ZeBu® Server LCA Features Guide

Version O-2018.09-SP1, June 2019

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Trademarks

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

Free and Open-Source Software Licensing Notices

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

Third-Party Links

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

www.synopsys.com

Contents

List of Figures

Preface

About This Book

The *ZeBu® Server LCA Features Guide* describes the Limited Customer Availability (LCA) features and enhancements that are available in the ZeBu Server O-2018.09 release.

The LCA features are categorized as follows:

- Compilation stability improvements
  - Enhanced **zPar** routing
  - Performance Driven Multi-die Multiplexing Support
  - Iterative block selection in partitioner
- **zTime** frequency improvements
  - Half-board partitioning support in ZeBu Server 4
  - Analytic pin assignment
  - Architecture-aware partitioning
- Language support
  - Verilog switch-level modeling support in compilation

Contents of This Book

The *ZeBu® Server LCA Features Guide* has the following sections:

Section	Describes...
Language Support	features to improve the Verilog standard support in the compilation flow
Compilation Features	features to improve the compilation time, stability, and theoretical design performance
Runtime Features	features to improve the runtime performance

Section	Describes...
ECO	usage of ECO feature
SimXL	usage and features of SimXL flow

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;
Reworked 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 once or multiple times
| (Vertical bar)	A list of choices out of which you can choose one
. . . (Horizontal ellipsis)	Other options that you can specify

Related Documentation

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

1 Language Support

This section provides information about LCA features introduced to improve ZeBu usability with O-2018.09.

1.1 Verilog Switch-Level Modeling Support in Compilation

The ZeBu compilation flow supports Verilog description in the switch-level modeling to describe how these switch-level networks can be handled and how it should behave when detecting an unsupported case.

SystemVerilog allows scalar net signal values to have a full range of unknown values and different levels of strength or combinations of levels of strength. This multiple-level logic strength modeling resolves the combinations of signals into known or unknown values to represent the behavior of hardware with improved accuracy. Therefore, SystemVerilog provides accurate modeling of signal contention, bidirectional pass gates, resistive MOS devices, dynamic MOS, charge sharing, and other technology-dependent network configurations.

This section describes the following subtopics.

- *Enabling Switch-Level Modeling Support*
- *Limitations*

1.1.1 Enabling Switch-Level Modeling Support

To enable or disable the switch-level modeling in SystemVerilog, use the following UTF command:

```
design -convert_switches true
```

1.1.2 Limitations

The switch-level modeling does not implement dynamic strength resolution because the additional logic to be inserted while manipulating all the strengths is large.

2 Compilation Features

This section provides information about LCA features introduced to improve compilation time, stability, and theoretical design performance.

This section describes the following subsections:

- *Iterative Block Selection in Partitioner*
- *Half-Board Partitioning Support in ZeBu Server 4*
- *Architecture-Aware Partitioning*
- *Enhanced zPar Routing*
- *Analytic Pin Assignment*
- *Performance Driven Multi-die Multiplexing Support*
- *zTime Estimation Using Vivado Arc Timing Information*

2.1 Iterative Block Selection in Partitioner

An iterative block selection flow is supported to analyze partitioning results, including resource overflow and high cuts. It also identifies the block that needs further review or analysis. In the next iteration, incremental block selection is automatically done based on the first iteration. The selected blocks from the analyzer are reviewed in a sequence and the better solution between two iterations is chosen as a final solution. The iterative flow can be applied to both system-level and **zCore**-level partitioning.

The flow diagram illustrates the iterative block selection flow.

