data

VCS additional compile-time option to generate diagnostic data in the current run directory:

```
-simxl=diag
```

To generated VCS compile diagnostic file at the VCS stage, which is the early stage of **zCui** compilation, specify the following:

```
-simxl=comp_diag
```

The `simxlCompDiag.txt` file is generated. This file contains the following:

- All Top level modules including TB
- All DUT top modules
- All embedded TB modules (ETB)
- VCS command line: Switches used for VCS elaboration
- Simulation Acceleration interface signals
 - ❑ List of INPUT signal full instance name exported with bit width and module/interface name. In the end, total input signals exported with total bit width.
 - ❑ List of OUTPUT signal full instance name exported with bit width and module/interface name. In the end total output signals exported with total bit width.
- Simulation Acceleration task functions
 - ❑ List of exported tasks (full task scope) with detailed argument list(bit width, direction, name) and its module name.

- ❑ List of imported tasks (HW module caller instance) with detailed argument list (bit width, direction, name) and its module name.
- Simulation Acceleration zcei message port cost: Rough estimate of "In/Out" port message port cost per instance
- Transactor name and full instance path: All the transactor name/hierarchy

This report does not contain:

- Transactor information is dumped at VCS side only in NO_DVE_FLOW. For other flow, transactor information is dumped at ZEBU side. Since we only track VCS side diagnostics, the transactor information will not show up consistently in `simx1CompDiag.txt` file.
- Only embedded TB modules (ETB) are reported. TF/blocks are not covered.

On-Screen Messages

When you run `simv`, by default, only critical information, such as the following, is displayed on-screen (in terminal):

- Testbench messages
- Necessary ZeBu messages, such as log file location and license checks status
- Error messages and fatal messages

To view all information, see the log files.

If you need all information to be displayed on-screen, set the `ZEBU_DISABLE_PRINT` environment variable to any value other than OK, such as the following:

```
setenv ZEBU_DISABLE_PRINT 0
```

By default, the `ZEBU_DISABLE_PRINT` environment variable is to OK and therefore, only critical messages are displayed on-screen.

1.3 Simulation Acceleration Coding Guidelines

The following is a set of guidelines used to reduce the number of issues related to compile, runtime, and performance:

- Use supported coding styles mentioned in this document.
- Do not use unsupported coding styles mentioned in this document.
- Never instantiate a SW module in the HW side, use SystemVerilog bind for associating a SW module to HW in a modular fashion. You can then use compile-time options, attributes, or UTF commands to move the SW module to execute in SW.
- Do not use #-delays in code residing in the SW side, use clocks coming from the HW side to perform synchronization.
- Make sure to use single timescale across HW and SW.
- Avoid using precision units in the timescale as much as possible.
- Make sure all clocks driving HW are implemented in HW using RTL clocks only.
- When implementing RTL clocks using #-delay and procedural code, make sure that the procedural code does not contain mixed in implementations for other features. The tool requires RTL clock implementations to be standalone within its own procedural block.
- Try to use clocks derived from a single fastest clock as much as possible. There are techniques to do this even for fine-grained clock frequencies. Consult with Synopsys AEs.
- It is possible to define utility tasks, functions, and DPI in the HW top-level wrapper. The tool supports calling import methods from the HW top-level wrapper, which utility tasks, functions, and DPI may need.
- Avoid doing excessive backdoor writes to non-memory sequential elements because the mechanism that does backdoor writes is extremely slow in ZeBu. It is preferred to do front door writes.
- Do not initialize memory arrays using procedural SV code because memory arrays are initialized to zero by default when running on ZeBu. Use +vcs+initreg+random when compiling with VCS and +vcs+initreg+0 when running with simv to achieve similar behavior for simulation only.
- Always baseline functionality with current step before moving to next steps. See steps mentioned in [Introduction to Simulation Acceleration Cosimulation](#).

- Always bring-up a Signal-Based Acceleration (SBA) environment (with steps mentioned in [Introduction to Simulation Acceleration Cosimulation](#)) first before bring-up of Transaction-Based Acceleration (TBA) environment. Signal-Based Acceleration (SBA), where the HW top is the actual top.
- Manage all synchronization from the HW side; avoid scheduling on the SW side to reduce the SW overhead.
- Avoid using behavioral code as much as possible. RTL code is preferable as it is easier to debug and has less transformations.
- Use zero-time consuming functions and tasks when communication between HW and SW as much as possible. Implement state machines to manage the transaction.
- Do not access elements in HW from your UVM tests, sequences, and higher-level components. Leave them as abstract as possible to avoid HW compiles when they change. Essentially HW elements should be referenced only from UVM drivers and monitors only.
- Do not use clocking blocks in your interfaces this does not work well for acceleration.
- Implement tasks and state machines that UVM driver uses in your interface rather than driving signals directly from your UVM driver.
- Implement monitor loops in HW interfaces that call import tasks and functions implemented in SW in the UVM Monitor.
- Avoid using SV code that requires signal strength resolution to resolve multidriver logic on the boundary between SW and HW. This includes SV constructs, such as `tran*`, and so on.
- Avoid using bidirectional ports on the boundary between HW and SW as much as possible.
- Avoid using a mixture of blocking and non-blocking statements in the same always block in the SimXL transactor.
- Avoid using same variable driven from multiple always blocks in the SimXL transactor.

1.4 SystemC Support in Simulation Acceleration

Pass the `-sysc=show_sc_main` option to the elaboration command. You are required to pass the DUT path that is not prefixed with "sc_main" because of the current use model of SystemC in the Simulation Acceleration flow.

SystemC Limitations

SystemC specific limitations are as follows:

- Inout ports of two HW top modules connected in SystemC might not work because the port connectivity might not get traced through SystemC.

Example

```
module dut1(inout p1);  
    assign p1 = en ? drv1: 'bz;  
endmodule
```

```
module dut2(inout p2);  
    assign p2 = en ? drv2: 'bz;  
endmodule
```

- If ports `dut1.p1` and `dut2.p2` are connected to same highconn in SystemC, it is a multiple driver scenario. This can work only if the information is passed to ZeBu at compile-time (that these two are shorted). However, since connectivity cannot be traced in SystemC, this scenario would not work.

Note: There is no warning/error message for the above scenario.

- DUT ports connected through SystemC would not be optimized (only impacts Performance, no functional issue).

Example

If output port of `dut1` is connected to input port of another `dut2`, the value from the output port would go from HW to SystemC and then come back to SW through the input port. It only impacts performance, but not the functionality.

- Additional UTF commands to give directives on how HW subroutines are configured can be provided by identifying the interface that the HW subroutine resides in. For example, you can specify the following to reduce the size of the

Other Supported Features in Simulation Acceleration

communication mechanism of subroutines in `axi_interface` to enable ZeBu compile to converge:

```
zemi3 -module {axi_interface} -max_out_port_width 512 -
max_in_port_width 512
```

1.5 Other Supported Features in Simulation Acceleration

The other supported elements required for the Simulation Acceleration flow are as follows:

- *Interface Appearing in Both SW and HW*
- *Moving a HW Module/Interface/Task/Function/Block to Execute in SW*
- *SVA Assertions are Supported in SW and HW That Is Not Mapped to IF FPGAs*
- *VHDL Support and Limitations*
- *Moving Glue Logic to Improve Performance*
- *Support for Final Block in Hardware*
- *Moving dumpvars/dumpports to the DUT Automatically*
- *Miscellaneous Features*

1.5.1 Interface Appearing in Both SW and HW

Interface appearing in both SW and HW. See the following example.

Example

```
interface ift;
endinterface
module dut(ift if_dut);
endmodule
module tb(ift if_tb);
endmodule
module top;
```

```

    ift m_if;
    dut dut(m_if);
    tb tb(m_if);
endmodule

```

1.5.2 Moving a HW Module/Interface/Task/Function/Block to Execute in SW

See the following sections for moving a HW module or HW interface to execute in SW while maintaining scope of the intended HW:

- [Using Compile-Time Options](#)
- [Using an Embedded Source Code SystemVerilog Attribute, Comment Pragma, or UTF Command](#)
- [Moving Initial Blocks From HW to SW Automatically](#)

1.5.2.1 Using Compile-Time Options

It is common to bind verification SystemVerilog interfaces or modules to HW instances or to instantiate an interface in the top-level HW module that has access through a virtual interface from SW, which executes by default in HW.

With this capability, it is possible to move all instances of the specified interface or module to execute in SW.

It might also be helpful to move any HW module in the DUT to execute in SW if it is nonsynthesizable. Precautions should be taken if clock or data races occur.

Example

- For [V2VX Mode Compilation and Runtime Options](#), the following needs to be added to the VCS compile-time options:

```
-Xhdl_cosim_etb <module name>
```

Must be specified for each module or interface that is moved from HW to SW.

- For [V2Z Mode Compilation and Runtime Options](#), the following needs to be added to the UTF file:

```
simxl_move_to_tb -module {<module name list>}
```

All modules specified in the list above executes in SW, using their HW context.

1.5.2.2 Using an Embedded Source Code SystemVerilog Attribute, Comment Pragma, or UTF Command

Tasks, functions, or blocks implemented in modules or interfaces that are in HW can be moved to execute in SW.

Moving tasks, functions, or blocks from HW to SW is useful for a couple of reasons:

- Usage of code that is not synthesizable or not supported by the Behavioral Compiler
- Code that would be easier to debug executing in SW
- Usage of code that hinders on compile time and frequency (zTime results).

Example

```
module hwmod;

    // synopsys simxl_move_tf_to_tb
    task task1;
        // task contents
    endtask
endmodule

module hwmod;

    (* simxl_move_tf_to_tb *)
    task task1;
        // task contents
    endtask
endmodule

// UTF command example
simxl_move_tf_to_tb -target hwmod.task1
```

```

initial begin
    $sformat(inst,"MTK_%m.LOG");
    FID = $fopen(inst, "w");
end

always @ (posedge clk)
    if (<cond>)
        $fdisplay(FID, "VALUE OF DATA IS %h\n", DATA);

```

The example above uses constructs like `$sformat/%m` and other constructs to construct the file name (currently not supported in HW). Adding a pragma before the initial block moves that block to execute in the SW side. File handle can still be used in HW for doing `$fdisplay/$fwrite`. The initial block **is moved to HW** by adding the attribute `(* simxl_move_blk_to_tb *)`. See the following example.

```

(* simxl_move_blk_to_tb *)
initial begin
    $sformat(inst,"MTK_%m.LOG");
    FID = $fopen(inst, "w");
end

```

To move tasks or function from HW to SW when in the V2VX mode, use the following compile-time option:

```
-simxl=move_tf_to_tb+<module-name>+<task-function-name>
```

Where,

- `<module-name>`: Specifies the name of the module or interface where the task resides in.
- `<task-function-name>`: Specifies the name of the task/function that its execution is being moved from HW to SW.

1.5.2.3 Moving Initial Blocks From HW to SW Automatically

In a HW module if initial blocks are not synthesizable on ZeBu, you can move specific

initial blocks by using the `(* simxl_move_to_tb *)` pragma. However, this method is cumbersome for moving multiple initial blocks.

Simulation Acceleration enables you to automatically move initial blocks, which are not synthesizable on ZeBu, to SW and simulate them in VCS.

For more information, see the following subsections:

- [Enable Automatic Movement of Initial Blocks from HW to SW](#)
- [Movement Considerations](#)
- [Limitations](#)
- [Example](#)

Enable Automatic Movement of Initial Blocks from HW to SW

To move initial blocks to SW side automatically, specify the following VCS elaboration compile-time option:

```
-simxl=move2tb_zebuinit
```

See [Example](#).

Movement Considerations

The following considerations:

- As Simulation Acceleration provides multiple methods for moving initial blocks, you can use all methods. The movement priority is as follows, starting with the highest precedence:

- a. Specifying initial block with `(* simxl_not_move_to_tb *)`
- b. Specifying initial block with `(* simxl_move_to_tb *)`
- c. Using `move2tb_zebuinit` and `move2tb_allinit`

Therefore, even if you use the `move2tb_zebuinit` VCS elaboration compile-time option, initial blocks specified with the `(* simxl_not_move_to_tb *)` pragma are not moved.

- **Performance:** Any initial block that contains an effective `#delay`, i.e. the clock delay is enabled in that module by the `clock_delay` UTF command, is not moved. This ensures that high frequency drivers, such as clock generators, are not moved.

Initial blocks that do not have `clock_delay` UTF command enabled, can still be moved to the SW side. Such blocks have no delay and therefore the `#delay` is replaced with `#0`.

See [Example](#).

- **Race Condition:** For both structural and procedural writes, all initial assignments in SW occur before HW events. To avoid race conditions, Simulation Acceleration stops the driver clock in the HW side at the start of cosimulation and restarts the driver clock after all initial values are sent from the SW to the HW.

Limitations

You cannot move an initial block with any task or function call for import tasks in DUT.

Example

This example shows the following:

- Effect of specifying the `-simxl=move2tb_zebuinit` VCS elaboration compile-time option
- Impact of specifying `#delay` in the UTF file

Suppose you have the following file, `test.v`:

```
// SW Module
module testbench ();
  dut top();
endmodule

// HW Module
module dut();
  reg a, b;
  mid mid(a, b);

  initial a = 1; // (1)
  initial #1 b = 1; // (2)
```

Other Supported Features in Simulation Acceleration

```
endmodule
```

```
...
```

When you compile `test.v` with `-simx1=move2tb_zebuinit`, the initial block (1) and initial block (2) are simulated on the SW side.

To understand the impact of `#delay`, suppose the UTF file contains the `clock_delay` command for the dut module, as shown in the following:

```
// dut.utf
```

```
...
```

```
clock_delay -module dut
```

After compiling `test.v`, only initial block (1) is moved. This is because initial blocks with `#delay` are not moved.

Note that the `#delay` in (2) is ignored in HW because no `clock_delay` is specified. That is, reg b is set to 1 at time=0. This behavior should be kept after moving. Therefore, Simulation Acceleration moves the following two blocks to SW side.

```
initial a = 1; // (1)
```

```
initial #0 b = 1; // (2), #1 is replaced by #0
```

1.5.3 SVA Assertions are Supported in SW and HW That Is Not Mapped to IF FPGAs

SVA assertions are supported in SW and in HW that is not mapped to IF FPGAs . Use the following Simulation Acceleration VCS runtime options for special control on the HW side. Use regular VCS features for assertions executing on SW side.

- Enable HW SVAs:

```
-simx1=enable_dut_sva
```

- Disable specific HW SVAs:

```
-simx1=disable_dut_sva, assertFile=<filename>
```

Where, `<filename>` contains a list of assertion names to be disabled.

- Disable all assertion in DUT:

```
-simx1=disable_assertion_in_dut
```

1.5.4 VHDL Support and Limitations

VHDL Support

Simulation Acceleration supports SystemVerilog/VHDL mixed designs with VHDL. The support includes:

- VHDL hierarchy in HW:
 - ❑ VHDL constructs as supported by ZeBu.
 - ❑ Target signal can be port, variable or signal.
 - ❑ Selects (bit and part-select) are supported.
- VHDL hierarchy in SW.
- SystemVerilog Related:
 - ❑ SystemVerilog hierarchy under VHDL in HW.
 - ❑ Any level of SystemVerilog/VHDL nesting in HW.
 - ❑ Any level of SystemVerilog/VHDL nesting in the testbench.
- XMR Related:
 - ❑ Cross domain XMRs from SystemVerilog SW to VHDL hierarchy in HW.
 - ❑ Cross domain XMRs from SystemVerilog SW to SystemVerilog hierarchy under VHDL in HW.
 - ❑ XMR read in SW from signal in HW VHDL hierarchy.
 - ❑ XMR read in SW from signal in HW SystemVerilog hierarchy under VHDL.
 - ❑ XMR force from SW into HW SystemVerilog hierarchy under VHDL.
 - ❑ XMR write to HW Verilog hierarchy under VHDL, excluding Verilog ports.
 - ❑ XMR through VHDL block statement.
 - ❑ Case insensitive specification of names in VHDL hierarchy.
 - ❑ Direct XMR for read, force, and write (no generates).
 - ❑ Direct XMR for read, force, and write (with VLOG/VHDL generates).
 - ❑ Read XMR in `$hdl_xmr` (with `$hw_read`).

Other Supported Features in Simulation Acceleration

- ☐ XMR in `$hdl_xmr_force/$hdl_xmr_release` (with `$hw_force`).
- ☐ Write XMR in `$hdl_xmr` (with **`$hw_write`**), excluding Verilog ports.
- ☐ Complete shadow hierarchy of MX DUT hierarchy/XMR from VLOG DUT to VLOG TB (DUT2TB)/VLOG embedded testbench under VHDL.
- UCLI Related:
 - ☐ UCLI read of VHDL DUT signal from TB
 - ☐ UCLI force of VHDL DUT signal from TB
 - ☐ UCLI read of VLOG DUT signal from TB (under VHDL)
 - ☐ UCLI force of VLOG DUT signal from TB (under VHDL)
- Case insensitive specification of names in the VHDL hierarchy.
- Ability to move a SystemVerilog HW module/instance to SW under VHDL in HW.
- Support for both V2VX and V2Z flows.
- VHDL is supported in testbench hierarchy.
- Support for VLOG embedded testbench under VHDL.

Limitations

The following limitations exist with VHDL support in Simulation Acceleration:

- VHDL in DUT Hierarchy
 - ☐ HW top module should be a SystemVerilog module. It cannot be a VHDL module.
 - ☐ SW hierarchy above HW top should be Verilog only. It cannot have any VHDL.
- XMRs to VHDL (read/force) can be either direct XMR or specified through `$hdl_xmr` or `$hdl_xmr_force` system calls. However, XMRs to SystemVerilog under VHDL can be specified using direct XMR.
- Write XMRs to VHDL (writing a HW signal from SW) is not supported.
- XMR to complex type, such as records and arrays, is not supported.
- `$hdl_xmr/$hdl_xmr_force` should be accompanied by `$hw_read` and `$hw_force`. This is because `$hdl_xmr/$hdl_xmr_force` can have strings whose value is available only at runtime. However, if the `$hdl_xmr/$hdl_xmr_force` have constant string literals, then use the following compile-time option to let the tool parse the string literal:

```
-simxl=parse_hdl_xmr
```

This avoids the need for `$hw_read` and `$hw_force`.

This functionality might not work for some string literals. For example, it does not work if the string has a `/` instead of a `.` as an XMR separator.