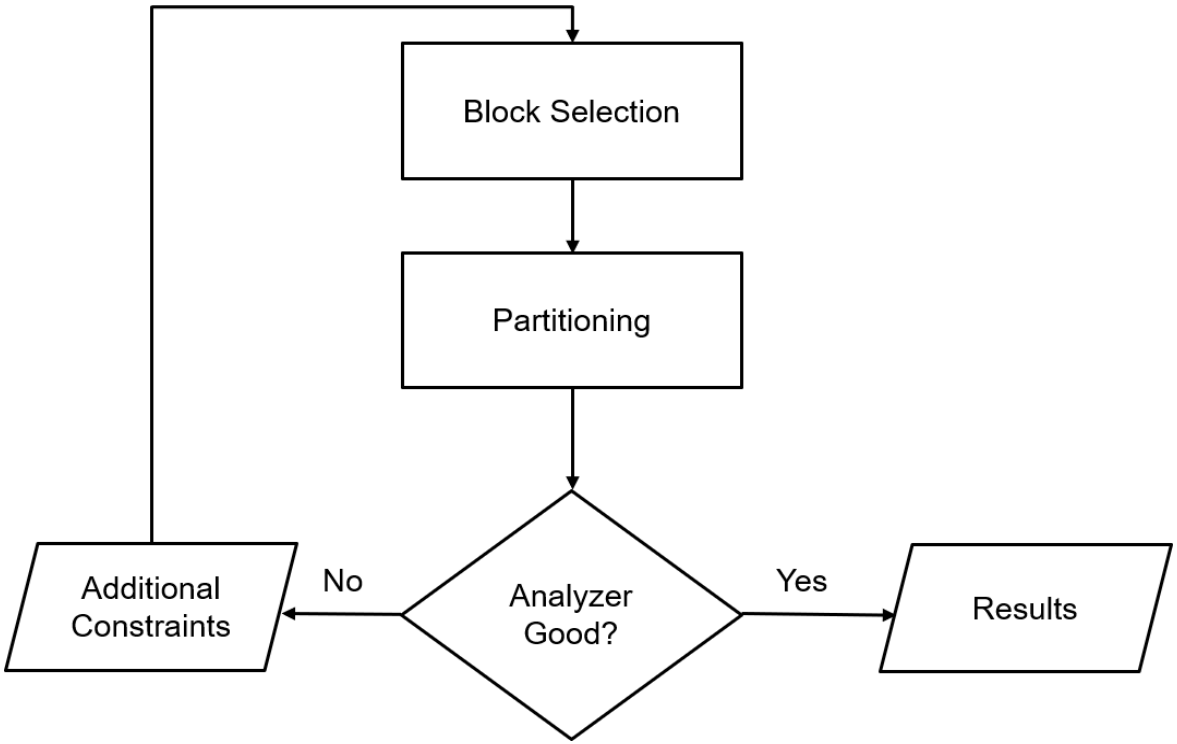


FIGURE 1. Iterative Block Selection

This section describes the following subtopics:

- *Enabling Iterative Block Selection*
- *Limitation*

2.1.1 Enabling Iterative Block Selection

To enable iterative block selection at system-level (**zTopPartitioning**), use the following UTF command:

```
ztopbuild -advanced_command {ztopclustering_command {enable
iter_block_selection}}
```

To enable iterative block selection at **zCore**-level (**zCorePartitioning**), use the following UTF command:

```
ztopbuild -advanced_command {zcorebuild_command * {zcoreclustering_command
{enable iter_block_selection}}}
```

2.1.2 Limitation

The iterative block selection flow only works with **zTopPartitioning** and **zCorePartitioning**.

To activate **zCorePartitioning**, add the following UTF command:

```
zcorebuild_command * {partitioning auto}
```

2.2 Half-Board Partitioning Support in ZeBu Server 4

To improve the performance and stability of ZeBu Server 4, the placement of DUT logic is enhanced by:

- Enabling the system partitioner to view the routing resources available between half boards
- Building a half-board partitioning at system level to minimize the cuts between half boards
- Providing the necessary information at **zCoreBuild** level such that the DUT logic communicating with Multi-Gigabit Transceiver (MGT) bridges are correctly aligned to minimize hops between half boards with MGT alignment
- Providing the data hub programming with more detailed information about the resource needed at the half-board level

This section describes the following subtopics:

- *Function*
- *Enabling Half-Board Partitioning*

2.2.1 Function

Each **zCoreBuild** imports the bipartition in two half boards and can use it to pre-map the DUT logic on the appropriate FPGA bridge. With Enhanced Auto-Clustering (EAC), each half board is partitioned separately and blocks communicating to bridge are attached by MGT bridge or backplane bridge.