- VHDL architecture cannot be specified as a module that can be moved from HW to SE. This is ignored with a warning.
- VHDL cannot be specified as an embedded testbench.
- Incremental compile with VHDL is not supported. However, to force incremental compile, you can specify `-simxl=incr_comp_mx`. This assumes that neither DUT nor communication has changed. This switch has no effect in the first (regular) compile. However, the switch causes the following in subsequent (incremental) compiles:
 - ☐ Simulation Acceleration assumes that DUT has not changed.
 - ☐ Simulation Acceleration assumes that communication has not changed.
 - ☐ Simulation Acceleration does not perform any checks to ascertain the above. You must ensure that it is safe to generate `simv` from TB hierarchy.
 - ☐ Simulation Acceleration does not invoke UFE/Simon or ZeBu backend. `simv` is generated from TB hierarchy.
- UCLI write of VHDL DUT signal from TB is not supported.
- Task Function Calls:
 - ☐ Task Function calls from VLOG TB to VLOG DUT (under VHDL)
 - ☐ Task Function calls from VLOG TB to VHDL DUT
 - ☐ Task Function calls from VHDL DUT to VLOG TB
 - ☐ Task Function calls from VLOG DUT to VLOG TB

1.5.5 Moving Glue Logic to Improve Performance

Glue logic is code in the testbench that connects multiple DUT ports. To increase performance, you might want to run the glue logic in the hardware, instead of the testbench. For example, suppose there is a clock-generator module, which is also marked as DUT. The clock output of the module comes to the testbench and then the clock is inverted and fed to the actual DUT module. In such a case, for performance enhancement, you can move the inverter to the hardware.

Movement of glue logic is always from the testbench to the hardware.

For more information, see the following subsections:

- [Types of Glue logic Supported](#)
- [Moving Glue Logic](#)

Types of Glue logic Supported

Simulation Acceleration supports the cont-assign glue logic. Both cont-assign of simple nodes and cont-assign of XMRs are supported. However, support is provided only for one-level XMRs (target XMRs should be in the direct child of the module where cont-assign is located). XMRs, such as 'TB.clkgen.mid.bot.a', are not supported.

Error messages are reported if you attempt to move the following types of glue logic:

- Primitive gates, such as AND
- Always/Always_comb (sometimes Always_ff/Always_latch)
- User-defined primitives (UDP)

Moving Glue Logic

To move glue logic, perform the following:

1. In the testbench, identify the construct that you want to move.
2. Update the testbench code with the following in-code pragma:

```
(* simxl_move_to_dut *)
```

In the following code snippet, the cont-assign `assign clk_n = ~clk` is identified for movement and consequently simulated in the hardware.

```
// SW Modules
module TB;
    wire clk, clk_n;
```

```
    (* simxl_move_to_dut *) // pragma for moving
    assign clk_n = ~clk; // logic to be moved
```

```

    clkgen clkgen(clk);
    dut dut(clkn);
    initial begin
        #10 $finish();
    end
endmodule

```

3. Specify the following compile option to see status of the movement:

```
-simxl=diag
```

SIMXL-DIAG messages and other diagnostic information is printed on the screen. See the SIMXL-DIAG message to see if the movement is successful. A sample message is shown as follows:

SIMXL-DIAG: Cont-assign ' (* simxl_move_to_dut = 1 *) assign clkn = (~clk); ' has been inserted into waitlist.

SIMXL-DIAG: ' (* simxl_moved_to_dut = 1 *) assign clkn = (~clk); ' has been moved to tb shadow.

4. Open the translog report located in the current working directory to view the improved performance after moving glue logic. You can generate the translog report by using the following runtime option:

```
-simxl=translog
```

For information on translog options, see [Debug Options for Communication Mechanisms](#).

1.5.6 Support for Final Block in Hardware

Simulation Acceleration implements a final block by moving the block from the hardware side to the software side. The following are supported in the final block:

- Access to SW nets and variables
- Call to SW DPI or function/tasks
- Access to HW registers

You can enable this support by setting the following compile option:

```
-simxl=move2tb_final
```


Limitation

Call to HW tasks or functions is not supported.

1.5.7 Support for Concurrent Assertion in Hardware

Simulation Acceleration provides task/\$display task support in HW hierarchy for Concurrent Assertion failure. The following cases are supported:

- Combination of XMR to task without parameters and display task
- Combination of XMR to task with constant and non-constant parameters and display task
- Combination of a mixture of statements of display and XMR tasks

For more information, see the following subsections:

- [Enabling Concurrent Assertion in Hardware](#)
- [Example](#)
- [Limitations](#)

Enabling Concurrent Assertion in Hardware

To enable task/\$display in HW modules (trigger execution of the task/\$display on TB at compile time), set the following compile-time option:

```
-simxl=assert
```

To enable task/\$display in HW modules (trigger execution of the task/\$display on TB at runtime), set the following runtime option:

```
-simxl=enable_dut_sva
```

Example

This example shows Concurrent Assertion in SystemVerilog.

```
# Concurrent Assertion
```

```
module dut;
```

```
.....
```

```
    task mytask;
```

```
        $display("Hello World!!");
```

```

    endtask
...
always @(negedge clk)
    B1: begin
        S1:assert property (p1(select,gnt_1,gnt_2))
            $display("Assertion P1 Passed");
    else begin
        mytask();
    end

        S2:assume property (p1(select,gnt_1,gnt_2))
            $display("Assume P1 Passed");
    else
        mytask();
    end
.....
endmodule

```

Limitations

For the following limitation, the Simulation Acceleration tool reports error messages:

- Assertion fail cannot have statement other than task calls or \$display system tasks:

1.5.8 Moving dumpvars/dumpports to the DUT Automatically

In the ZeBu flow, signals that need to be written out are usually specified in a separate file that contains the \$dumpvars or \$dumpports specifications. However, in the Simulation Acceleration flow, the module can be marked as a TB module. Such

Other Supported Features in Simulation Acceleration

modules are ignored by ZeBu, if you have not marked the module as a DUT top in the UTF file. To avoid this overhead, the Simulation Acceleration flow supports specifying the `$dumpvars` and `$dumpports` system tasks in the testbench.

The Simulation Acceleration tool identifies `$dumpvars/$dumpports` system task statements in an initial block of a TB module with a target of DUT hierarchy. These `$dumpvars/$dumpports` system task calls are moved to the DUT. The following cases are considered:

- If the `$dumpvars/$dumpports` is in a standalone module, the partition of the module is changed from TB to DUT. Therefore, the whole module is moved to the DUT side.
- If the `$dumpvars/$dumpports` is in a dedicated initial block, but the parent module has other constructs, the Simulation Acceleration tool moves the dedicated initial block with `$dumpvars/$dumpports` to a newly created DUT module, which is then used to save signals. Otherwise, if the `$dumpvars/$dumpports` is not in a dedicated initial block, the movement does not occur.

The movement is performed only in the following cases:

- When `$dumpvars/$dumpports` system task statements are in an initial block of a TB module with a target of DUT hierarchy
- When no Simulation Acceleration Not Yet Implemented (NYI) messages are reported.

For more information, see the following subsections:

- [Example 1: Dedicated Initial Block](#)
- [Example 2: Non-Dedicated Initial Block](#)
- [Limitations](#)

Example 1: Dedicated Initial Block

This example shows a `$dumpvars` specification in a TB module. The specification is in a dedicated initial block.

```
module TB
...
not n1(out, in); // other constructs
initial in = 1'b1;    // other constructs
```

```
initial begin          // dedicated for dumping
    $dumpvars(0, DUT);
end
```

Here, the `$dumpvars` specification has the following arguments:

- 0: Specifies that all signals need to be written out
- DUT: Specifies the module name or instance name

Only the dedicated initial block is moved instead of the whole module to the DUT side. The Simulation Acceleration tool creates destination module for moving is a special module called `hwcosim_dumpvar_mod`.

Example 2: Non-Dedicated Initial Block

This example shows a `$dumpvars` specification in a TB module. The specification is in a non-dedicated initial block.

```
module TB
...
initial begin          // not a dedicated initial block
    a = 1'b0;
    $display(a);
    $dumpvars(0, DUT);
end
```

Limitations

The following specification is not supported. If the `$dumpvars/$dumpports` refers both TB and DUT instances, the `$dumpvars` are not moved because the movement might invalidate the writing of for TB instances. The Simulation Acceleration tool reports an error message.

```
module TB
...
not n1(out, in); // other constructs
initial in = 1'b1; // other constructs
```

Other Supported Features in Simulation Acceleration

```
initial begin          // dedicated for dumping
    $dumpvars(0, sub.a.b, tb.a);
end
```

Here, `$dumpvars` has the following arguments:

- 0: Specifies that all signals need to be written out
- sub.a.b: Hierarchy under the hardware
- tb.a: Hierarchy under the software

In this case, only the signals on TB side are written out.

1.5.9 Miscellaneous Features

- Packages shared by HW and SW to enable common data types
- Packages used only by HW with supported HW constructs
- HW subroutines in an interface can call other HW subroutines in a different interface
- There is no limit on the nested calls of subroutines between HW and SW. For example, SW calls a HW subroutine, that calls a SW class method, and so on
- VHDL DUT wrapped in a SystemVerilog module is supported in HWOther Simulation Acceleration Limitations and Temporary Known Limitations

In Simulation Acceleration, the access limitations that do not include signals or subroutines are as follows:

- [Other Limitations That Do Not Include Signals or Subroutines](#)
- [Temporary Known Simulation Acceleration Limitations](#)

Other Limitations That Do Not Include Signals or Subroutines

See the following Simulation Acceleration limitations:

- **Accessing SystemVerilog events between SW and HW**

```
interface ift;
    event ev;
    task task1(output o1);
```

```
begin
    ->ev;
end
endtask
function integer calculate();
    return(0);
endfunction
endinterface
module HW();
    ift m_if();
endmodule
module SW();
    logic a;
    initial
        begin
            @(HW.m_if.ev); // Illegal
            $display("A: %d, CALC: %d", a, HW.m_if.calculate());
        end
endmodule
```

■ Accessing SystemVerilog chandle variable between SW and HW

chandle is not supported in HW.

```
chandle variable_name;
```

■ SW disabling a named block in HW

```
interface ift;
    always @(posedge clk)
        begin: myblock
            ...
        end
end
```

Other Supported Features in Simulation Acceleration

```

endinterface
module HW();
    ift m_if();
endmodule
module SW();
    initial
        begin
            disable HW.m_if.myblock; // Illegal
        end
endmodule

```

■ Accessing SystemVerilog dynamic types between SW and HW

Dynamic types are not supported in HW.

```

interface ift;
    byte amem [int]; // Illegal
    task task1(output o1);
        begin
            o1 = 1;
        end
    endtask
    function integer calculate();
        return(0);
    endfunction
endinterface
module HW();
    ift m_if();
endmodule
module SW();
    logic a;

```

```

initial
begin
    a = HW.m_if.amem[0][0] // Illegal HW.m_if.task1(a);
    $display("A: %d, CALC: %d", a, HW.m_if.calculate());
end
endmodule

```

- **Accessing SW elements from HW that reside in a SystemVerilog package** (including the default \$unit package) except for data types that are support in HW and class handles used for calling import methods that reside in a class object.

- **VCS system task \$set_toggle_region is not supported**

// None of the VCS system tasks below are supported.

```

initial begin
    $read_lib_saif("inputFile");
    $set_gate_level_monitoring("on");
    $set_toggle_region(Scope);
    $toggle_start;
    $toggle_stop;
    $toggle_report("outputFile", timeUnit, Scope);
end

```

- **No support for \$dumps in SW referencing HW hierarchies**
- **No support for same interface in both HW and SW**, where HW instantiation is being referred by virtual interface.

```

interface ift;
    byte amem [int]; // Illegal
    task task1(output o1);
        begin
            o1 = 1;
        end
    endtask
endtask

```


Other Supported Features in Simulation Acceleration

```

function integer calculate();
    return(0);
endfunction
endinterface
module HW();
    ift m_if();
endmodule
module SW();
    logic a;
    virtual interface vif;
    ift m_if_sw(); // Illegal
    initial
    begin
        vif = HW.m_if;
        a = HW.m_if.amem[0][0] // Illegal HW.m_if.task1(a);
        $display("A: %d, CALC: %d", a, HW.m_if.calculate());
    end
endmodule

```

- **\$stop is not supported**
- **Usage of Clock Delay Ports in Simulation Acceleration flow is not recommended.** Clock Delay Ports timescale by default is 1 ps/1 ps and cannot be changed. This does not work well with other timescales.
- **tran, tranif, or rtranif in SW connected to the DUT boundary port, which acts as a true bidirect:** Simulation Acceleration does not support tran, tranif, or rtranif in SW connected to the DUT boundary port, which has drivers in both SW and HW (true bidirect).
- Simulation Acceleration ignores a signal that is initialized to 1 in an initial block at zero time in a transactor.

Temporary Known Simulation Acceleration Limitations

Following are known limitations with the current release of Simulation Acceleration:

- To enable debug for Simulation Acceleration, use the following UTF command:

```
debug -verdi_db true
```

Do not use the `debug -all true` UTF command to enable debug for Simulation Acceleration.

- In testbench, if there are some events happening at the same time as `$finish` in a different thread, there could be difference in the order of execution of the threads between simulation and acceleration. In this case, a small delay to ensure `$finish` executes at the end.

2 Supported Syntax in Simulation Acceleration

For more information, see the following topics:

- [*Supported Syntax for Using Signals as the Communication Mechanism*](#)
- [*Signal Communication Using Primitive Data Types*](#)
- [*Supported Syntax for Using Subroutines as the Communication Mechanism*](#)
- [*Supported and Unsupported Behavioral Compiler Syntax*](#)

2.1 Supported Syntax for Using Signals as the Communication Mechanism

In Simulation Acceleration, the elements supported for signal-level communication between HW and SW are described in the following subsections:

- [*Signal Communication Using Primitive Data Types*](#)
- [*Signal Communication Using Aggregate Data Types*](#)
- [*Signal Communication Using Interface Ports*](#)
- [*Signal Communication Using XMRs*](#)
- [*Optimizing Performance for Clock XMRs From SW to HW \(clkopt\)*](#)
- [*Optimizing Performance for XMRs With Force and Release Statements*](#)
- [*SW Accessing HW Memory Using \\$readmem and \\$writemem System Tasks*](#)
- [*Accessing and Forcing HW Signals From Within SW Using \\$hdl_xmr and \\$hdl_xmr_force*](#)
- [*Signal Communication Through External Subroutines*](#)
- [*Strength Analysis in Simulation Acceleration*](#)
- [*Limitations With Signal-Based Communication*](#)

2.2 Signal Communication Using Primitive Data Types

Signal communication using the following primitive data types for signals:

- wire, wand, wor, tri, and so on.
- reg
- logic
- enum
- byte
- integer
- int
- shortint
- longint

Signed modifiers can be applied to any of these data types.

Example

```
module HW(input wire clock, input logic reset);
endmodule

interface ift;
logic clock;
assign HW.clock = clock;
endinterface

module SW;
    ift m_if();
endmodule
```

2.2.1 Signal Communication Using Aggregate Data Types

Signal communication using the following aggregate data types:

- Packed arrays

Signal Communication Using Primitive Data Types

- Unpacked arrays
- Multidimensional arrays
- Packed structs
- Unions

Any composition of these data types is supported in Simulation Acceleration.

Example

```
typedef struct packed{
    integer source;
    integer destination;
} descriptor;
module HW(input wire clock, input logic reset, input wire descriptor d[100]);
endmodule
interface ift;
descriptor m_d[100];
genvar i;
for (i=0; i<100; i++)
    assign HW.d[i] = m_d[i];
endinterface module SW;
    ift m_if();
endmodule
```

2.2.2 Signal Communication Using Interface Ports

Signal communication using interface ports:

- Interface ports containing signals with any of the types stated in the previous sections
- Modports

Example

```
interface ift;
endinterface
module dut(ift if_dut);
```

```
endmodule
module tb(ift if_tb);
endmodule
module top;
    ift m_if;
    dut dut(m_if);
    tb tb(m_if);
endmodule
```

2.2.3 Signal Communication Using Real Types

The Simulation Acceleration flow supports:

- Ports with `real` type at the boundary between the testbench and HW
- Predefined `nettype` with either of the following types: `current_r`, `voltage_r`, `wreal1driver`, `wreal4state`, `wrealavg`, `wrealmax`, `wrealmin`, `wrealsum`
- An Embedded testbench (ETB) module with `real` port or predefined `nettype` with `real` type

Use Model

To enable this support, specify the following vcs switch:

```
-simx1=enable_realport
```

Limitations

The following is not supported:

- 1) Inout port with `nettype`
- 2) Cross-boundary xmr to `real` type variable or `nettype`
- 3) Task/function/DPI calls between testbench and HW, which have the following argument type:
 - `nettype` or array of `nettype`

- ❑ struct containing `real` or array of `real`
- 4) Cross-boundary force/release on the `real` type or `nettype` node

2.2.4 Signal Communication Using XMRs

XMR support is enabled by default. The following table shows the supported elements for signal communication using XMRs.

TABLE 1 Supported Elements for Signal Communication Using XMRs

Supported Elements	Example
Signal communication using XMRs with structural code XMRs between SW and HW using continuous assignment and port connections.	<pre>module dut(input a); endmodule module tb; assign dut.a = 1'b1; endmodule</pre>
Signal communication using XMRs with behavioral code XMRs from SW to HW for reading HW signal values of any type. Note that behavioral reads of data types mapped to ZeBu memories are also supported in Simulation Acceleration.	<pre>module dut(input a); reg b; endmodule module tb; reg c; initial c = dut.b; endmodule</pre>
Signal communication using XMRs with behavioral code XMRs from SW to HW for writing to HW signals of any data type. Note that behavioral writes of data types mapped to ZeBu memories are also supported in Simulation Acceleration.	<pre>module dut(input a); reg b; endmodule module tb; reg c; initial dut.b = c; endmodule</pre>

TABLE 1 Supported Elements for Signal Communication Using XMRs

Supported Elements	Example
Signal communication using XMRs with behavioral code XMRs from SW to HW for applying force or release to HW signals of any data types.	<pre>module dut(input a); reg b; endmodule module tb; reg c; initial force dut.b = c; endmodule</pre>
Signal communication using XMRs that require a named block as a segment in the hierarchical access.	<pre>module sub(); reg b; endmodule module dut(input a); genvar i; for (i=0; i<10; i++) begin : NESTED sub sub(); end endmodule module tb; reg c; initial dut.NESTED[1].sub.b = c; endmodule</pre>

2.2.5 Optimizing Performance for Clock XMRs From SW to HW (clkopt)

Clocks are high frequency signals that cause frequent synchronization between SW and HW when used in SW to reference HW clocks through XMRs, slowing down performance. When using HW clock references in a repeat statement, there is only a need to synchronize between SW and HW only when the specified number of clock iterations complete. To enable an optimization such that there are not frequent synchronizations, use the following VCS elaboration compile-time option:

```
-simx1=clkopt
```



```
// Example of repeat statement reference HW clock in testbench code
repeat (100) @(posedge vif.clock);
```

Support for iff Construct with clkopt

The event control to be optimized by clkopt can have the iff construct. In this case, clkopt moves the entire (clock and condition) conditional event control to HW.

The reference to clock and condition should be via a virtual interface where the same virtual interface is used in both the clock and condition.

```
@(posedge vif.clock iff vif.rst==1'b1);
```

Enhancements to clkopt Optimization

The clkopt optimization is enhanced to support:

- body of repeat statement and
- an if-else block, with one having posedge and other having negedge clock.

The following code snippet shows the new support:

```
task automatic wait_n_edges(int rep, bit pos_edge);
    repeat(rep) begin
        if (pos_edge) @(posedge tb.d1.pl_s_clk);
        else @(negedge tb.d1.pl_s_clk);
    end
endtask
```

2.2.6 Optimizing Performance for XMRs With Force and Release Statements

This optimization avoids expensive SW to HW accesses needed for force assignments by moving the execution of the force/release to HW.

For example, suppose the testbench has a `force` statement as shown in the following example:

```
force tb.dut.xmr.to.lhs = tb.dut.xmr.to.rhs;
```

Similarly, a `release` statement can exist in the testbench.

The optimization moves the execution of the assignment above to HW because both LHS and RHS are XMRs to the HW.

To enable this optimization, use the following VCS elaboration compile-time option:

```
-simx1=forceopt
```

2.2.7 SW Accessing HW Memory Using `$readmem` and `$writemem` System Tasks

SW accessing HW memory using `$readmem` and `$writemem` system tasks is supported in Simulation Acceleration. Moving a HW initial block containing `$readmemh` from HW to SW is supported. The `$readmemh` and `$writemem` system tasks are ignored if placed in HW.

All variations of these tasks are supported:

- `$writememb`, `$writememh`
- `$readmemb`, `$readmemh`

The start and finish address arguments are optional as specified in SystemVerilog LRM.

Example

```
interface ift;
    logic [0:255] mem [4096];
endinterface
module dut;
endmodule
module HW;
    ift m_if();
    dut dut();
endmodule
module SW;
```

```

initial
begin
    $readmemh("data.hex", HW.m_if.mem);
    $writememh("data_copy.hex", HW.m_if.mem);
end
endmodule

```

2.2.8 Accessing and Forcing HW Signals From Within SW Using \$hdl_xmr and \$hdl_xmr_force

The following system tasks are supported in Simulation Acceleration:

- *\$hdl_xmr: Access HW Signals From Within SW*
- *\$hdl_xmr_force: Force HW Signals From Within SW*

2.2.8.1 \$hdl_xmr: Access HW Signals From Within SW

The syntax of the `hdl_xmr` procedure is as follows:

```
$hdl_xmr("source_object", "destination_object", [verbosity]);
```

Where:

- `source_object`: Specifies a VHDL signal or Verilog register or net using a string type. You can specify an absolute path or a relative path to the object.
Note: Use an absolute path instead of a relative path, if the source node resides in the VHDL part of the code or if the hierarchical path has a VHDL layer.
- `destination_object`: Specifies a VHDL signal or a Verilog register using a string type. You can specify an absolute path or a relative path to the object.

2.2.8.2 \$hdl_xmr_force: Force HW Signals From Within SW

You can use `$hdl_xmr_force()` calls to force a value on an existing Verilog or VHDL node. It allows you to force signals, from and at any level of the design hierarchy from within the VHDL architecture or the Verilog module and from any scope from where the standard defined system functions can be called. The

`$hdl_xmr_force()` procedure works similar to the Verilog `force` command. A status is returned from this function, which can be used to control the flow of the code.

The syntax is as follows:

```
$hdl_xmr_force(destination_object, value);
```

Where:

- `destination_object`: Specifies the hierarchical path as a string. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog register/net.
- `value`: Specifies the value to which the `destination_object` is to be forced. The value should be specified within double quotes as a constant string type. The specified value must be appropriate for the type and be specified in valid syntax for any radix, as supported by the language.

2.2.9 Signal Communication Through External Subroutines

Signal communication through external subroutines, such as PLI, VPI, and UCLI, is supported in Simulation Acceleration.

Declare signals in software to ensure the type of access is known at compile time. Use the `$hw_read`, `$hw_write`, and `$hw_force` VCS built-in system tasks. The `$hw_read` system task call takes an additional optional string literal argument called "passive", which causes the signal to be communicated passively, instead actively.

By default, XMR communication is active. This indicates that the HW sends the value of the signal to the SW whenever the value changes. However, in passive communication, the HW sends the signal to the SW along with an existing active XMR communication or a sync from SW to HW. Setting XMR communication to passive is less resource intensive.

Passive communication can be done if the usage of the signal of the XMR is in procedural context. If the usage is in structural context and the XMR communication is set to passive, the Simulation Acceleration tool reports an error.

Simulation Acceleration does not report an error for the following uses of XMR, when passive is specified:

- `$display(dut.xmr);`

- `initial lhs=dut.xmr;`
- `always @(posedge clk) lhs=dut.xmr;`

Simulation Acceleration reports an error for the following uses of XMR, when passive is specified:

- `assign lhs=dut.xmr;`
- `dut dut(dut.xmr); //usage in port`

Example 1

```
initial begin
    $hw_read(top.dut.w);
    $hw_write(top.dut.u);
    $hw_force(top.dut.v);
end
```

Example 2

This example sets the XMR communication to passive.

```
$hw_read(dut.xmr, "passive");
```

2.2.10 Strength Analysis in Simulation Acceleration

Strength analysis is used to check the strength of a wire in the HW. Strength is propagated when signals are passed through port connections from the SW to HW.

You can check the strength propagation for boundary ports and XMRs. When using Simulation Acceleration, you can set the strength of a net in the SW and consequently, check the strength of the net in the HW using the following ways:

- **Cont-assign:** See the following example specification:

```
assign (weak1, weak0) net1 = a & b;
```

- **Primitive-gate:** See the following example specification:

```
bufif1 (pull1, pull0) net2 = c & d;
```

- **MOS-gates:** See the following example specification:

```
rnmos(net3, in, ctrl);
```

Note

Strength analysis for triregs is not supported.

Enabling Strength Analysis

Set the following compile time option to enable strength propagation for boundary ports and XMRs:

```
-simxl=enable_strength
```

Viewing Strength Results in HW

At runtime, you can check the strength passed from SW to the HW (DUT) by viewing details in translog. Use the following runtime option:

```
-simxl=translog
```

For information on translog options, see the [Debug Options for Communication Mechanisms](#).

The strength is visible in the translog report, as shown in the following snippet:

```
SimTime: 0
```

```
[X] [Write] top.sd.d._vcs_hwcosim__tb2dut_init_wait [0:0] 'd0
```

```
[X] [Write] top.sd.d._vcs_hwcosim__tb2dut_init_wait [0:0] 'd1
```

```
[P] [SwHw] top.sd.d.in1 [0:0] 'bx, strength1 = 6, strength2 = 3
```

```
[P] [SwHw] top.sd.d.in2 [0:0] 'bx, strength1 = 6, strength2 = 7
```

```
[P] [SwHw] top.sd.d.in2 [1:1] 'bx, strength1 = 5, strength2 = 6
```

```
[P] [SwHw] top.sd.d.in2 [2:2] 'bx, strength1 = 6, strength2 = 5
```

```
[P] [SwHw] top.sd.d.clk [0:0] 'd0
```

```
SimTime: 2
```

```
[P] [SwHw] top.sd.d.in2 [0:0] 'd0, strength1 = 7, strength2 = 7
```

```
[P] [SwHw] top.sd.d.in1 [0:0] 'd0, strength1 = 3, strength2 = 3
```

```
[P] [SwHw] top.sd.d.in2 [2:2] 'd0,  strength1 = 5,  strength2 = 5
[P] [SwHw] top.sd.d.in2 [1:1] 'd0,  strength1 = 6,  strength2 = 6
```

2.2.11 Limitations With Signal-Based Communication

The following elements are not supported in signal-level communication between HW and SW.

- Signal communication with XMR using wreal, real, and shortreal data types.

An example of code that is not supported is as follows:

```
module dut();
  real a;
endmodule
module tb;
  assign dut.a = 1.05;
endmodule
```

- Signal communication with port or XMR using time data type.

An example of code that is not supported is as follows:

```
module dut(input time a);
endmodule
module tb;
  assign dut.a = $time;
endmodule
```

- Signal communication with port or XMR using tagged unions.

An example of code that is not supported is as follows:

```
typedef union tagged {
  void Invalid;
  int Valid;
} VInt;
module dut(input time a);
```

```
VInt vi1, vi2;
endmodule module tb;
assign dut.v1 = tagged Valid (23+34);
endmodule
```

■ Signal communication with signals in an interface have the following limitations :

- ❑ Should not be a signal declared within clocking blocks
- ❑ The interface should not have subroutines that are called from HW
- ❑ The interface should not have signals driven simultaneously by both SW and HW
- ❑ The interface should not be passed as port argument using a XMR

An example of code that is not supported is as follows:

```
interface ift;
clocking cb @(posedge clock);
endclocking
task a;
endtask endinterface
module dut(ift if_dut);
endmodule
module tb(ift if_tb);
endmodule
module top;
    ift m_if;
    dut dut(m_if);
    tb tb(m_if);
endmodule
```

■ Signal communication with port or XMR using signal references containing segments using named blocks not from generate statements.

An example of code that is not supported is as follows:

```
module top_legal;
```



```

initial begin
    for (int i=0; i<3; i++) begin: illegal //automatic int loop3=0;
        for (int j=0; j<3; j++)
            begin
                loop3++;
                $display(loop3);
            end
        end
    end
end
endmodule
module tb;
    initial
        top_legal.illegal.loop3 = 0; // illegal
endmodule

```

- Unpacked structs are not supported for signal-based communication.

2.3 Supported Syntax for Using Subroutines as the Communication Mechanism

For more information on the supported syntax for using subroutines as the Simulation Acceleration communication mechanism, see the following subsections:

- [*Interface Task/Function \(Subroutines\) Calls From SW to HW*](#)
- [*Class Method Calls From HW Interfaces to SW*](#)
- [*DPI Subroutine Call From HW Interfaces*](#)
- [*Streaming Export Methods*](#)
- [*UVM Messaging Support*](#)
- [*Import Calling an Export*](#)
- [*\\$test\\$plusargs, \\$value\\$plusargs Support*](#)
- [*System Tasks*](#)

- [System Task Support for \\$display](#)
- [Enabling Concurrent Export Task/Function Calls](#)

2.3.1 Interface Task/Function (Subroutines) Calls From SW to HW

Interface Task/Function (subroutines) calls from SW to HW:

- Interface subroutines should only consist of syntax supported by the Behavioral Compiler (see [Supported and Unsupported Behavioral Compiler Syntax](#)).
- The same interface instance subroutine cannot be called concurrently from two different places, regardless of whether it is automatic or not (see option to enable calling of concurrent subroutines in [Enabling Concurrent Export Task/Function Calls](#)).
- Access from SW to HW interface subroutine is done either by using a physical hierarchical XMR to the interface or using a virtual interface as a prefix to access the subroutine in the interface.
- Formal arguments of subroutines can be of any of the data types supported for signal communication mentioned in [Supported Syntax for Using Signals as the Communication Mechanism](#).
- To get values back through SW calls to HW tasks, use the output and inout modifiers.
- Ref modifier is not supported.
- Time arguments are supported when `clock_delay` is used. See [Supported and Unsupported Behavioral Compiler Syntax](#).
- This kind of call is often referred to as an "export call".
- Any interface or module that has an export task defined in it is referred to as a transactor.

Example

```
interface ift;
    task task1(output o1);
        begin
```

```

        o1 = 1;
    end
endtask
function integer calculate();
    return(0);
endfunction
endinterface
module HW();
    ift m_if();
endmodule
module SW();
    logic a;
    initial
        begin
            HW.m_if.task1(a);
            $display("A: %d, CALC: %d", a, HW.m_if.calculate());
        end
endmodule

```

2.3.2 Class Method Calls From HW Interfaces to SW

Class method calls from HW interfaces to SW:

- A SystemVerilog Class object handle must be passed from SW to HW. HW uses the handle to call class methods implemented in SW.
- The same method can be called concurrently from multiple places.
- Formal arguments of methods can be of any of the data types supported for signal communication mentioned in [Supported Syntax for Using Signals as the Communication Mechanism](#).
- Ref modifier is not supported.

- Time arguments are supported when `clock_delay` is used. See [Supported and Unsupported Behavioral Compiler Syntax](#).
- This type of call is often referred to as an "import call".
- Any interface or module that has an import task call is referred to as a transactor.

Example

```
class C;
    function void observe(input integer i1);
        $display("I: %d", i1);
    endfunction
endclass

interface ift;
    bit configure_done = 0;
    C m_handle;
    function void configure(C handle);
        m_handle = handle;
        configure_done = 1;
    endfunction
    initial begin
        wait (configure_done == 1);
        m_handle.observe(4);
    end
endinterface

module HW();
    ift m_if();
endmodule

module SW();
    C c1 = new();
    initial
```

```
begin
    HW.m_if.configure(c1);
end
endmodule
```

2.3.3 DPI Subroutine Call From HW Interfaces

DPI subroutine call from HW interfaces:

- The same subroutine can be called concurrently from multiple places.
- Formal arguments of subroutines can be of any of the data types supported for signal communication mentioned in [Supported Syntax for Using Signals as the Communication Mechanism](#).
- Ref modifier is not supported.

2.3.4 Streaming Export Methods

To enable streaming between SW and HW calls, use the following compile-time option:

```
-simxl=streaming_sw2hw.
```

With this option, if a HW task/function has none of the following:

- Return value
- Output argument
- Usage of #-delays

The SW to HW calls of that task/function is made streaming. With streaming, the VCS scheduler does not block and wait for HW to complete execution of the task, resulting in faster runtimes.

2.3.5 UVM Messaging Support

The following UVM report macros and function calls are supported in the HW:

- `uvm_info
- `uvm_warning

- ``uvm_error`
- ``uvm_fatal`
- `uvm_pkg::uvm_report_enabled`
- `uvm_pkg::uvm_report_info`
- `uvm_pkg::uvm_report_warning`
- `uvm_pkg::uvm_report_error`
- `uvm_pkg::uvm_report_fatal`

The UVM package is deduced as a SW module. Therefore, the calls to the above functions are treated as HW to SW function calls.

To facilitate UVM messaging in HW, you can use the `$sformatf/$sformat` system tasks to compose string messages in HW.

Example

```
interface hwmonitor(input a, input b);

initial
  forever
    if (a != b)
      `uvm_error("hwmonitor", "Variable a miscompares to b");
    else
      `uvm_info("hwmonitor", "Variable a compares to b");
endinterface
```

All existing functionality of `uvm_message` is consistent with messages coming from SW or HW. This includes message enumeration and runtime verbosity control using the regular runtime option: `+UVM_VERBOSITY=<verbosity-level>`.

Compile time verbosity is enabled using the following VCS elaboration option:

```
-simx1=UVM_COMPILE_VERBOSITY+<verbosity-level>
```

Simulation Acceleration compiles out the UVM message calls whose verbosity level cannot be met.

If there is no verbosity level provided (for example, `-simx1=UVM_COMPILE_VERBOSITY`), Simulation Acceleration assumes the default

value of UVM_MEDIUM, which is the same as the runtime +UVM_VERBOSITY option.

Example

```
int verbosity = 400;
`uvm_info(id1, "MY MESSAGE1", UVM_LOW)
`uvm_info(id2, "MY MESSAGE2", UVM_MEDIUM)
`uvm_info(id3, "MY MESSAGE3", UVM_HIGH)
`uvm_info(id4, "MY MESSAGE4", verbosity)
`uvm_info(id5, "MY MESSAGE5", UVM_DEBUG)
```

In the above example, by default all the above UVM reporting macros are compiled into the design. If you provide `-simx1=UVM_COMPILE_VERBOSITY+UVM_MEDIUM`, Simulation Acceleration would compile out the following macros:

```
`uvm_info(id3, "MY MESSAGE3", UVM_HIGH)
`uvm_info(id5, "MY MESSAGE5", UVM_DEBUG)
```

It does not delete the macro ``uvm_info(id4, "MY MESSAGE4", verbosity)` because its verbosity argument is not a compile constant.

2.3.6 Import Calling an Export

Import calling an export (HW calling a SW method that calls a HW subroutine):

- VCS runtime does not allow an import function to call a time-consuming export task or function. VCS runtime allows an import task to call a time-consuming export task or function.
- When an import task or function calls an export, the import must be a task. Simulation Acceleration adds a wait statement in a wrapper function that activates the export task or function, which causes it to be time-consuming. Therefore, the import must be a task.

Example

```
function void dutTaskOne; // export wrapper on SW

input logic i_x;
input logic i_y;
```

```

output logic o_x;
output logic o_y;
automatic bit[1:0]
vcs_hwcosim_packedInputs;
begin
    _vcs_hwcosim_0_isHwDone = 1'b0;
    begin
        _vcs_hwcosim_packedInputs = { >> 1{i_x, i_y}}; // pack inputs
    end
    $$hwcosimRxData(_vcs_hwcosim_idbId[0], _vcs_hwcosim_packedInputs); //
    initiate call to HW; actual call happens in preLER region
    wait ((_vcs_hwcosim_0_isHwDone === 1'b1)); // wait until HW is done
    { >> 1 {o_x, o_y}} = _vcs_hwcosim_0_packedOutputs[33:32]; // unpack
    outputs
end
endfunction

```

2.3.7 \$test\$plusargs, \$value\$plusargs Support

`$test$plusargs` are used to check if a runtime option is being used on the `simv` command line. `$value$plusargs` is used to obtain the value of a runtime option being provided on the `simv` command line.

These system tasks are supported in SW, transactor HW modules and in regular HW modules.