2.2.2 Enabling Half-Board Partitioning

To enable EAC, use the following UTF commands:

```
ztopbuild -advanced_command {partitioning auto -npf}
ztopbuild -advanced_command {cluster enable -half_board_partitioning}
ztopbuild -advanced_command {cluster enable -force_placement}
ztopbuild -advanced_command {zcorebuild_command * {cluster enable -
half_board_partitioning}}
ztopbuild -advanced_command {zcorebuild_command * {cluster enable -eac}}
```

2.3 Architecture-Aware Partitioning

During **zCoreBuild** partitioning, a cost function is provided based on probabilistic routing estimates for the partitioning engine. The cost function tries to optimize the sum of XDRs. Consequently, the sum of XDR along the critical path is optimized and **zTime** frequency is improved. The **zTime** frequency is reported in the zTime.log file.

This section describes the following subtopics:

- *Function*
- *Enabling Architecture-Aware Partitioning*

2.3.1 Function

The cost function considers the actual physical connections between FPGAs and

performs board-level partitioning (**zCores** -> **FPGAs**) to improve the performance of partitioning.

2.3.2 Enabling Architecture-Aware Partitioning

To activate this feature, use the following UTF command:

```
ztopbuild -advanced_command {zcorebuild_command * {partitioning auto -xdraware}}
```

2.4 Enhanced zPar Routing

ZeBu Weighted Congestion-driven Router (ZCR) replaces the "Routing Iterations Step" of the existing **zPar** router to:

- Enable at least one diversion to route each inter-FPGA net. Therefore, the number of diversions is reduced.
- Minimize the maximum total XDR that is a sum of the XDR of every inter-FPGA net of the timing path. Therefore, the compile time of the **zPar** router is reduced and calling **zTime** less time.
- Stop "splitting" to reduce the performance sensitivity.

This section describes the following subtopics:

2.4.1 Function

ZCR optimizes timing, diversions, feed-throughs, and compile-time based on a maze routing framework using the following methods:

- **Multi-shortest path construction:** Creates shortest paths for each routing target
- **Iterative congestion-driven routing to spread congested nets:** Provides uniform congestion to minimize XDR
- **Weighted-XDR technique to reserve dedicated XDR for critical paths:** Reserves more dedicated routing resource for critical paths for better timing

- **Restricted diversions to enlarge solution space:** Reduces FPGA congestion

2.4.2 Enabling ZCR

To enable ZCR, use the following UTF command:

```
zpar -advanced_command {System routerType wirelength_timing_coopt}
```

2.5 Analytic Pin Assignment

ZeBu Placement and Routing, or **zPar**, performs FPGA placement (assigning netlist partitions to FPGAs) and inter-FPGA net routing. As the number of inter-FPGA nets can exceed the number FPGA I/O pins, Time Division Multiplexer (TDM) is used to route multiple inter-FPGA nets through the same FPGA pin. Therefore, each FPGA pin has an XDR value, which represents the number of inter-FPGA nets connected to a specific FPGA pin.

A new analytic pin assignment algorithm, ZAP, is introduced to improve the compile time and performance of **zPar** pin assignment. **zPar** pin assignment is a step that assigns every inter-FPGA net to each FPGA pin (assigning a correct XDR value for each FPGA pin).