To enable the support in both HW module types, add the following lines to the UTF file:

```

system_tasks -task {$test$plusargs} -enable
system_tasks -task {$value$plusargs} -enable

```

Example

```

integer i1;
always@(posedge clk) begin
    if ($test$plusargs("HELLO"))

```



```

        $display("Hello argument found.");
    if ($value$plusargs("TEST=%d", il))
        $display("value was %d", il);
end

```

Limitation

`$test$plusargs` or `$value$plusargs` are not supported at time 0. For example, the following scenario is not supported:

```

initial
begin
    if ($test$plusargs("myarg")) $display("hello");
end

```

2.3.8 System Tasks

The following system tasks are supported:

- `$display`, see the [System Task Support for `\$display`](#) section.
- `$fdisplay`, together with: `$fdisplayh`, `$fdisplayb`, `$fdisplayo`
- `$write`, together with: `$writeh`, `$writeb`, `$writeo`
- `$fwrite`, together with: `$fwriteh`, `$fwriteb`, `$fwriteo`
- `$fopen`
- `$fclose`
- `$random`, together with: `$urandom`, `$urandom_range`
- `$finish`
- `$hdl_xmr`, see the [\\$hdl_xmr: Access HW Signals From Within SW](#) section for usage.

Usage requires adding the system task that is used in UTF file.

Simulation Acceleration has its own way of implementing system tasks, which results in the module/interface using the system task to become a Transactor processed by BC.

To return to regular ZeBu support for system tasks and disable the Simulation Acceleration implementation, add the following option to VCS compile:

```
-simxl=streaming_display
```

Example

```
// UTF commands to enable system tasks
system_tasks -task "\$display" -enable
system_tasks -task "\$write" -enable
```

Note

The file I/O operations are better done on the SW side even though supported for Transactor Code.

2.3.9 System Task Support for \$display

The following system tasks are supported:

- \$display
- \$displayh
- \$displayb
- \$displayo

The Simulation Acceleration tool can optimize \$display by **moving the display task to the shadow hierarchy of the corresponding module using a HW trigger**. Display optimization takes precedence over 'system_tasks -task {\$display} -enable'. If a \$display cannot be optimized, it falls back to the system_tasks implementation.

For more information on display optimization, see the following subsections:

- [Enabling \\$display Optimization for All HW Modules](#)
- [Enabling \\$display Optimization for Specific HW Modules](#)
- [Enabling \\$display Optimization for Backdoor Communication](#)
- [Pragma Support](#)
- [UVM Report Support](#)
- [Limitations of Display Optimization](#)

Enabling \$display Optimization for All HW Modules

To enable \$display optimization for all HW modules, specify the following compile-time option:

```
-simxl=displayopt
```

Enabling \$display Optimization for Specific HW Modules

When display optimization is used, there is more overhead because of increased communication between HW and SW. This leads to a reduction in performance. To increase the performance, you can enable \$display optimization for specific HW modules.

To do this, specify the list of modules in a text file and pass the file using the following compile-time option:

```
-simxl=displayopt+modules=<text-file-containing-list-of-modules>
```

Where, the <text-file-containing-list-of-modules> contains a module name in each line of the text file. Hash character is a comment and can be specified at the beginning of the line. In addition, blank lines are ignored.

Example:

```
-simxl=displayopt+modules=modules.txt
```

Enabling \$display Optimization for Backdoor Communication

By default, Simulation Acceleration transactors are used to communicate the arguments of the optimized \$display from HW to SW. However, by using the following compile-time option, you can enable SW to use backdoor read mechanisms to read the values from the HW:

```
-simxl=displayopt+backdoor
```

This can lead to an improvement in performance.

Pragma Support

You can annotate individual \$display statements using pragma to provide fine control over \$display optimizations. Pragma optimizations override module-level \$display optimizations.

The following pragmas are supported:

TABLE 2 Pragma Support for \$display Optimizations

Pragma	Function
(* simxl_displayopt *) or (* simxl_displayopt = "xtor" *)	Enables displayopt for a specific \$display to use transactor for communication. Note: \$display in loops always use backdoor.
(* simxl_displayopt = "backdoor" *)	Enables displayopt for a specific \$display to use backdoor for communication.
(* simxl_displayopt = "none")	Disables displayopt for a specific \$display.

UVM Report Support

In addition to \$display, the display optimization supports the following UVM reporting tasks:

- uvm_report_info
- uvm_report_warning
- uvm_report_error
- uvm_report_fatal

If you are using UVM macros, such as `UVM_INFO, the macro calls the uvm_report task inside an if condition that checks for the uvm_report_enabled() function.

Example

```
always @(posedge clk)
begin
    if (uvm_report_enabled(UVM_HIGH))
        uvm_report_info("ID4",
            $sprintf("wait_for_idle_credits() start"),
```

```

        UVM_HIGH);
end

```

Limitations of Display Optimization

Display optimization does not support the following:

- **Argument type:** Class, Struct, Interface, enum, memory, function xmr
- **Format specifier type:** %p is not supported
- **Blocks:** The following are not supported:

- ❑ System task in combinational blocks, as follows:

```

always_comb
always @(*)
always @(in1 || in2...)

```

- ❑ System task in function which used in combinational statements, as follows:

```

assign x = func(a,b);
function byte func(int x, int y);
    func = x + y;
    $display("x = %d, y= %d", x, y);
endfunction

```

2.3.10 Enabling Concurrent Export Task/Function Calls

Currently Simulation Acceleration allows only one execution occurrence of a specific export TF (task/function) at any given time, during run-time. If multiple calls to the same TF are made simultaneously, it results in an error, and stops Simulation Acceleration execution. Use the following options to enable multiple simultaneous occurrences of the same task, by specifying the name of the task/function and number of needed simultaneous occurrences of the respective task/function:

Use Model

V2Z Mode (UTF Option):

```

simx1_allow_concurrent -module <module-name> -tf {<task-function-

```

```
list>} [-count <number-of-occurrences>]
```

Where,

- **<module-name>**: Specifies the module or interface where the task/function resides.
- **<task-function-list>**: Specifies a space separated list of tasks/functions in the module/interface above.
- **<number-of-occurrences>**: Specifies the maximum number of simultaneous occurrences required for the specific task

Examples

```
simxl_allow_concurrent -module subdut -tf {dutTaskOne}
```

```
simxl_allow_concurrent -module sub2 -tf {T1 F1} -count 2
```

V2VX Mode (VCS option):

```
-simxl=concurrent+<module name>+<tf name>[+count]
```

Examples (equivalent to above mentioned UTF commands)

```
-simxl=concurrent+subdut+dutTaskOne
```

```
-simxl=concurrent+sub2+T1+2
```

```
-simxl=concurrent+sub2+F1+2
```

Note:

- In both modes, count is optional, and defaults to a value of 3.
- In ZeBu mode, the list of TFs in one module, that have same concurrent count, can be specified in one command.
- If the TFs correspond to a different module, or have different concurrent counts, they have to be specified as a separate UTF command
- In VCS mode, each TF spec has to be specified as an individual suboption 'concurrent'. The values in the command are separated by a '+' and recognized by position.

2.4 Supported and Unsupported Behavioral Compiler Syntax

For information on the supported and unsupported Behavioral Compiler syntax, see the following subsections:

- [Supported Behavioral Compiler Syntax in Simulation Acceleration](#)
- [Unsupported Behavioral Compiler Syntax in Simulation Acceleration](#)

2.4.1 Supported Behavioral Compiler Syntax in Simulation Acceleration

The following behavioral compiler syntax is supported:

- [#delay Support](#)
- [Clocking Edge in for Loops](#)
- [Multiple Clocks or Edge Expressions in the Same Process](#)
- [Complex Edge Combination](#)
- [Combinations of Edge and Level Events](#)
- [Bounded Loops](#)
- [Unbounded Loops](#)
- [Wait Statements](#)
- [Named Events](#)
- [Behavioral Compiler Supports all SystemVerilog Data Types](#)
- [Clocking Blocks](#)
- [Fork/Join](#)
- [SystemVerilog Strings, \\$sprintf, %m are Supported](#)
- [If -zebu_string is not provided, Simulation Acceleration reports an error stating that the -zebu_string should be enabled.Specifying a Module That Should Not Be Processed by Behavioral Compiler](#)
- [Usage of Memory in Behavioral Code](#)

2.4.1.1 #delay Support

- Used to model clocks and resets with #delay.
- Use the `clock_delay UTF` command to specify the module #delay is used in. Note that the UTF command only applies to the module contents at one level and not to any of the instantiated modules.
- You can use this to model delays in behavioral code as well.
- You can use variable delays that can be reconfigured.
- There is no support for having `$readmem*` system calls embedded in initial statements that are used to implement clocking and reset behavior.
- Use the `clock_config -accuracy 16|24|32 UTF` command to change bit-width to enable finer accuracy of clock delay precision.

Example

```
module HW;
bit  aclk;
bit  aresetn;
initial begin
    aresetn = 1'b0;
    #5000;
    aresetn = 1'b1; end
initial begin
    aclk <= 1'b0;
    forever begin
        aclk <= 1'b1; #1000;
        aclk <= 1'b0; #1000;
    end
end
endmodule
```


2.4.1.2 Clocking Edge in for Loops

See the following example:

```
for (i=0;i<128;i=i+1) begin
    @(posedge clk);
    mem[i] = 0;
end
```

2.4.1.3 Multiple Clocks or Edge Expressions in the Same Process

See the following example:

```
always @(posedge clk) begin
    a = 0;
    @(negedge clk);
    a = 1;
    @(negedge reset);
    a = 2;
end
```

2.4.1.4 Complex Edge Combination

See the following example:

```
always @(posedge clk or negedge clk2)
```

2.4.1.5 Combinations of Edge and Level Events

See the following example:

```
always @(clk or reset or posedge sig)
```

2.4.1.6 Bounded Loops

See the following example:

```
initial
for (int i=0; i< 5; i++) begin
    c <= c + 1;
end
```

Bounded loops are unrolled if there are no synchronizations between HW and SW within the loop. For example, if there is a `$display` in the loop, it is not unrolled.

2.4.1.7 Unbounded Loops

See the following example:

```
initial
while (1) begin
    @(posedge clk1)
    c <= c + 1;
end
```

2.4.1.8 Wait Statements

To enable correct wait behavior, see the following example:

```
wait(clk);
```

2.4.1.9 Named Events

See the following example:

```
event my_event;
initial -> my_event;
always begin
```

Supported and Unsupported Behavioral Compiler Syntax

```
@(my_event);
a <= b;
end
```

2.4.1.10 Behavioral Compiler Supports all SystemVerilog Data Types

Behavioral Compiler supports all SystemVerilog data types specified above.

2.4.1.11 Clocking Blocks

The following limitations apply:

- Cannot access signals in interface's clocking block using XMRs from SW.
- To enable clocking blocks, add the following temporary VCS compile option:
-Xzemi3=clocking_block.

Example

```
clocking cb @(posedge aclk);
    default input #`SETUP_TIME output #`HOLD_TIME;
    input aresetn ;
    input awaddr ;
endclocking
initial begin
    @(cb);
    A <= cb.awaddr;
end
```

Note

Simulation Acceleration reports a Not Yet Implemented (NYI) error message when a clocking block is used for the interface port in the HW hierarchy.

2.4.1.12 Fork/Join

Supports the following variations:

- `fork/join`
- `fork/join_none`
- `fork/join_any`

The following limitations apply:

- `Disable` is not supported.
- Process class for thread introspection and control is not supported.
- `fork/join_none` requires significant FPGA area and should be used carefully.

To enable `fork/join`, add the following VCS compile option:

```
-Xzemi3=fork_join
```

Example

```
fork
    do_A();
    do_B();
join
```

2.4.1.13 SystemVerilog Strings, `$sformatf`, `%m` are Supported

The support of `$sformat` and `$sformatf` system task/function calls on HW need the communication of string variables across the HW/SW boundary.

- The system function `$sformatf` returns a string argument.
- The system task `$sformat` has an output port of type string.

Use Model:

The support of `$sformat` and `$sformatf` can be enabled with the combination of below two switches.

- `-simxl=<systasks or the corresponding entry in the UTF file.>`
- `-zebu_string=<MAX_STRING_LENGTH>:-The -zebu_string variable length is limited to a maximum of 128 characters by default.To increase the limit`

Supported and Unsupported Behavioral Compiler Syntax

of maximum characters allowed in string variables, `-zebu_string` can be modified to `-zebu_string=<desired_maximum_capacity>`. For example, `-zebu_string=256`, strings of character lengths 256 are needed to run the design

If `-zebu_string` is not provided, Simulation Acceleration reports an error stating that the `-zebu_string` should be enabled. Specifying a Module That Should Not Be Processed by Behavioral Compiler

Specifying a module not to be processed by Behavioral Compiler, which is instantiated in a transactor, is done using the attribute `(*ZebuIgnoreZemi3Processing*)` for that module.

Alternatively, you can use the following switch in the VCS command script:

```
-Xhwcossim=enable_zemi3_ignore vcs
```

This switch enables VCS to mark all modules under a transactor to skip zemi3 compiler if there is no task/function in the modules. This option yields better compilation results and lifts the limitation of using hardware top as a transactor in the DUT.

It is a recommended practice to do this for modules that contain code, which can be processed by the ZeBu synthesis tools.

In the example, both IIP_TOP and BOT2 are not processed by the Behavioral Compiler.

```
module xtor(input CLK1, input [31:0] dout1, input [31:0] dout2);
wire [31:0] din1;
wire [31:0] din2;
integer i ;
task waitn (input bit [31:0] cycle ) ;
begin
    for (i=0 ; i < cycle ; i= i+1 ) begin
        @(posedge CLK1) ;
    end
end
endtask
```

```
IIP_TOP iip_top(dout1, dout2, din1, din2);
SUB1 s1(dout1, din1, din2);

endmodule

(* ZebuIgnoreZemi3Processing *)
module IIP_TOP(input [31:0] dout1, input [31:0] dout2, output bit
[31:0] din1, output bit [31:0] din2);
assign din1 = dout1 + 100;
BOT2 b2(dout2, din2);
endmodule

module BOT2(input [31:0] dout2, output bit [31:0] din2);
assign din2 = dout2 + 1000;
endmodule
```

2.4.1.14 Usage of Memory in Behavioral Code

By default, memory is considered as a state element. Behavioral compile maintains a copy of a state element in a variable suffixed with '_0' for every state variable. The variable samples the original state variable every driver clock cycle. This mechanism, causes the memory to be bit blasted and it is not mapped to a ZeBu memory.

Use the following UTF command:

```
memories -zmem -instance {<xtor name>.<memory name>}
```

Where:

- **<xtor-name>**: Specifies the name of the transactor module
- **<memory-name>**: Specifies the name of the memory instance within the transactor

Memory inferencing is required to keep the size of the generated logic for Transactors

instantiated many times manageable. During VCS elaboration, messages are reported in the log file. These messages specify the memories inferred and memories not inferred. When memory is not inferred, it is important to review the messages in the VCS elaboration log file and correct the behavioral code accordingly, if memory inferencing is required.

Typically, the VCS elaboration log file messages are reported for signals which are above `memSizeThreshold`, but you might not be intending to infer a memory for them. If you want them to be a memory, the read-write access pattern of the signals should match a typical memory usage.

The following lists examples of messages reported in the VCS elaboration log file.

Message 1

```
Memory memA is skipped as could not extract write port for this
sequential vector signal.
```

Reason for Message

`memA` is one-dimensional packed array and is written using full signal assignment. In this case, it is recommended to make this vector into a two-dimensional array with a reasonable bus size.

Memory inferencing is also difficult if there are multiple assignments to different bits in the indexed array. Collect all bits in a temporary variable and then assign the indexed array location with a single assignment of the temporary variable.

Message 2

```
Memory memB is skipped as this sequential signal has neither
reset sequence nor write port
```

Reason for Message

This is a generic message for any type of memory candidate signal that has a write pattern from which neither write-ports nor reset condition can be successfully extracted. This occurs when it is not obvious how the condition for writing enables the location by the given address to get written.

Message 3

```
Memory memC is skipped as concat expression couldn't be separated to
extract port writes.
```

Reason for Message

This message appears if the next state of the sequential signal has a concatenation-based assignment and write ports could not be extracted from it. This occurs if the implementation does not use an assignment to an array with an index but uses a full array assignment using concatenation.

Message 4

Memory memD is skipped as it is a dangling mem.

Source Node for dangling: _ASN_66

Reason for Message

This message appears when write-port or reset is extracted correctly but this memory is read fully in an assignment. In general, if a memory can be accessed without going through read ports, it is marked as dangling memory.

The following is an example that causes a dangling memory:

```
assign out = memD;
```

The following is the expected read pattern:

```
assign out1 = memD[raddr];
```

Message 5

Memory memE is skipped as it is a dangling mem.

Source Node for dangling: Child instance inport

Reason for Message

The reason for this message to appear is similar to that of [Message 4](#). In this case, the memory is completely passed to input port of a child instance. Therefore, the whole memory is read completely.

2.4.2 Unsupported Behavioral Compiler Syntax in Simulation Acceleration

The following SystemVerilog constructs are not supported by the Behavioral Compiler:

- SystemVerilog semaphores are not supported
- SystemVerilog mailboxes are not supported

Supported and Unsupported Behavioral Compiler Syntax

- SystemVerilog classes are not supported. Note passing a class object as a formal argument is supported.
- SystemVerilog dynamic data types are not supported (for example, dynamic arrays, associative arrays, queues, and so on).
- Usage of UVM config_db to assign physical interfaces to virtual interfaces is not supported in HW.

3 Scheduling Semantics and Race Conditions

The Simulation Acceleration time-coupled mode guarantees that advancement of time on the SW side is consistent with advancement of time on the HW side. Event scheduling between HW and SW is essential to maintain consistency between a VCS built model run and a V2Z built model run. For more information on V2Z, see [V2Z Mode](#).

Scheduling semantics of code running on SW side is compliant with the IEEE Std 1800-2012 LRM. Scheduling semantics of code running on HW side is based on a cycle-based semantics that is followed by tools like Emulation and Formal Verification driven by design clocks for non-transactor (for example, DUT) code and driven by a ZeBu driver clock (that is, `uclock`) for transactor.

Implementation of scheduling semantics for the handshaking between HW and SW has assumptions made to get better performance and to avoid other implementation limitations. In this clause, we try to specify what assumptions have been made to enable the user to ask the right questions when observing behavior that is not expected due to race conditions.

In the following clauses, race conditions are discussed that occur during:

- Initialization time
- SW/HW communication using signals
- SW/HW communication using subroutines

For more information, see the following topics:

- [Initialization Between SW and HW](#)
- [Resolving Clock/Data Race Conditions Between SW and HW](#)
- [Signal Handshake Between SW and HW](#)
- [Subroutine Handshake Between SW and HW](#)

3.1 Initialization Between SW and HW

Initialization is done in different stages. The considerations are as follows:

- Ordering between SW and HW initializations
- Initialization of static variables with their declaration time initial values
- Initialization of variables in initial block at time zero

You should look at key variables at end of time zero using debug facilities and rationalize the assigned values. Comparison with the results achieved with a simulation only run is essential.

3.2 Signal Handshake Between SW and HW

The SW side completes a delta cycle following the LRM Scheduling diagram as depicted in clause 4 of the LRM. The HW side then executes based on synthesis semantics a few driver clocks until it is ready to update the SW side. The SW code that is sensitive to intermediate events of HW signals are not observed.

3.3 Resolving Clock/Data Race Conditions Between SW and HW

When clocks are implemented in HW and SW uses them in event control statements, it is possible to indicate to the SW side that clock events coming from HW should be ordered to occur first from a SW perspective to avoid race conditions.

There are two orthogonal ways to specify the ordering.

1. Runtime option:

```
-simxl=clock_file:<filename>
```

Where, <filename> is the file that contains a list of hierarchical paths to clocks residing in HW. The file should contain one path per line.

2. Compile-time option:

```
-simxl=tbclock
```

This identifies clocks that appear as HW XMR when used in @ or wait statements in SW.

These options can be used both in the [V2VX Mode](#) and the [V2Z Mode](#).

3.4 Subroutine Handshake Between SW and HW

For more information, see the following subsections:

- [Import Calls and Non-Blocking/Blocking Assignments](#)
- [Export Calls Scheduling](#)

Import Calls and Non-Blocking/Blocking Assignments

Calling a none time consuming subroutine from HW to SW assigns arguments to the subroutine like a blocking assignment. Then, control moves over to the SW side at the end of the delta cycle. If there is code executing after the subroutine call in the HW side, that code executes in a different delta cycle than it originally was expected to.

In the following example, there is a none time-consuming subroutine call (that is, observed), followed by a blocking assignment (that is, `data_out*`) from variables that have a non-blocking assignment (that is, `m_data_out*`) at the clock edge of the same delta cycle. In a regular VCS run, the variables that have been assigned in a non-blocking assignment have their previous values. When running with Simulation Acceleration they (that is, `data_out*`) retain their current values. The blocking assignments occur in a different delta cycle than the non-blocking assignments.

```
// HW interface
interface itf (input clk, rstn);
reg data_out_vald;
reg [0:7] data_out;
always @(posedge clk) begin
If (!rstn) begin
data_out <= 0;
data_out_valid <= 1'b0;
else begin
// NBA - Non-Blocking Assignment
data_out <= data_out + 1;
data_out_valid <= !data_out_valid;
```

```
end
task monitor();
  forever begin
    // Block assignment occurring in different delta cycle than NBA
    m_data_out_valid = data_out_valid;
    m_data_out = data_out;
    @(posedge clk);
    if (m_data_out_valid) begin
      // Calling none time consuming function in SW
      m_monitor.observed(m_data_out_valid, m_data_out);
    end
  end
endtask
```

Export Calls Scheduling

When calling a function/task from SW to HW (that is, export), Simulation Acceleration does not immediately call function/tasks on the HW side because it would be very expensive. Instead, the SW wrapper initiates the actual HW call that is scheduled in preLER region (when control is given to HW). HW executes the task and control returns to SW. VCS schedules events in various runtime queues. preLER is pre Last Event Routine and is part of the Prepostponed region as described in the LRM. Runtime will trigger a call to ZeBu during this region.

4 Incremental Compile

Incremental compilation is implemented in the Simulation Acceleration flow. The Simulation Acceleration tool creates an incremental compile database to track all the HW and SW dependencies.

With incremental compilation, Simulation Acceleration compilation detects if the incremental compile interface database in the current compilation is a subset of the database in a previous Simulation Acceleration compilation. If it is a subset, the Simulation Acceleration compilation skips the ZeBu compilation.

Incremental compile enables separating SW compile (for example, UVM Testbench) from HW compile (for example, DUT and Transactors). Therefore, changes to SW do not cause a recompilation of the HW, which can take significant compile time. SW only compile uses existing VCS fast incremental compile capabilities.

The incremental compile database is created with no additional **zCui** options and it is tracked there on.

For more information, see the following subsections:

- [Modes for Performing Incremental Compile with zCui](#)
- [Support for \\$hw_force ECO](#)
- [Key Messages in Simulation Acceleration Incremental Compile](#)

4.1 Modes for Performing Incremental Compile with zCui

In Simulation Acceleration, there are two modes for performing incremental compile with **zCui**:

- **First Mode:** This is a regular compile. This is the current conservative mode and likely results in a HW recompile.
- **Second Mode:** In this mode, either of the following are possible:
 - ❑ **User-aware incremental compile:** Generates a `simv` based on the current HW database.

- ❑ **Shared HW database:** Generates a `simv` pointing to a shared HW database. This enables multiple different `simv`'s to be created based on a single HW database.

The `zCui` switch is as follows:

```
zCui --simxlSharedDB <zcu.work directory> <additional zCui options>
```

Note

There is no difference between how the two last modes are used, the difference is in how the users arrange their testbench environment. There are two kinds of use models:

- 1) Users that have a single `zcu.work` database that they share between different test cases, knowing that the test cases do not have any HW dependencies and they only require SW compiles.*
- 2) Users that have a `zcu.work` for every test and they need the incremental compile feature for each of the tests separately.*

4.2 Support for \$hw_force ECO

Simulation Acceleration saves you from a complete HW recompilation when there is no change in HW and there is no change in communication or if communication between previous and current compile has changed such that current is subset of previous. To further reduce the need to perform a complete HW recompilation, support is added for situations where new force statements (`$hw_force`) were added in the testbench to force HW signals.

For more information, see the following subsections:

- [Prerequisites](#)
- [Use Model](#)
- [Limitations](#)

Prerequisites

Before using this feature, address the following prerequisites:

- You must have a successful first compilation.

- In the first compilation, you must specify the following UTF command to enable ECO support for the second compilation:

```
eco -reserve_fw 1024 -reserve_qiwc 0 -reserve_force 128
```

Use Model

The following use model describes the flow after you have added new XMRs to the testbench. These would include statements to force HW signals.

Perform the following steps:

1. Delete or rename the existing eco.work.
2. Run the second compilation with the following VCS options:

```
-simx1=no_dut_recompile,incr_comp_eco_force
```

Simulation Acceleration detects the newly added XMRs and provides guidance on using zECO support to compile the newly added XMRs.

3. To use zECO, run **zCui** with the **-d** option.

The hwcosim_eco.tcl file is saved in the current working directory and the ZeBu database is saved as eco.work. VCS automatically runs the zECO command.

4. Run simv with the following command:

```
simv +zebu.work=eco.work/zebu.work
```

Limitations

The following limitation exists:

- The **-reserve_force_assign zCui** option is not supported during the second compilation.

4.3 Key Messages in Simulation Acceleration Incremental Compile

The following sections describe the different scenario occurrences when using the two operation modes:

- *No Warnings Reported When Generating simv*
- *Warnings Reported While Generating simv*
- *Communication Changes Caused Warnings While Generating simv*
- *New simv Cannot be Generated*

No Warnings Reported When Generating simv

- Testbench source code might have changed.
- There is no change in DUT source code.
- There is no change in communication facilities.
- Testbench these changes are not reported.

Warnings Reported While Generating simv

- Testbench source code might have changed.
- DUT source code has changed.
- There is no change in communication facilities.
- VCS lists the DUT changes for you to review.
- The DUT change report indicates the modules that have been changed, added, or removed.
- The exact DUT constructs that have changed are not be reported.
- You can review the module change report to determine if the changes are important for the test.
- You can decide if this simv can work without HW recompile.
- Testbench changes are not reported.
- Review the warning messages related to changes in DUT.

Communication Changes Caused Warnings While Generating simv

- Similar to *Warnings Reported While Generating simv*, but there are changes in communication.
- The exact communication changes are reported.
- The changes in communication are subset of previous communication and therefore Simulation Acceleration can recreate the necessary Verilog for the missing communication based on previous communication database.
- Review warning messages related to changes in DUT and communication.

New simv Cannot be Generated

- Communication has changed such that the changes are new or superset of previous communication.
- The exact communication changes are reported.
- simv, if generated, does not work without HW recompile.
- Error is reported and simv is not generated.

Compile

5 SVA Assertion and Coverage Support

In Simulation Acceleration, to enable functional coverage and assertions at compile time, use the following UTF commands:

```
coverage -enable true // Enables covergroups specified in HW
assertion_synthesis -enable ALL // Enables assertions and coverage
properties specified in HW
```

For more information, see the following topics:

- [Enabling Functional Coverage and Assertions at Runtime](#)
- [Controlling SVAs at Runtime](#)

Enabling Functional Coverage and Assertions at Runtime

To activate functional coverage and assertions at runtime, add the following options to the `simv` command:

```
-simxl=enable_dut_fcov,enable_dut_sva
```

Coverage data is generated in the `simv.zebu.vdb` file.

No special options are required to enable functional coverage and assertions in SW.

To merge coverage data from HW with coverage data from SW, use the following URG command:

```
urg -dir simv.vdb simv.zebu.vdb
```

Controlling SVAs at Runtime

The SVAs in the testbench side are enabled or disabled using existing `vcs` SVA controls, while the SVAs in the HW side are controlled by using runtime `hwcosim` options. You can perform the following activities:

■ Enabling SVAs During Runtime

By default, the SVAs in the HW side are disabled at runtime in Simulation Acceleration, however you can enable it by using the following options:

```
-simxl=enable_dut_sva
```

Specific SVAs in the HW can be disabled by passing an Assert file along with the

`disable_dut_sva` option. In the Assert file, state the names of the assertions that you want to disable.

```
simxl=disable_dut_sva,assertfile:<filename>
```

■ Enable Postprocessing of SVA

You can enable postprocessing of the SVAs in the HW side using the following options:

```
-simxl=post_process_sva,dumpdir:<dir.zsva>
```

The results are outputted in the ZTDB file, which can be converted to text format using the `zSvaReport` utility.

■ Handling Fatal Assertions

Specify the following option to stop emulation and simulation when a fatal assertion (`$fatal/$error`) failure is detected on the SVA in the HW side:

```
-simxl=stop_on_svafailure
```

On stopping, a UCLI prompt is available for further debugging.

■ Change Report Severity Level

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

By default, messages are reported on the screen when there is failure for the following SVAs:

- ☐ SVAs without an action block
- ☐ SVAs with an action block containing a `$error` or `$fatal`

The minimum level of severity from which messages are displayed can be changed using:

```
-simxl=set_report_severity,level=<Severity_level>
```

Where, `<Severity_level>` can be one of the following:

- ☐ `ZEBU_SVA_Failed_Fatal` (highest severity)
- ☐ `ZEBU_SVA_Failed_Error`
- ☐ `ZEBU_SVA_Failed_Warning`
- ☐ `ZEBU_SVA_Failed_Display` (lowest severity)

Example

When the severity argument is set to `ZEBU_SVA_Failed_Warning`, assertions that have an action block with `$warning` and higher severities (`$error` and `$fatal`) are displayed. This applies only on SVAs in the HW side.

6 Generating the Profiler Report

As you begin to achieve runtime functionality with a Simulation Acceleration V2Z environment ([V2Z Mode](#)), it is important to start becoming performance and functional aware.

Functional awareness is handled by making sure test passes as expected and matches the golden data. Performance awareness is required to ensure that the runtime environment is on the right track for achieving the required performance goals.

Simulation Acceleration [Communication] Profiler is a tool to aid users in Simulation Acceleration performance analysis. It generates a Simulation Acceleration Profiler report at runtime that consists of the details of communication between HW and SW during simulation acceleration.

Simulation Acceleration Profiler report includes the number of communication events between HW and SW and the amount of data transferred due to these events. It also reports the time consumed in different types of events in the Simulation Acceleration run. Frequent and expensive communication between HW and SW can indicate performance bottlenecks.

By analyzing the profile report, you can leverage this information to make appropriate changes to the environment to achieve the intended performance goals. These changes might involve modifying code or moving the code blocks into the HW.

For more information, see the following topics:

- [Enabling Profiler](#)
- [Profiler Report Sections](#)
- [Example: Communication Profiler Report](#)

6.1 Enabling Profiler

To enable Profiler, use the Simulation Acceleration runtime options described in the following table.

TABLE 1 Runtime Options to Enable Profiler

Runtime option	Description
<code>-simxl=profile</code>	Enables Profiler. This command is required to write TS Interface information in the report.
<code>-simxl=profile,profile_log:myrep.txt</code>	Renames the report generated. By default, the report is generated in a file called <code>simxlProfile.txt</code> .
<code>-simxl=profile,nheavy:<n></code>	Specifies the number of expensive steps. By default, 10 expensive steps are printed. Max value of <n> is 100.

The following UTF command is also required at compile-time to write TS Interface information in the report:

```
ztopbuild -advanced_command {set_simxl_opt -enable_profile yes}
```

Note: Multiple options require only one specification of `-simxl=profile`. There are no compile-time options needed.

6.2 Profiler Report Sections

The report comprises the following sections:

- Header
 - ☐ Tool Version
 - ☐ Start/End time
 - ☐ Clock frequency and edge count
 - ☐ Computed emulation time
- Time consumption
 - ☐ Simulation Acceleration Elapsed time
 - ☐ Communication time (Data Preparation + Synchronization)
 - ☐ Testbench time
 - ☐ VCS wait time
- SW/HW Synchronization
 - ☐ Number of time steps requiring synchronization
 - ☐ Number of synchronization events
 - ☐ Number of time steps requiring time streaming
- Histogram, elapsed time distribution, and expensive time steps
- Data transferred and event counts
 - ☐ Each type of signal/memory
 - ☐ Each transactor
- Details of data transfer (for individual signals, memories, tasks and functions)

Data preparation time for most expensive times is broken down into following:

- PIO data packing
- TF data packing
- Signal Writeback time
- Signal Flushing time
- Memory Readback time

- Memory Writeback time
- Memory Flushing time

Details of the time are more clear in the translog report.

6.3 Example: Communication Profiler Report

In the following example report, the following applies:

- The numbers in tables above are representative and not necessarily related.
- By default, 10 expensive steps are printed. Use the following suboption to change the default setting:

```
nheavy:<n>" [max of 100]
```

FIGURE 1. Communication Profiler Report

```
SimXL Communication Profiler Report
=====

+-----+
| VCS Version      | O-2019.06-Alpha_Full64 (ENG)
| VCS Build Date   | Build Date = Aug 16 2018 07:22:59
+-----+
| SimXL Start Time | Wed Aug 29 11:37:23 2018
| SimXL End Time   | Wed Aug 29 11:37:23 2018
| Mode             | Zebu
| VCS Simulation Time | 200
+-----+
| ZEBU Version      | O-2019.06-Alpha
| Zebu work path     | [full path: zebu.work]
| ZRDB Id           | 55124962
| Driver Clock Frequency | 6250 kHz
| Driver Clock Cycles | 1124948
| Timestamp Clock Cycles | 20
+-----+

1. Summary:
=====

1.1 Elapsed time
-----

NOTE: %time is the percentage of Elapsed Time.
```

Example: Communication Profiler Report

The breakup of Communication Time indicates % of Communication Time/% of Elapsed Time.

NOTE: HW time is not reported here, as the primary purpose of this report is the identification and analysis of communication and testbench bottlenecks.

	Time Spent	%time
SimXL Elapsed Time	4.960 ms	100
Test Bench Time	1.166 ms	23.5
Communication Time	3.795 ms	76.5
CoEmulation Sync Time	0.000 ms	0/0
Synchronization	3.765 ms	99.2/75.9
PIO data packing	0.026 ms	0.7/0.5
TF data packing	0.004 ms	0.1/0.1
Signal Writeback	0.004 ms	0/0
Signal Cache Flush	0.004 ms	0/0
Memory Readback	0.3 ms	0/0
Memory Writeback	1.4 ms	0/0
Memory Cache Flush	2.06 ms	0/0

Zebu initialization time

ZEBU Load Time	4.139 s
ZEMI3 Init Time	0.130 s
ZEMI3 Close Time	0.005 s

1.2 Synchronization events

Number of simulation time steps requiring synchronization: 20
 Number of SW/HW synchronization events: 53
 Number of simulation time steps requiring time communication only: 0
 Number of simulation time steps in SW time streaming mode: 0
 Number of synchronization events in SW time streaming mode: 22
 Average time spent per synchronization event: 0.013 ms

Average number of time steps per second: 4.98

Number of simulation time steps in SW time streaming mode: 0
Number of synchronization events in SW time streaming mode: 22

1.3 Histogram:

Sync Calls	Time Step count
----- -----	
>8	14
6	8
3	21
1	1

1.4 Elapsed time distribution:

Time Range	Time step count
----- -----	
10.000 ms - 100.000 ms	2
1.000 ms - 10.000 ms	2
0.100 ms - 1.000 ms	30
0.010 ms - 0.100 ms	7

1.5. Most expensive Time steps:

[Sorted by time in V2Z, by no of calls (i/p + o/p + tf, sync) in V2VX]

Timestamp	Synchronization	Input	Output	Task/Function	Elapsed
(ns)	calls	changes	changes	calls	Time
----- ----- ----- ----- ----- -----					
110	3	1	2	1	0.023 ms
70	3	1	2	1	0.014 ms
190	3	1	1	1	0.013 ms
150	3	1	1	1	0.014 ms
30	3	1	1	1	0.022 ms
0	3	1	2	0	0.067 ms
130	3	1	1	0	0.015 ms
100	3	1	0	1	0.015 ms
90	3	1	1	0	0.014 ms
40	3	1	0	1	0.130 ms

1.6 Transactor Information:

Number of transactors: 9
Number of zemi3 xtor threads: 4

Example: Communication Profiler Report

2. Event summary by type:

=====

Signals:

Type	Total Changes	Total data
Input	21	52 b
Output	14	448 b

InOut Signals:

Type	Total Changes	Total data
Input	3	192 b
Output	11	352 b

XMRs:

Type	Total Changes	Total data
SW->HW	23	736 b
HW->SW	6	192 b

Procedural Writes:

Type	Total Changes	Total data
SW->HW	15	45 b

Forces:

Type	Total Events	Total data
Force	7	224 b
Release	1	32 b

Memory:

Type	Total Calls	Total data
Read	6	48 b
Write	4	24 b

Task/Function communication data:

[for user transactors, and XMR TF calls]

[Import is HW->SW call, Export is SW->HW call]

HW->SW	SW->HW	HW->SW	SW->HW	Instance Name
Calls	Calls	Total Data	Total Data	
5	2	128 b	160 b	tb.u_duv

3. Details of Data Transfer:

=====

3.1 Details of signal changes:

Total Changes	Width	ID	Signal name
-----	-----	----	-----

Inputs:

20	1 b	0	tb.u_duv.clk
1	32 b	1	tb.u_duv.din

Outputs:

11	32 b	3	tb.u_duv.counter
3	32 b	2	tb.u_duv.douts

InOut: SW->HW:

3	64 b	2	tb.u_duv.counter
---	------	---	------------------

InOut: HW->SW:

11	32 b	2	tb.u_duv.counter
----	------	---	------------------

XMR: SW->HW:

23	32 b	7	tb.u_duv.u_sub.wval
----	------	---	---------------------

XMR: HW->SW:

3	32 b	5	tb.u_duv.u_sub.tb.wdin
3	32 b	6	tb.u_duv.u_sub.val

Procedural Write: SW->HW:

5	3 b	1	tb.u_hwtop.u_duv.A_0
5	3 b	3	tb.u_hwtop.u_duv.A_1
5	3 b	5	tb.u_hwtop.u_duv.A_2

Forces:

Force Calls	Release calls	Width	ID	Signal name
-----	-----	-----	----	-----
7		1	32 b	4 tb.u_duv.u_sub.val

3.2 Details of Task/Function communication:

NOTE: For internal communication:

HW->SW calls have 32 bits added to Input size [For HW->SW ID communication]

SW->HW calls have 32 bits added to Output size [For HW->SW ID communication]

Class and Virtual interface method calls (indicated with '**') have 64 bits added to Input size [For class/interface handle]

Instance 1: tb.u_duv.u_sub

Total Calls	Direction	Input Size	Output Size	Task ID	Task/Function name
-----	-----	-----	-----	----	-----
5	HW->SW	32 b	0 b	1	tb.v_task
2	SW->HW	32 b	32 b	0	tb.u_duv.u_sub.setVin

Example: Communication Profiler Report

Instance 2:

Total Calls	Direction	Input Size	Output Size	Task ID	Task/Function name
-----	-----	-----	-----	-----	-----

3.3 Details of Memory data transfers:

Reads	Writes	Read size	Write size	ID	Memory name
-----	-----	-----	-----	----	-----
3	2	24 b	12 b	7	top.dut.mem
3	2	24 b	12 b	8	top.dut.mem1

4. Details of Data Transfer for SimXL infrastructure:

=====

NOTE: This section is intended for internal RnD usage.