This section describes the following subtopics:

2.5.1 Function

ZAP models the pin assignment as integer linear programming that includes the following constraints:

- Timing constraints derived from **zTime** timing graph
- Inter-FPGA pin-pair constraints representing the number of available pin-pairs
- Possible XDR values constraints representing the restrictions on the TDMs
- Inter-FPGA nets between the same pairs of FPGAs in the same direction

2.5.2 Enabling Pin Assignment Algorithm

To enable the ZAP algorithm, use the following UTF command:

```
zpar -advanced_command {System pinAssignmentEffort zap}
```

2.6 Performance Driven Multi-die Multiplexing Support

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

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

Note *The use of zTime estimation using vivado arc timing information is mandatory. For details, see zTime Estimation Using Vivado Arc Timing Information.*

This section describes the following subtopics:

- *Enabling PDM Support*
- *zCui Flow*

2.6.1 Enabling PDM Support

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

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

where,

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

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

2.6.2 zCui Flow

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

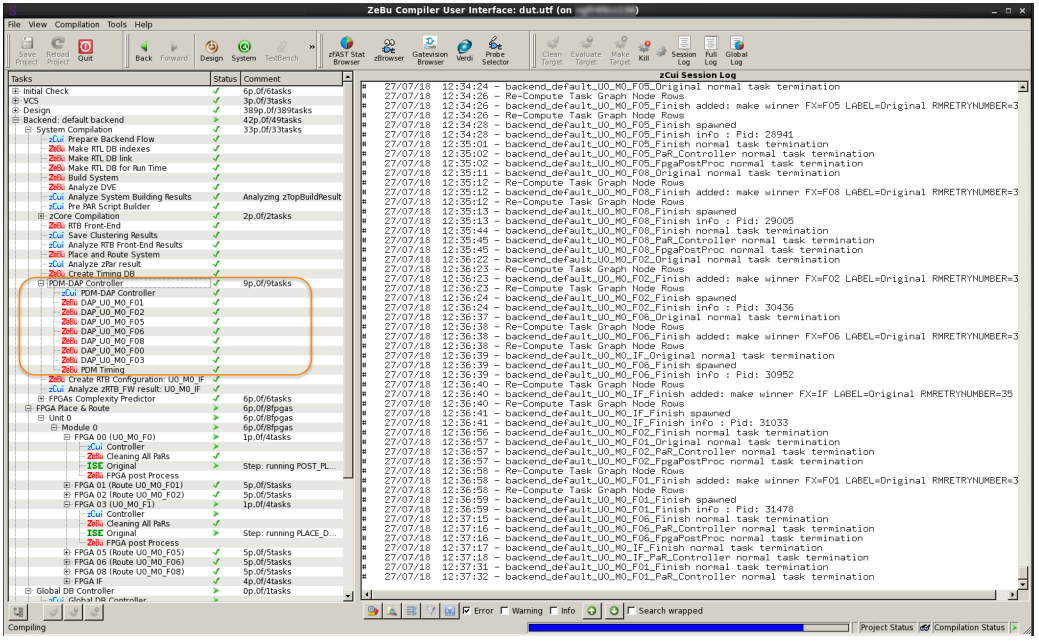


FIGURE 2. PDM Flow in zCui

2.7 zTime Estimation Using Vivado Arc Timing Information

Vivado timing arc information (SDF file) is used to accurately compute the internal FPGA delay and improve zTime estimation.

To enable this feature, set the value for `-post_fpga` as `BACK_ANNOTATED` in `timing_analysis` as follows:

```
timing_analysis -post_fpga BACK_ANNOTATED
```

In the UTF project file, if `timing_analysis -post_fpga BACK_ANNOTATED` is present, **Create Timing DB (SDF Mode)** task is added in **zCui** during FPGA compilation and it is shown in the following figure.