4.1 Data Transfer Summary:

Type	Total calls	Total data	Average calls	Average data
-----	-----	-----	-----	-----
Input	20	5.000 Kb	1.000	256.000 b
Output	11	2.750 Kb	0.550	140.800 b
Force	7	1.750 Kb	0.350	89.600 b
SW->HW call	2	128 b	0.100	6.400 b
HW->SW call	5	160 b	0.250	8.000 b
Memory Read	0	0 b	0.000	0.000 b
Memory Write	0	0 b	0.000	0.000 b

4.2 Details of Signal data transfers:

XTOR ID	Task ID	Type	Size	Change count
-----	-----	-----	-----	-----
1	0	Input	256 b	22
1	0	Output	256 b	11
2	0	Input	256 b	21
2	0	Output	256 b	3
3	0	Force	256 b	7

4.3 Details of Time communication:

Function Name	Type	Size	Call count
-----	-----	-----	-----
cosimTime_0_emuDelta	HW->SW	64 b	1
cosimTime_0_simDelta	SW->HW	64 b	1

4.4 Details of Control communication:

Function Name	Type	Size	Call count
-----	-----	-----	-----

```
|-----|-----|-----|-----|
|  cosimCtrlIOSync  | HW->SW | 32 b |      21 |
|  cosimCtrlISync   | SW->HW | 32 b |      32 |
```

4.5 Details of Task/Function Mask communication:

XTOR Name	ID	Type	Size	Call count
cosimPIO_1_	1	POMask	32 b	10
cosimPIO_1_	1	PIMask	32 b	20
cosimTF_2_	2	POMask	32 b	7
cosimTF_2_	2	PIMask	32 b	7

5. TS interface information:

```
=====
```

```
tb.CLKCTRL.rtl_cclkrdy: VC#: 6251458, !Ready Cycle#: 85704063
tb.cosimConf_2_.zemi3_readyForClock: VC#: 3, !Ready Cycle#: 2
tb.cosimCtrl.zemi3_readyForClock: VC#: 1993107, !Ready Cycle#: 996553
tb.cosimFR_1_.zemi3_readyForClock: VC#: 2208021, !Ready Cycle#: 1196924
tb.cosimTime_0_.zemi3_readyForClock: VC#: 1986365, !Ready Cycle#: 993182
tb.u_duv.clk0: VC#: 1553044
tb.u_duv.clk1: VC#: 1911
tb.u_duv.clk10: VC#: 161441027
```

6. Additional Information:

```
=====
```

```
Number of simulation time steps in HW time streaming mode: 1
Number of synchronization events in HW time streaming mode: 76
===== End SimXL Communication Profiler Report =====
```

7 Debugging and Reports

For more information, see the following topics:

- [Debug Options for Communication Mechanisms](#)
- [Functional Debug in Simulation Acceleration](#)
- [Using simprofile with Simulation Acceleration](#)

7.1 Debug Options for Communication Mechanisms

By default, the logs show the packed value of all arguments in each task/function call. This is useful for run-to-run comparisons. If you want to see detailed information on the task function arguments, use the following compile-time option:

```
-simxl=enable_log_tf
```

To enable logging the communication transactions that occur between SW and HW, use the following runtime options.

TABLE 1 Runtime Options for Logging Communication Transactions Between SW and HW

Options	Description
<code>-simxl=translog,translog_start:10,translog_end:100</code>	Enable logging from 10 to 100 time units
<code>-simxl=translog,translog_file:hwlog.txt</code>	Enable logging and print messages in file
<code>-simxl=translog,translog_level:1</code>	Enable logging and set verbosity level

It is recommended to use these options before going to debug through waveform

signals. This provides a high-level view of the interactions going on between HW and SW.

To analyze a specific timestamp to get a hint as to what is happening on the SW side, perform the following:

1. Create a shell script with run permission, "mon_stack":

```
#!/bin/bash

if [ -z "$1" ]; then
    echo "Usage: $0 pid"
    exit
fi

pid=$1

echo "Starting spying on $pid ..."
while [ -e /proc/$pid/status ]; do
    echo
    /bin/date --rfc-3339=ns
    echo "====="
    pstack $pid
    usleep 100000
done
```

2. Add the UCLI command to the VCS UCLI script or run from the UCLI prompt:

```
ucli% exec mon_stack [pid] > pstack_dump.log &
```

Run to the timestamp (Using the `run -absolute <timestamp> UCLI` command) before the slow timestamp. Execute the above command and stop simv after the slow timestamp. The script "mon_stack" will stop after simv stopped.

3. Check `pstack_dump.log` to find repeated called API or any other hint from call stacks. Send, capture, or share with R&D for analysis.

7.2 Functional Debug in Simulation Acceleration

Simulation Acceleration debug attempts to mimic VCS UCLI-based debug methodology. The UCLI provides basic functionality:

- Outputting waveforms using Dynamic Probe/FWC/QIWC
- Ability to set breakpoints in testbench code.
- Ability to force or deposit signal in HW.
- Ability to set breakpoint on signal changes in HW.

This section describes the following Simulation Acceleration debug-related information:

- [Enabling UCLI Signal Access for ZeBu Compile](#)
- [Enabling UCLI Waveform Output for ZeBu Compile](#)
- [Supported UCLI commands in Simulation Acceleration](#)
- [Verdi Interactive Debug for Simulation Acceleration](#)
- [Waveform Outputting Using UCLI](#)

7.2.1 Enabling UCLI Signal Access for ZeBu Compile

UTF commands used to enable UCLI control of signals in DUT

■ **Force/Release**

```
zforce -rtlname <Zebu_mapped_signal>
```

This enables force or release from the command line.

■ **Deposit Signal**

```
zinject -rtlname < Zebu_mapped_signal>
```

This UTF command makes transformations to enable deposit signal values.

■ **Read signal values**

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

This UTF command allows reading signal values in UTF commands.

This applies to dynamic signals required when read in UCLI, such as usage of the UCLI get command.

You can use SystemVerilog commands to enable UCLI control of signals in DUT:

- Usage of `$hw_read` in SystemVerilog code is required in the SW side to specify the signal used in the UCLI. Note that these signals cross between HW and SW on every update. This is required for stop UCLI commands.
- Usage of `$hw_force` in SystemVerilog code is required in the SW side to force/deposit the HW signal in UCLI.

7.2.2 Enabling UCLI Waveform Output for ZeBu Compile

- Waveform outputting must be specified in HW Verilog module to be able to output waveforms using FWC/QWIC.
- It is recommended to create all the FWC/QWIC groups in a separate module.
- Make sure the following commands are specified in the UTF file:

```
debug -waveform_reconstruction true
debug -verdi_db true
```

- Use an initial block with a label to identify a group of signals with a name. This name is used in the dump commands as the "value set". The name of the signal group is the same name as the "value set".

```
module dumpvars ();

    initial begin: fwc_waves_grp1
        (*fwc*) $dumpvars(0, hw_top.inst.a);
    end

    initial begin: qiwc_waves_grp1
        (*qiwc*) $dumpvars(0, hw_top.inst.b);
    end

    initial begin: fwc_waves_grp2
        (*fwc*) $dumpvars(0, hw_top.inst.c);
    end
endmodule
```

```
end
initial begin: qiwc_waves_grp2
    (*qiwc*) $dumpvars(0, hw_top.inst.d);
end
endmodule
```

7.2.3 Supported UCLI commands in Simulation Acceleration

See the following subsections:

- [*get UCLI Command*](#)
- [*memory UCLI Command*](#)
- [*stop UCLI Command*](#)
- [*run UCLI Command*](#)
- [*force, release, show UCLI Commands*](#)
- [*restart UCLI Command*](#)
- [*System Information Commands*](#)
- [*dump Commands*](#)
- [*sniffer Command*](#)

7.2.3.1 get UCLI Command

Obtain the value of a signal or variable

Syntax

```
get <signal> [-radix string]
```

Get the value of a signal <signal> in the format specified by the -radix option.

Example

```
get u_duv.ctrl -radix b
```

```
get u_duv.ctrl -radix d
get u_duv.ctrl -radix h
get u_udv.mem0.m -radix d # also works on memory
get u_duv.mem0.m\[100\] -radix d
```

Limitation

To display signals on the HW side, you must specify them at compilation time.

Use Model

To display signals on the HW side, specify them at compilation time in one of following ways:

1. Use `$hw_read` system task call.
2. Use `UTF probe_signals` command.
3. Use `$display` or any other read access in SW.
4. XMR procedural access from the SW side.

7.2.3.2 memory UCLI Command

Load or write memory values from or to files, or initialize memory with a specified value.

Syntax

```
memory -read|-write -file <fname> [-radix <radix>] [-start  
start_address] [-end end_address]
```

Example

```
memory -read u_duv.mem0.m -file in_mem_bin.txt -radix b
memory -read u_duv.mem0.m -file in_mem_hex.txt -radix b
memory -read u_duv.mem0.m -file out_mem_hex.txt -radix hex
memory -write u_duv.mem0.m -file hello.txt
```

The following are the equivalents of the Verilog system task `$readmem`,


```
$writemem.  
memory -read -radix b -> $readmemb  
memory -write -radix h -> $writememh
```

Use Model

To be able to read or write HW memory, enable access to the memory.

1. Use `$hw_read` and/or `$hw_write` system task call.
2. XMR usage in TB in procedural context.

7.2.3.3 stop UCLI Command

Stop the current run when value of a HW signal value changes.

Syntax

```
stop -change <signal>
```

Example

```
stop -change top.dut.a
```

Use Model

To be able to use the `stop -change` command, specify HW signal at compilation time in one of following ways:

1. Use `$hw_read` system task call in initial block.
2. Use the `probe_signals` UTF command. See [Read signal values](#).

For `stop` command usage when Dynamic Trigger and Runtime Trigger, see the following subsections:

- [Simulation Acceleration Dynamic Trigger](#)
- [Simulation Acceleration Runtime Trigger](#)

Simulation Acceleration Dynamic Trigger

The following table shows dynamic trigger options that pertain to the `stop`

command.

TABLE 2 Dynamic Trigger Options for the stop Command

stop Command	Description
<code>stop -expression <expr> -name <name></code>	Enables the specified <name>, where. - <name> is the instantiation path - <expr> is the equation
<code>stop -command <proc> <name></code>	Enables execution of a <proc> Tcl procedure when the equation is true. Execute this command after the following command: <code>stop -expression <expr> -name <name></code>
<code>stop -enable <name></code>	Enables the specified stop condition <name>
<code>stop -disable <name></code>	Disables the specified stop condition. Here, <name> should be a notifier name or a trigger path.
<code>stop -is_dynamic <name></code>	Returns 1 if <name> refers to a dynamic stop, else 0 is returned.

Simulation Acceleration Runtime Trigger

Before using runtime triggers in Simulation Acceleration, make sure you have the following files:

- The .cel file that describes the FSM. The description is in the Complex Event Language (CEL).
- The Dumpvars file (FWC, QiWC) contains all the signals defined in the .cel file.

The runtime trigger options that pertain to the `stop` command are as follows:

- **List all configured stop conditions, including runtime trigger objects:**

Specify the `stop` command with no options. For example:

```
$ stop
1: -cel a -fid ZTDB0 -once -action d
2: -cel a -fid ZTDB0 -repeat -action d
```

- **Enable the runtime trigger:** Specify the following command:

```
stop -cel [-fid <fid>] [-action <TclProcName>] [-once] [-repeat <N>
| repeat_forever]
```

Where,

- ❑ **-action <TclProcName>**: Specify to execute a Tcl procedure when the stop condition is reached. The Tcl procedure must have the following format:

```
proc call {module RTLTimeStamp lastNotify} {
    [proc body]
}
```

Where,

- ♦ **module**: Represents the module name where the notification occurs.
- ♦ **RTLTimeStamp**: Represents the RTL timestamp when the notification occurs.
- ♦ **lastNotify**: If software notifier was enabled in repeat, the value for this parameter is **1** only for the callback for the last notification, otherwise the value is **0**.
- ❑ **-once**: Specify to run the CEL FSM once. This is the default behavior if -repeat or -repeat_forever is not specified.
- ❑ **-repeat <N>**: Specify to run the CEL FSM for a specific number of times <N>.
- ❑ **-repeat_forever**: Specify to continuously run the CEL FSM until you disable the runtime trigger.

- **Disable the runtime trigger**: Specify the following command:

```
stop -disable <moduleName>
```

To enable the runtime trigger, specify the following command:

```
stop -disable <moduleName>
```

Example: Using the Runtime Trigger in Simulation Acceleration

For FWC, you need to specify the `-add_value_set` option, as shown in the following snippet:

```
dump -file fwc.ztdb -type fwc
```

```

dump -add_value_set ZFWC_DFLT_GRP -fid ZTDB0
dump -disable -fid ZTDB0
stop -cel ../run_zebu/pattern.cel -fid ZTDB0 -once
dump -enable -fid ZTDB0
run 500ns

```

For QiWC, you do not need to specify the `-add_value_set` option.

An example of the callback procedure is as follows:

```

proc callback {module rtl_time is_last_notify} {
    echo "***** CALLBACK *****"
    echo "MODULE          : $module"
    echo "USER CYCLE        : $rtl_time"
    echo "LAST NOTIFY       : $is_last_notify"
    echo "*****\n"
}

```

```

dump -file fwc.ztdb -type fwc
dump -add_value_set ZFWC_DFLT_GRP -fid ZTDB0
dump -disable -fid ZTDB0
stop -cel ../run_zebu/pattern.cel -action callback -fid ZTDB0 -once
dump -enable -fid ZTDB0
run 500ns

```

To view the module name of the loaded CEL file, specify the following command:

```
stop -cel
```

7.2.3.4 run UCLI Command

To start a simulation run that continues up to a specified time or event.

Example

```
run -absolute 10
run -posedge clk
run -absolute 5ns
```

7.2.3.5 force, release, show UCLI Commands

Force or deposit a value on a signal or variable.

Use the `force -list` command to show UCLI forced signals.

Syntax

```
force <signal> <value> -deposit|-freeze
force -list
```

If the `-deposit` option is specified, value is deposited on the current cycle to the signal `<signal>`.

If the `-freeze` option is specified, the value is frozen until released.

The `release <signal>` command releases a previously frozen signal.

Example

```
run 50ns
force counter 32'd1 run 50ns
release
```

To see if a signal is freezable or depositable, use the following command:

```
show -freezable|-depositable <signal>
```

This command returns if the signal passed as parameter is forceable, injectable, or writable.

Example

```
ucli% force dut_top.master_bfm_inst.wdata 0
ucli% run 100
```

```
100100 ps
ucli% force -list
dut_top.master_bfm_inst.wdata
ucli% release dut_top.master_bfm_inst.wdata
ucli% force -list
No signal has been forced via UCLI.
```

Note:

- The signals that are forced with the `-deposit` option are not listed
- The signals that are already released by the `release` command are not listed
- It only lists the UCLK forced signals. The signals forced by Verilog system task are not listed

7.2.3.6 restart UCLI Command

To restart the simulation at time zero, use the following UCLI command:

```
restart
```

7.2.3.7 System Information Commands

The UCLI commands used to retrieve system information is as follows:

TABLE 3 UCLI Commands to Retrieve System Information

UCLI command	Description
<code>senv</code>	Display one or all <code>synopsys::senv</code> array elements.
<code>senv value_sets</code>	Returns all value sets existing in design compilation.
<code>senv driver_clk_frequency</code>	Returns driver clock frequency in kHz.
<code>senv zebu_work</code>	Returns <code>zebu.work</code> path.
<code>senv time</code>	Returns current emulation time.

7.2.3.8 dump Commands

See the following dump commands:

- `dump -file <FILE> -type fwc|dynamic_probe`

Creates a waveform database directory named <FILE> for both SW and HW. Returns a file ID (that is, FID) that can be captured in Tcl. The subsequent commands use the FID. The default FID is "ZTDB0" and can be used directly without capturing it in a Tcl variable. In the waveform database directory, an FSDB file named `simx1.fsdb` is used to capture all SW signals.

- `dump -add <list of hierarchical paths> -depth <positive number> -fid <FID>`

Add signals that will be outputted to the provided FID. Specify the instance and depth the signals will be taken from. Depth 0 provides all signals in the provided instance and below.

- `dump -add_value_set <value_set> -fid <FID>`

Add <value_set> to a given <FID>. Multiple value set can be added. Value sets are specified using `$dumpvars` commands together with block labels.

- `dump -enable/-disable -fid <FID>`

Enables and disables outputting before starting or continuing simulation.

- `dump -close -fid <FID>`

Closes the file associated with the specified FID.

- `dump -flush -fid <FID>`

Flushes the contents into the file associated to the specified FID.

- `dump -load_selection [<path>] -fid <fid>`

Used to specify a selection database. If no path is provided, the following path is used:

```
./zcui.work/zebu.work/zrdb/csa_supports.zrdb
```

The `csa_supports.zrdb` database file is created by either of the following commands:

```
debug -waveform_reconstruction true
```

or

```
debug -all true
```

7.2.3.9 sniffer Command

Sniffer is the technology that records the ZeBu state and saves the stimuli. Sniffer records frames. Each frame is a window of Stimuli with the ZeBu State at the beginning of the stimuli window. To use the Sniffer technology, specify the `sniffer` UCLI command.

For more information, see the following subsections:

- [Prerequisites](#)
- [Supported sniffer Options for Simulation Acceleration](#)
- [Setting the Runtime Generated Data Directory](#)
- [Examples](#)

Prerequisites

Before using the `sniffer` UCLI command, specify the following UTF command:

- New engine for Stimuli Capture and Replay (Recommended):

```
debug -offline_debug true -offline_debug_params {INCL_XTORS=t}
```
- Legacy Stimuli Capture and Replay flow:

```
debug -offline_debug true
```

Supported sniffer Options for Simulation Acceleration

The following options are supported for the `sniffer` UCLI command:

- `sniffer -start_frame`
- `sniffer -at_speed`
- `sniffer -config save_state_at_stop`
- `sniffer -config keep_frames <n>`
- `sniffer -config output_record`
- `sniffer -config no_memory_copy`
- `sniffer -auto_create <minutes>|<seconds>s|<minutes>m|<hours>h [-prefix <prefix>]`
- `sniffer -list [-dbs | <db_name>]`
- `sniffer -info [<name>]`
- `sniffer -status`

For more information on the sniffer UCLI command and options, see the ***ZeBu UCLI User Guide***.

Note

Each sniffer UCLI command has a corresponding Tcl command with the same name and usage.

Setting the Runtime Generated Data Directory

By default, runtime generated data is stored in the directory called `emulation_data_<DATE>_<TIME>`, which is in the current working directory. To set a different location for the runtime generated data directory, use the following command:

```
simv -ucli -simx1=db_path:<path>
```

Where, `<path>` is the new directory name in the current working directory. For example, in the following command, `db2` is the new directory name:

```
simv -ucli -simx1=db_path:db2
```

Examples

In the following examples, the final label names are `sn_0`, `sn_10`, `sn_20`, and so on.

Example 1

```
set idx -1
sniffer -create sn_[incr idx]
stop -relative 10ns -continue -repeat -command {
    sniffer -create sn_[incr idx]
}
```

Example 2

```
sniffer -create sn_$now
stop -relative 10ns -continue -repeat -command {
    sniffer -create sn_$now
}
```

7.2.4 Verdi Interactive Debug for Simulation Acceleration

Verdi interactive debug supports the Simulation Acceleration flow. With Simulation Acceleration interactive debug, you can debug both software (testbench) and hardware (DUT) in the Verdi interactive debug environment. Simulation Acceleration interactive debug also allows you to output testbench signals and DUT signals into one ZTDB file.

For more information, see the ***ZeBu Verdi Integration Guide***.

7.2.5 Waveform Outputting Using UCLI

- Simulation Acceleration UCLI uses dump commands described in the previous section to control the outputting.
- The key differences between VCS and ZeBu output is as follows:
 - ❑ VCS does not have any special groupings to output waveforms.
 - ❑ VCS only requires read access enabled for outputting.
 - ◆ All the hierarchies and signals are available for outputting.
 - ❑ ZeBu has the following separate modes for outputting:
 - ◆ **Dynamic Probe**: This mode can be used to output any hierarchy without compile directive, but is very slow.
 - ◆ **FWC**: This mode is fastest but requires significant hardware resources.
 - ◆ **QiWC**: This requires fewer HW resources and has intermediate performance.
 - ❑ ZeBu outputs waveforms in a compact format called ZTDB:
 - ◆ ZTDB format must be converted into the ZWD format to be enabled in Verdi.
 - ◆ ZTDB only outputs register values and "combinatorial signals" are calculated on-the-fly by Verdi.
 - ◆ Unlike pure VCS, the waveform outputting should be properly planned before compiling the design.

For more information, see the following subsections:

- [Dynamic-Probe](#)
- [FWC Probe](#)
- [QiWC Probe](#)
- [Multi-ZTDB Output](#)
- [Using Verdi for Debug](#)
- [Output ZeBu XTOR Internal Signals](#)
- [Waveform Expansion in Simulation Acceleration](#)
- [Waveform Outputting config slice for zSimzilla](#)

7.2.5.1 Dynamic-Probe

Advantages

- ☐ No compile directives required
- ☐ Full scope of design is available for dumping

Disadvantages

- Very slow outputting. Probably suitable for only few cycles for a large design.
- Recommend to run within a very short window using absolute run commands (see below)
- UCLI Tcl outputting example:

```
run -absolute 34000ns
set v_fid [dump -file rwc_wave.ztodb -type dynamic_probe -driverClk];
dump -load_selection -fid $v_fid;
dump -enable -fid
$v_fid;
run -absolute 34010;
dump -fid $v_fid -flush;
dump -fid $v_fid -close;
```

7.2.5.2 FWC Probe

Advantages

- This is fastest way to output probes.

Disadvantages

- Must be enabled at compile time
- Uses extensive HW capacity and can only be enabled for selected "essential signals"

Example

```
set v_fid [dump -file fwc_wave.ztdb -type fwc]
dump -add_value_set fwc_waves_grp1 -fid $v_fid
dump -add_value_set fwc_waves_grp2 -fid $v_fid
dump -enable -fid $v_fid
run 10us
dump -fid $v_fid -flush
dump -fid $v_fid -close
```

7.2.5.3 QiWC Probe

In V2VX mode, you can do the same waveform outputting and debug as in regular VCS flow.

In V2Z mode, you can output the waveform in zRci GUI. For details, see the **ZeBu User Guide**. Simulation Acceleration V2Z mode also allows you to use ZeBu QiWC waveform capture through UCLI. The steps are as follows:

1. In the DUT, add following source code to enable ZeBu QiWC waveform output:

```
initial begin : GROUP
    (* qiwc *) $dumpvars(0, tb.dut);
end
```

2. Create a UCLI script to output the waveform.

```
//mydump_qiwc.tcl:
dump -file myqiwc.ztodb -type fwc
dump -add_value_set GROUP -fid ZTDB0
run 372685000
dump -close
```

3. Run the simulation with this UCLI script as shown in the following command:

```
% simv -ucli -i mydump_qiwc.tcl <other runtime options>
```

4. After simulation you can open and debug the outputted waveform with Verdi by using the following command:

```
% verdi -emulation --input <your dumped ztodb directory> --zebu.work
<your zebu.work directory>
```

7.2.5.4 Outputting Signals of HW Module Instances That Were Moved to SW

By default, module instances that were moved from HW to SW are not outputted. Outputting request needs to be specified explicitly using the following UCLI command:

```
dump -add {<module-instance> } -fid <File-Identifier-Name>
```

Example

```
dump -add {top top.dut.subdut.tb_inst1} -fid ZTDB0
```

Note

These signals are sampled into the `simxl.fsdb` file residing in the waveform database directory associated with the FID, named `ZTDB0`.

7.2.5.5 Multi-ZTDB Output

Simulation Acceleration supports multi-ZTDB output through UCLI. Several FWC and QiWC waveform files can be captured simultaneously as long as they do not share any Value-Set. Only one dynamic-probe waveform can be captured at a time.

Example

```
dump -file fwc.ztdb -type fwc
dump -file fwc2.ztdb -type dynamic_probe
dump -add / -depth 0 -fid ZTDB0
dump -load_selection -fid ZTDB1
dump -add_value_set ZFWC_DFLT_GRP -fid ZTDB0
run 10ns
dump -close ZTDB0
run 20ns
dump -close ZTDB1
```

7.2.5.6 Output FWC/QiWC Waveform Only on Time Advance

You can use the `sample_on_time_adv` option to reduce the time stamp or raw data file size in the ZTDB file as pound zero (#0) occurs in the design. This option enables sampling on time-advance while FWC or QiWC is outputted. Converting the reduced files saves waveform conversion time to increase convert performance.

Prerequisite

The Verilog RTL file must contain #0.

Procedure to Enable

1. Run `simv` with the `sample_on_time_adv` option, as shown in the following:

```
simv -ucli -simxl=sample_on_time_adv
```

2. Output FWC/QiWC using UCLI. See the [Waveform Outputting Using UCLI](#) section.

If the sample-on-time-advance run is successful, a `config_timestamp` that contains sample information in the ZTDB directory is created, as shown in the following:

```
SAMPLING_ONLY_ON_TIME_ADVANCE TRUE
```

Limitations

- Supported only for ZeBu Server 4
- The `-simxl=disable_alt_exec_flow` VCS command-line options is not supported

7.2.5.7 Using Verdi for Debug

- [Verdi Interactive Debug for Simulation Acceleration](#) is supported. See the **ZeBu Verdi Integration Guide** for more information.
- Use "-kdb -lca" VCS option to output Verdi KDB database like VCS.
- Open the Verdi database with the ZTDB file using the following command:

```
verdi -nologo -emulation --zebu.work <ZEBU_WORK> --input
<ZTDB_FILE> <other verdi command line options>
```

This opens Verdi to be run with the iCSA engine.

7.2.5.8 Output ZeBu XTOR Internal Signals

When using the subroutine-based Simulation Acceleration flow, you might want to output XTOR internal signals. This is useful, for example, when you want to check XTOR key FSM signals "imp_state_net_*" to debug an XTOR behavior. You can output these signals through the Simulation Acceleration `dynamic_probe` by performing the following steps:

1. Run `zSelectProbes` to select the signals to output.



2. Select **Save selection** from the **File** menu to save the selects.
3. Exit `zSelectProbes`.
4. Specify the following UCLI commands to output the signals:

```
set fid_dyn [dump -file mydyn.ztdb -type dynamic_probe -driverClk]
dump -load_selection -fid $fid_dyn
dump -enable -fid $fid_dyn
run 8135000
dump -flush -fid $fid_dyn
dump -close $fid_dyn
```

After running `zSelectProbes`, besides outputting the selected signals through UCLI command line, you can output the selected signals in `zRci`.

7.2.5.9 Waveform Expansion in Simulation Acceleration

After the ZTDB file is outputted, you can use the ZeBu Waveform Expansion utility, `zSimzilla`, to get the ZWD file for debugging with Verdi. For more information, see the ***ZeBu Debug Guide***.

Example

The following command writes FSDB files from **myqiwc.ztdb** to the **dut_fsdb** directory. You can then debug the design with the generated FSDB files.

```
%zSimzilla --work zebu.work --ztdb myqiwc.ztdb dut_fsdb --timescale
1ns
%verdi -ssf dut_fsdb.vf
```

For XTOR internal signals outputted through `dynamic-probe`, you can only convert ZTDB to FSDB. There is no waveform expansion available for these internal signals. The `zSimzilla` command is as follows:

```
%zSimzilla --capture-only --work zebu.work ...
```


7.2.5.10 Waveform Outputting config slice for zSimzilla

Compile-Time Prerequisite

To use zSimzilla, make sure the following command is specified in the UTF file:

```
debug -waveform_ reconstruction true
```

Outputting ZTDB Type QiWC or FWC

Specify the following command for outputting ZTDB of type QiWC and FWC:

```
config waveform_capture_slicing
(auto|<seconds>[s]|<size>MB|<samples>total_samples|<slices>slices|{<s
amples>total_samples,<slices>slices})
```

Outputting ZTDB Type dynamic_probe

Specify the following command for outputting ZTDB of type dynamic_probe:

```
config dynamic_probe_slicing
(auto|<seconds>[s]|<samples>total_samples|<slices>slices|{<samples>to
tal_samples,<slices>slices})
```

Example 1

This example shows how to enable config slice feature.

Specify the following to enter interactive mode:

```
../simv -ucli
```

In readback, specify the following:

```
config dynamic_probe_slicing auto
dump -file rb.ztdb -type dynamic_probe
dump -load_selection -fid ZTDB0
run
```

For FWC, specify the following:

```
config waveform_capture_slicing auto
dump -file fwc.ztdb -type fwc
```

```
dump -add_value_set {VAL} -fid ZTDB0  
run
```

Run zSimzilla to convert ZTDB to ZWD by entering the following commands:

```
zSimzilla --zebu-work ../zcui.work/zebu.work \  
    --ztdb fwc.ztdb \  
  
    --zwd hello.zwd  
nWave hello.fsdb.vf
```

Example 2

This example shows some syntactical use of `waveform_capture_slicing` and `dynamic_probe_slicing`:

```
config waveform_capture_slicing 3MB  
config waveform_capture_slicing 3s  
config dynamic_probe_slicing 3s  
config dynamic_probe_slicing 100total_samples  
config dynamic_probe_slicing 10slices  
config dynamic_probe_slicing 100total_samples,10slices
```

7.3 Using simprofile with Simulation Acceleration

The Unified Simulation Profiler (simprofile) is supported in Simulation Acceleration. For more information, see the following subsections:

- [Viewing Timeline Using simprofile](#)
- [Viewing Delay Events Using simprofile](#)

Viewing Timeline Using simprofile

With simprofile, it is possible to get a breakdown of the testbench code and see how much time is used by the Simulation Acceleration communication overhead. To enable, use the regular simprofile compile time and runtime options.

It is also possible to get a profile over time to see how the testbench is a bottleneck at different timepoints throughout the run. This feature is the simprofile timeline.

To enable, perform the following:

1. Specify the following compile option:

```
-simprofile
```

2. Specify the following runtime option:

```
-simprofile timeline
```

Viewing Delay Events Using simprofile

Using simprofile, you can report the number of delay events being generated during simulation. A delay event is usually generated by Verilog expressions, which contains delay controls. For example:

```
#1;  
a = #1 0;  
a <= #1 1;  
#1 -> e;
```

However, after VCS optimizations, delay controls might be removed or merged. Therefore, the number of delay events can be less than the delay controls executed. Instrumentation on each delay event can be significant and therefore the performance overhead caused by profiling overhead is large.

Note

Zero-delay (#0) event is not supported.

To enable, perform the following:

1. Specify the following compile option:

```
-simprofile
```

2. Specify the following runtime option:

```
-simprofile delay
```

To view the profiling report in a web browser, such as Firefox, specify the following:

```
firefox profileReport.html
```

To view the profiling report, which enables you to view the source line information, specify the following:

```
profrpt -firefox profileReport.html
```

The profiling report displays information in views. To view the source code, choose the **Construct** view. Choose the construct. For example, in the figure below, the highlighted link is clicked.

Counter Time Construct View		
Name	Count	Percentage
▼Initial	3703	60.01 %
NoName {line:7}	2468	39.99 %
NoName {line:2}	1234	20.00 %
NoName {line:17}	1	0.02 %
▼Always	2468	39.99 %
NoName {line:14}	2468	39.99 %
total	6171	100.00 %

FIGURE 1. Profiling Report: Counter Time Construct View

To view the source code, scroll down and click the **Source Information** link.

Construct Information	
Construct Name	NoName {line:7}
Time	2468
Construct Type	Initial
Parent Module	
Source Information	/remote/scratch14/yichun/0401/tsv.simx/Source/unit_RUNTIME/simprofile/delay_prof/t.v:7-12

FIGURE 2. Profiling Report: Construct Information

The source code is displayed in a pop-up window. The counter of delay events are annotated to each source code line, as shown in the following screenshot.

Using simprofile with Simulation Acceleration

```

01 module top;
02 initial begin
1234 20.00% 03     repeat (1234) #1;
04     $display("block end");
05 end
06 reg a;
07 initial begin
1234 20.00% 08     repeat (1234) a = #1 $time%2;
09     $display("a=", a);
1234 20.00% 10     repeat (1234) a = #1 $time%2;
11     $display("a=", a);
12 end
13 reg b;
2468 39.99% 14 always @(a) begin
15     b <= #1 a;
16 end
1 0.02% 17 initial begin
18     #3000;
19     $display("b=", b);
20 end
21 endmodule
22

```

FIGURE 3. Profiling Report: Source Code Information

8 Timing Analysis

Simulation Acceleration Transactor Code developers allow themselves a certain level of flexibility in code being mapped to ZeBu HW. Alternatively, RTL code developers for code mapped to silicon, have strict coding guidelines to meet size, power, timing and P&R constraints. However, Simulation Acceleration Transactor code mapped to HW still needs to meet timing constraints in order to function correctly. Therefore, understanding timing reports is important. Typically, performance of HW is determined by the **zTime**, which is the maximum frequency achieved by the driver clock.

To understand driver clock timing, critical paths timing reports can be analyzed. Then, coding changes to achieve a better **zTime**, which may lead to better performance, can be made.

For more information, see the following subsections:

- [UTF File Changes](#)
- [Viewing the zTime Report for Critical Output Routing Data Paths](#)
- [Analyzing a Critical Path Using zTime](#)

8.1 UTF File Changes

The usage of complex math operators (for example, modulo, divide, multiply, and so on) in the Testbench code mapped to HW that gets mapped by default to DSP Xilinx FPGA components, are not timed correctly. Therefore, it is recommended to disable DSP mapping by providing the following command in the UTF file:

```
optimization -dsp_mult_threshold 1024 -dsp_limit 1
```

This makes sure that operators using operands with less than 1024 bits are not mapped to a DSP Xilinx component.

It is recommended that you use the more accurate timing flow, which is called the SDF flow. This flow uses SDF timing arcs from the Xilinx component library to achieve a more accurate flow.

Add the following compile options to the UTF file:

```
# Enables SDF timing
```

Paths

```
timing_analysis -post_fpga BACK_ANNOTATED
# Enable better timing engines around memory timing analysis
timing_analysis -advanced_command {improved_memory_timing on}
ztopbuild -advanced_command {enable zmem_clock_domain_instrument}
```

The first command enables SDF timing. SDF timing provides additional timing information over traditional timing flows that work without using this command. With SDF timing, typically the **zTime** is lower than the **zTime** without SDF timing. With this, the timing is more conservative and runtime performance is worse. This does not mean your design will not run functionally correct if you do not apply the sdf UTF command.

For the timing analysis flow described in this section, SDF is required. After you have completed the timing analysis and timing fixes, you should remove this command to see if you can get a functional model without using SDF.

The second and third command are recommended because they enable better timing engines around memory timing analysis.