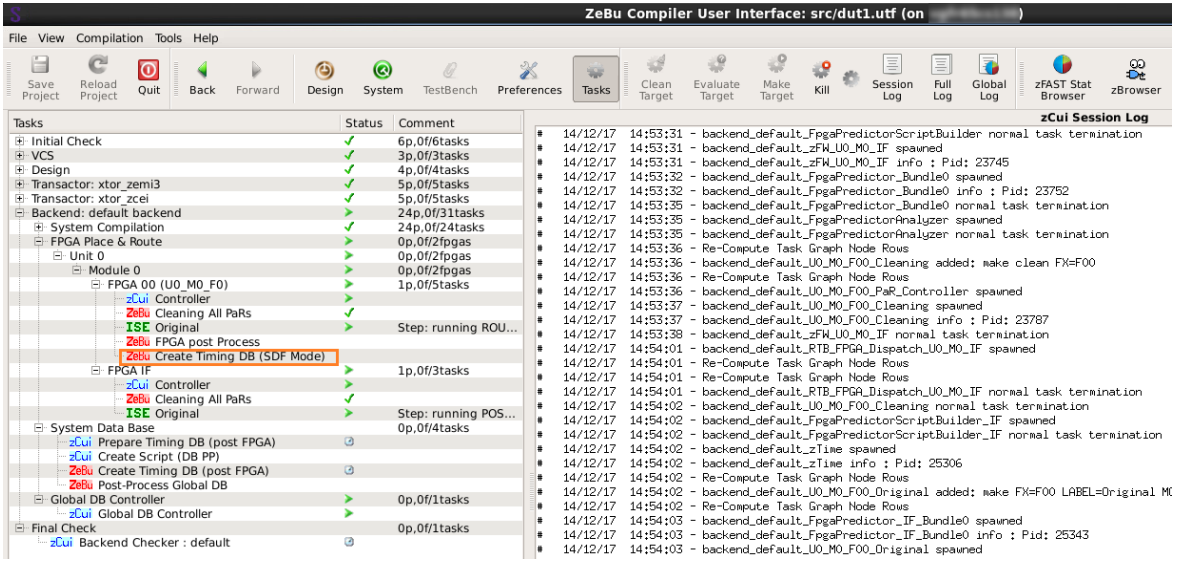


FIGURE 3. Vivado Arc Timing Information in zCui

The internal path delays are generated during the FPGA compilation and these values can be injected into the post-FPGA design timing analysis to estimate the accurate **zTime**.

2.7.1 Limitations

Multi-cycle path detection is not handled correctly in **zTime** estimation using the Vivado arc timing information feature.

3 Runtime Features

This chapter describes the runtime LCA features in the following subtopics.

- *ZeBu Server 4 Bitstream Cache Enhancements*
- *Customized DMTCP Tool*
- *Refactoring of Timing Parameters Computation*

3.1 ZeBu Server 4 Bitstream Cache Enhancements

To reduce the loading time of a design when it is run several times on the same modules, set `ZEBU_DEBUG_CACHE_BITSTREAM_DISABLE` to `false`. Therefore, the time required to initialize a design is reduced.

You can find the relevant information in the log file as follows:

```
-- ZeBu : 18:44:44.933937   39 : 269108 : zServer : NOTE : 480
Bitstreams have been programmed from cache.
-- ZeBu : 18:44:44.934998   39 : 269108 : zServer : Ux_HMx_Fxx
loading, all threads statistics : real time = 5 s, user time = 2 s,
system time = 2 s
```

For example, a design uses all modules of a 10 unit system. When the design is run without setting the preceding environmental variable, it takes 128s to load the FPGAs. When the design is run after setting `ZEBU_DEBUG_CACHE_BITSTREAM_DISABLE` to `false`, it consumes 134s in the first run and 38s for the subsequent runs.

3.2 Customized DMTCP Tool

A custom Distributed MultiThreaded Checkpointing (DMTCP) 2.5.2 tool is introduced in ZeBu to save an emulation hardware and software state to restart the emulation later. This tool enables you to restart the emulation before a point of interest (for example, after an OS boot sequence) for debug purpose.