8.2 Viewing the zTime Report for Critical Output Routing Data Paths

After the **zCui** compile is complete, you can read and analyze a **zTime** report at the system level. Since the zTime report is in HTML format, open an HTML browser, such as Mozilla FireFox, in the `<zcu.work>/backend_default` directory.

To open a file, select a file URL with the full path in your Linux environment, using the `file://` directive. For example:

```
file://<full-path-to-your-html-file>
```

Perform the following steps to open the **zTime** report and view the critical output routing data paths:

1. Open the `zTime_fpga.html` report in an HTML browser, such as Mozilla FireFox. For example:

```
file:///home/projects/zcu.work/backend_default/zTime_fpga.html
```


Viewing the zTime Report for Critical Output Routing Data Paths

The **zTime Report: Index** page appears.

zTime Report: Index

Date: Tue Jan 15 23:17:32 2019

[Filter paths](#)[Routing data paths](#)

- Click **Routing data paths**. The **zTime Report: Critical Output Routing Data Paths** report appears. This report lists the critical paths, starting from the longest critical path.

zTime Report: critical output routing data paths

Date: Tue Jan 15 23:17:31 2019

Slack	Required Time	Delay	Fpga	From	To
0 ns	1566 ns (1 driver clock)	1566 ns	3 (2)	U0_M0_F11.zcbsplit_top_2345	U0_M0_F2.zcbsplit_top_638
0 ns	1566 ns (1 driver clock)	1566 ns	3 (2)	U0_M0_F11.zcbsplit_top_2345	U0_M0_F2.zcbsplit_top_638
176 ns	1566 ns (1 driver clock)	1389 ns	3 (2)	U0_M0_F11.zcbsplit_p_in[6401]	U0_M0_F1.zcbsplit_p_zpor...
176 ns	1566 ns (1 driver clock)	1389 ns	3 (2)	U0_M0_F11.zcbsplit_p_in[6401]	U0_M0_F1.zcbsplit_p_zpor...
189 ns	1566 ns (1 driver clock)	1377 ns	3 (2)	U0_M0_F11.zcbsplit_p_in[1970]	U0_M0_F0.zcbsplit_p_zpor...
189 ns	1566 ns (1 driver clock)	1377 ns	3 (2)	U0_M0_F11.zcbsplit_p_in[1970]	U0_M0_F0.zcbsplit_p_zpor...
216 ns	1566 ns (1 driver clock)	1350 ns	3 (2)	U0_M0_F11.zcbsplit_top_1701	U0_M0_F3.zcbsplit_top_850
216 ns	1566 ns (1 driver clock)	1350 ns	3 (2)	U0_M0_F11.zcbsplit_top_1701	U0_M0_F3.zcbsplit_top_850
880 ns	1566 ns (1 driver clock)	686 ns	3 (2)	U0_M0_F11.zcbsplit_top_3800	U0_M0_F4.zcbsplit_top_271
996 ns	1566 ns (1 driver clock)	570 ns	11 (7)	U0_M3_F10.ztbsplit_p_zport_w...	U0_M0_F2.ztbsplit_top_117

FIGURE 1. zTime Report: Critical Output Routing Data Paths

The following table provides a description of the columns in the **zTime Report: Critical Output Routing Data Paths** report.

TABLE 1 zTime Report: Critical Output Routing Data Paths Column Descriptions

Column Name	Description
Slack	The amount of time in nanoseconds from the worst critical path. In the screenshot above, note that there is a sudden jump from 216 ns to 880 ns. This means that you get significant gains if you resolve the first 7 critical paths.
Required Time	This specifies how much delay is required for the most critical path. Our goal is to reduce this delay as much as possible. This column also specifies how many driver clock periods are used to complete the path. Note that the delay is based on the worst path for all the other paths.
Delay	This specifies the delay for the path.
Fpga	This specifies the number of hops required for the path. The number of hops is the number of FPGAs the path exits an FPGA and enters a different FPGA. In the parenthesis is the overall number of different FPGAs this path goes through. In the above example, there is a path with 11 hops; this should be analyzed and reduced.
From	This is the source of the path. Refer to other documentation on how to use zBrowser to find this signal in the netlist, then how to find the relevant RTL signals in the Model's code.
To	This is the target of the path.

8.3 Analyzing a Critical Path Using zTime

After opening the **zTime Report: Critical Output Routing Data Paths** report, you can view the critical paths, starting from the longest critical path.

To analyze a path, in the **Slack** column, click the hyperlink for the path you want to analyze. In this section, the fifth path is analyzed, as highlighted in the following screenshot.

zTime Report: critical output routing data paths

Date: Tue Jan 15 23:17:31 2019

Slack	Required Time	Delay	Fpga	From	To
0 ns	1566 ns (1 driver clock)	1566 ns	3 (2)	U0_M0_F11.zcbsplt_top_2345	U0_M0_F2.zcbsplt_top_638
0 ns	1566 ns (1 driver clock)	1566 ns	3 (2)	U0_M0_F11.zcbsplt_top_2345	U0_M0_F2.zcbsplt_top_638
176 ns	1566 ns (1 driver clock)	1389 ns	3 (2)	U0_M0_F11.zcbsplt_p_in[6401]	U0_M0_F1.zcbsplt_p_zpor
176 ns	1566 ns (1 driver clock)	1389 ns	3 (2)	U0_M0_F11.zcbsplt_p_in[6401]	U0_M0_F1.zcbsplt_p_zpor
189 ns	1566 ns (1 driver clock)	1377 ns	3 (2)	U0_M0_F11.zcbsplt_p_in[1970]	U0_M0_F0.zcbsplt_p_zpor
189 ns	1566 ns (1 driver clock)	1377 ns	3 (2)	U0_M0_F11.zcbsplt_p_in[1970]	U0_M0_F0.zcbsplt_p_zpor
216 ns	1566 ns (1 driver clock)	1350 ns	3 (2)	U0_M0_F11.zcbsplt_top_1701	U0_M0_F3.zcbsplt_top_850
216 ns	1566 ns (1 driver clock)	1350 ns	3 (2)	U0_M0_F11.zcbsplt_top_1701	U0_M0_F3.zcbsplt_top_850
880 ns	1566 ns (1 driver clock)	686 ns	3 (2)	U0_M0_F11.zcbsplt_top_3800	U0_M0_F4.zcbsplt_top_271
996 ns	1566 ns (1 driver clock)	570 ns	11 (7)	U0_M3_F10.ztbsplt_p_zport_w	U0_M0_F2.ztbsplt_top_117

FIGURE 2. zTime Report: Analyzing Critical Output Routing Data Paths

The following page appears:

Fpga	Delay	Arrival	XDR XTYPE	zCore	Port
Internal					DriverClock domain
U0/M0/F11	17 ns	17 ns		part_U0_M0	U0_M0_F11.zcbsplt_p_in[1970]
	148 ns		144 LVDS		part_U0_M0@UC
U0/M0/F05	8 ns	173 ns			
	135 ns		128 LVDS		
U0/M0/F00		308 ns		part_U0_M0	U0_M0_F0.zcbsplt_p_in[1970]
Internal	1069 ns	1377 ns			DutClock domain

FIGURE 3. zTime Report: Segments in Critical Path

This page provides the different segments in the path. Each segment involves exiting an FPGA (starting FPGA) and entering a different FPGA (ending FPGA). The following

table provides a description of the columns.

TABLE 2 zTime Report: Column Description of Segments in Critical Path

Column	Description
Fpga	Name of the FPGA from where the current segment begins
Delay	The time spent in a segment
Arrival	The absolute time the segment started
Port	Source signal of the segment

In the preceding screenshot, notice that the time in the FPGA specified as **internal** is unexpectedly large and sometimes seems that it repeats for all segments going from FPGA x to FPGA y. Timing in ZeBu is conservative. For each segment, the tool accounts for the most critical path known in the FPGAs in the respective segment.

The only way to improve on timing related to FPGAs is to produce a critical path report for the specific FPGA. Go to the following location to view FPGA timing reports at FPGA-compile level:

```
<path-to-zcui.work>/backend_default/U*/M*/F*/fpga_reports/vivado/  
timing_optimized_hold.rpt
```

Analysis of these critical paths is beyond the scope of this document.

9 Using Time Decoupled Mode for Performance Improvements

Simulation Acceleration enables you to mark functions, which only have inputs, so that these functions are not monitored or controlled by Simulation Acceleration transactors. This is known as time decoupling of functions, which improves performance. This technique does not affect the timestamp synchronization.

For more information, see the following subsections:

- [Consideration for Using Time Decoupling](#)
- [Use Model for the Time Decouple Mode](#)
- [Limitations of Time Decouple Mode](#)

9.1 Consideration for Using Time Decoupling

Before using time decoupling, review the following considerations:

- With the time decoupled mode, the result from run-to-run might be different. When the waveform outputting (FWC/QiWC/read-back) is enabled, the discrepancy is significant between runs that use time coupled mode and those that do not.
- The timestamp is non-deterministic in the following cases:
 - ❑ If the criterion to finish the emulation does not rely on a `#delay`, the timestamps between the testbench and HW at the end of emulation might be different and non-deterministic.
 - ❑ If a decoupled function on the testbench side called by HW triggers a primary inputs (PI), the timestamp at which this PI arrives from the HW side is non-deterministic. From a HW perspective, PI refers to the data transmitted from the testbench to DUT by anyone of following ways: Port Connections, XMR, and Functions Calls.

Port Connection

```
Module Testbench;  
  Wire tb_sig;  
  DUT u_DUT(.sig_in (tb_sig));  
Endmodule
```

XMR

```
Assign Testbench.u_DUT.dut_sig = tb_sig;
```

Function Call

```
Testbench.u_DUT.func();
```

- If a HW function takes multiple driver clocks to complete execution in the same delta time, it is not recommended to be marked as time decoupled. For the following snippet, ZEMI3 will stop CcClock until the function completes execution. Therefore, the performance cannot be improved in the time decoupled mode.

```
function void initialize_memory (input bit [31:0] init_val);  
    int i = 0;  
    while (i < 8192) begin  
        mem[i] = init_val; // mem is a 2-D memory  
        i++;  
    end  
endfunction
```

9.2 Use Model for the Time Decouple Mode

To use the time decouple mode, see the following subsections:

- [Compile-Time Use Model](#)
- [Runtime Use Model](#)

9.2.1 Compile-Time Use Model

The following methods are available to compile functions in the time coupled mode.

- [Automatically Infer All Input-Only Functions](#)
- [Manually Specify Input-Only Functions](#)

Note

Only functions, which are compiled in the time coupled mode, are configurable at runtime time decoupled mode.

Automatically Infer All Input-Only Functions

When you use this method, Simulation Acceleration compiles **all** functions, which have only inputs ports, in the time decoupled mode. These functions are then configurable during runtime.

To automatically infer and compile all functions, which have only input ports, specify the following switch:

```
-simxl=time_decouple+input_only_function
```

Manually Specify Input-Only Functions

If you want to select specific input-only functions to compile in the time coupled mode, you need to first make RTL changes to the functions. In the RTL code, use the `simxl_time_decouple` attribute on a function that has input ports only. You can then control these functions during runtime in time coupled mode (see [Runtime Use Model](#)).

In the following RTL code snippet, the `user_func` function is compiled in time decoupled mode and configurable at runtime.

```
(* simxl_time_decouple=1 *)
function void user_func(input arg);
endfunction
```

To make sure that a function, which contains only input ports, is neither compiled in time decoupled mode, and therefore not configurable at runtime, set the `simxl_time_decouple` attribute to **0**.

To compile the functions in the time decoupled mode, pass the following compile-time switch during compilation using `simv`:

```
-simxl=time_decouple
```

9.2.2 Runtime Use Model

The `-simxl=time_decouple` runtime command enables the time decoupled mode on all the functions that are compiled for time decoupled mode. To enable the time decouple mode work normally, set the `SNPS_SIMXL_DISABLE_CCC` environment variable to 1.

To facilitate debugging during runtime, you can overwrite the default settings specified at compile time. To do this, pass a configuration file by specifying the following:

```
-simxl=time_decouple_cfg:<file path>
```

Note

If `-simxl=time_decouple+time_decouple_cfg:<>` is passed, the configuration file overrides the functions enabled by the `-simxl=time_decouple` command.

The runtime configuration file contains entries for each function call that you want to run in the time decoupled mode. The format of each entry is as follows:

```
<module_name::tf_name>, <time_decouple_val 0|1>, <list_of_ids  
corresponding to the tf>
```

Simulation Acceleration outputs a sample runtime configuration file, which you can copy, edit, and use in your runtime. The location of this file is as follows:

```
zcui.work/default_backend/time_decouple_cfg.csv
```

In addition, see the following file for detailed information on each function call. Information, such as function call ID with reference to the compilation and information about function call scope and location, is available. The location of this file is as follows:

```
zcui.work/default_backend/time_decouple_tf_call_info.json
```

9.3 Limitations of Time Decouple Mode

The following limitations exist with this Simulation Acceleration feature:

- Functions with output ports are not supported for the time decouple mode. VCS reports an NYI error at compile time.
- Tasks are not supported for the time decoupled mode.
- Under the compile time switch, `-simxl=time_decouple`, the `$display` family of system tasks and UVM calls do not have time decouple support.

10 Enabling zDPI in Simulation Acceleration

ZeBu supports a subset of DPI-C standard through the zDPI, where only the data is going from HW to SW. Therefore, if the HW design contains import DPI functions with only inputs as arguments and no return value, DPI functions can process using zDPI.

You can enable zDPI support in Simulation Acceleration in both [V2Z Mode](#) and [V2VX Mode](#). For more information, see the following topics:

- [Use Model for Enabling zDPI in the V2Z Flow](#)
- [Use Model for Enabling zDPI in the V2VX Flow](#)
- [Limitations of zDPI Support in Simulation Acceleration](#)

10.1 Use Model for Enabling zDPI in the V2Z Flow

To enable zDPI in the [V2Z Mode](#), you need to make compilation and runtime changes, as described in the following topics:

- [Compile-Time Use Model](#)
- [Runtime Use Model](#)

Compile-Time Use Model

To enable zDPI, specify the following UTF command:

```
simx1_enable_zdpi -all <0|1> | -modules {modules list}
```

Where:

- `-all`: Use to enable zDPI for all modules.
- `-modules`: Use to enable zDPI for specific modules. This option takes a space-separated list of modules. For the specified modules, Simulation Acceleration skips its own DPI processing and the specified modules get processed as zDPI by Unified FrontEnd (UFE).

In addition, specify the following UTF command:

```
dpi_synthesis -enable ALL
```

Runtime Use Model

By default, zDPI is disabled at runtime. To enable zDPI at runtime, specify the following option:

```
-simxl=zdpi:path_to_so
```

If the zDPI allows multiple shared objects, specify the `-simxl=zdpi:path_to_so` option for each object.

10.2 Use Model for Enabling zDPI in the V2VX Flow

In the *V2VX Mode*, use the following option to enable zDPI globally:

```
-simxl=enable_zdpi
```

To enable zDPI for specific modules, use the plus sign (+) to specify the module names. For example, the following command specification enables zDPI in module1 and module2:

```
-simxl=enable_zdpi+module1+module2
```

10.3 Limitations of zDPI Support in Simulation Acceleration

The following limitations are applicable to all imported DPI in zDPI modules:

- Only input ports are supported. The following are not supported: output ports, inout ports, and ref ports
- The import DPI should not have a return value
- Import DPI tasks are not allowed because they may potentially be blocking

If these guidelines are not followed and zDPI is enabled on modules, an error is reported.

However, if zDPI enabled globally, instead of module-wise, a message is not reported and zDPI might not be able to work correctly.

11 Estimating DUT Power Requirements Using Power Profiling

The Power Profiling feature in Simulation Acceleration enables you to estimate the power requirements of the DUT, which resides in the hardware. During Simulation Acceleration runtime, power profiling is performed using the `uip` UCLI command.

For more information, see the following topics:

- [Enabling Universal IP \(UIP\) Power Profiling](#)
- [Use Model of the `uip` Command](#)

11.1 Enabling Universal IP (UIP) Power Profiling

To enable UIP power profiling, specify the following UTF command at compile time:

```
power_profile -weight_profile <file>
```

Where, `-weight_profile <file>` specifies the power weight file generated by SpyGlass.

11.2 Use Model of the `uip` Command

In Simulation Acceleration, you specify `uip` UCLI commands in runtime. To perform power profiling in Simulation Acceleration, you can specify a UCLI script or enter commands manually in UCLI mode.

The `uip` command is used with the following options:

```
uip
  -defined
  -init
  -close
  -list atc_threshold|enabled|threshold|weight -domain <name>
  -output_dir [-set <path>]
```

```

-total weight|threshold|atc_threshold
-mode [-per_bucket|-total_bucket]
-bucket -config_file [-default|-import <path>|-export <path>]
-bucket -name <name> -enable|-disable
-bucket -name <name> -threshold|-atc_threshold|-weight <value>
-bucket -name <name> -create_from <bucket_list>
-bucket -name <name> -delete
-dump -enable|-disable

```

Where:

- **-defined:** Used to verify whether power profiling is available in the session.
- **-init:** Used to initialize power profiling.
- **-close:** Used to close and flush all power profiling operations.
- **-list atc_threshold|enabled|threshold|weight -domain <name>:** Used to list property values of domains.
- **-output_dir [-set <path>]:** Used to retrieve the current output path of power profiling or set the output path of power profiling.
- **-total weight|threshold|atc_threshold:** Used to retrieve the total value of a specific property.
- **-mode [-per_bucket|-total_bucket]:** Used to retrieve the current UIP run mode. You can set the run mode by specifying either **-per_bucket** or **-total_bucket**.
- **-bucket -config_file [-default|-import <path>|-export <path>]:** Used to set the uip configuration values to default values. Also, used to set the configuration values from a imported configuration file. In addition, this option is used to export current configuration values to a file.
- **-bucket -name <name> -enable|-disable:** Used to enable or disable a bucket for power profiling.
- **-bucket -name <name> -threshold|-atc_threshold|-weight <value>:** Used to set the property values of a given bucket to a given value.
- **-bucket -name <name> -create_from <bucket_list>:** Used to create a new bucket from a list of existing buckets.

Use Model of the uip Command

- `-bucket -name <name> -delete`: Used to delete a bucket from power profiler.
- `-dump -enable|-disable`: Used to enable and disable power profiler dumping operations.

Sample UCLI Script Using uip Command Specifications

The following is a sample UCLI script that has uip command specifications for power profiling.

```
==== ucli.tcl====  
uip -defined  
uip -init  
uip -mode  
uip -output_dir  
uip -bucket -config_file -default -export ./default_config.txt  
uip -dump -enable  
run 1000  
uip -dump -disable  
exit
```