3.2.1 Environment Setup

To use this feature, set the following environment variables:

```
setenv DMTCP_HOME <path_to>/dmtcp-2.5.2
setenv PATH ${DMTCP_HOME}/bin:${PATH}
setenv LD_LIBRARY_PATH ${DMTCP_HOME}/lib:${LD_LIBRARY_PATH}
setenv ZEBU_CKPT_USE_ALL_ENV OK
```

To run a job with this DMTCP, use the following command:

```
dmtcp_launch --modify-env <run command>
```

3.2.2 Saving and Restoring an Emulation State

You can use the following script commands based on the interface you chose to run the job and save the emulation state:

- **zRun**: `ZEBU_checkpoint "checkpointname"`
- **zRci**: `checkpoint -fullsave <checkpointname>`
- **C++**: `Board::checkpoint ()`

To restart a checkpoint, use the following command:

```
<checkpointname>.n/dmtcp_restart_script_zebu.sh
```

Where,

- `n` is a placeholder for an integer

3.3 Refactoring of Timing Parameters Computation

A new software library that is shared between **zTime** and runtime is introduced to improve the calculation of the timing parameters such as, Driver Clock frequency, `zClockSkewTime` and `zClockFilterTime`.

During the compilation, **zTime** considers the runtime constraints to compute the values.

The runtime considers the timing constraints computed by **zTime** to program the ZeBu hardware.

To activate the shared library, set `ZEBU_USE_CONSTRAINTS_LIB` to `YES`.

When the shared library is activated, observe the following during compilation in `zTime.log`:

```
#   step SAVE RUN PARAM : $driverClk.Frequency = 8333 kHz (8333 kHz
before rounding)
```

When the shared library is activated, observe the following during runtime in

zServer.log:

```
-- zServer : Using constraints library
```

4 ECO

The newly developed Engineering Change Order (ECO) engine provides the capability to:

- Capture new signals using FWC technology
- Capture new hierarchies using QiWC technology
- Add force/inject signals

This is applicable on a reference compilation database when hardware resources have been initially allocated. This is the ECO-Aware compilation.

There are two main use-models:

- New `$dumpvars` or `zForce`: **zCui** performs the incremental compilation
- CEL file supplied for Runtime Trigger: **zECO** performs the incremental compilation

The ECO technology is applicable to any ECOable signals in the design. This is true for any signals directly connected to an FPGA register or latch. This information about ECOability could be get from the ECO-Aware Compilation with the provided check feature described later.

This section describes the features in the following subtopics.

- *ECO-Aware Compilation*
- *ECO Flow for Incremental Compilation (with zCui)*
- *ECO Flow for Runtime Triggers*

4.1 ECO-Aware Compilation

To prepare initial/reference compilation with default resource reservations for future ECO flow activation, use the following UTF command:

```
utf> eco
```

The full command prototype is as follows:

```
eco
    [-recompile_on_error <NEVER|ON_OVERWRITE_ONLY>] # Specify the
recompilation mode on error
    [-skip_checking <bool>] # Enable/Disable the skip checking design changes
    [-reserve_force <int>]  # Force bit number to reserve in each FPGAs
    [-reserve_fwc <int>]    # FWC bit number to reserve in each FPGAs
    [-reserve_qiwc <int>]   # QiWC bit number to reserve in each FPGAs
```

4.2 ECO Flow for Incremental Compilation (with zCui)

The incremental compilation is applied with **zCui** in the following cases:

- New `$dumpvars`: **zCui** performs the incremental compilation
- New Force/Inject: **zCui** performs the incremental compilation

4.2.1 zCui: Batch Mode

There are two ECO compilation modes that exist from the **zCui** command line as follows:

- Overwriting mode (reusing the same zcui.work than the initial compilation)

```
# Overwriting mode usage
%> zCui –u <utf_file> --eco -w <new zcui.work>
```

- Non-overwriting mode (reusing the initial zcui.work as a reference)

```
# Non overwriting mode usage
%> zCui –u <utf_file> --base <reference zcui.work> --eco -w <new
zcui.work>
```

In the non-overwriting mode, the ECO-aware compilation must be supplied with the **zCui**'s *--base* option as the reference `zcui.work` for the ECO-enabled

incremental compilation.

There are other **zCui** options available for the ECO flow as follows:

- `--ecoSkipChecking`: Skips design change and analyzes the module with `dumpvar` only.
- `--ecoRecompileOnError`: Recompiles the design from scratch when ECO is not applicable.

To enable **zECO** in **zCui** compilation, which automatically prepares the essential data (write the checkpoint, routed.dcp) of each chip and also reserves the required resources for **zECO**, use the following command:

```
zCui --eco --base <zcui.work> -w <new_zcui.work> -n
```

If the reserved bit number is not specified for each resource, the default number is taken.

4.2.2 zCui: GUI Mode

To run **zECO** from **zCui**, perform the following steps:

1. Click **Make Target**  from the toolbar.

   The **Choose the components to compile** dialog box opens. It lists the parameters specific to ECO.
2. Click **Default** in the **Backends** section.

   NOTE: *Only default backend is supported.*
3. Click the browse button to navigate to the base directory in **ECO Base Name** section.

   NOTE: *If the base directory is left blank, the over-writing mode is used.*
4. Select the **ECO SkipChecking** check box.

   NOTE: *Do not select the* **ECO Recompile On Error** *check box because this option is not supported.*

   The following figure displays the **zCui** with the ECO options.

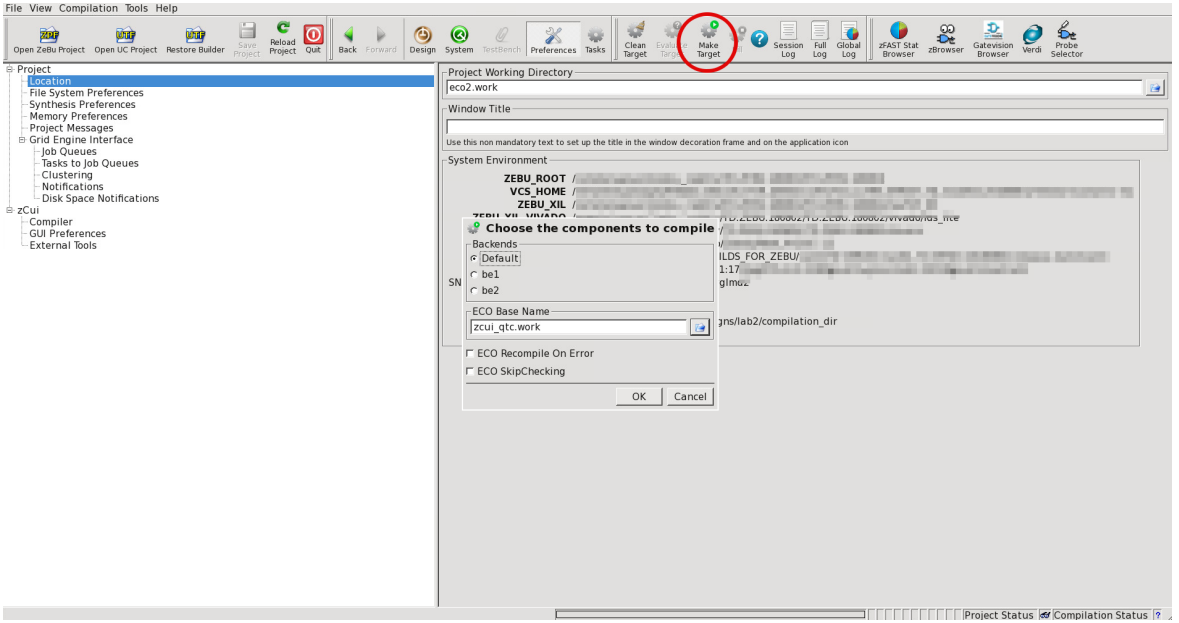


FIGURE 4. zECO in zCui

4.3 ECO Flow for Runtime Triggers

In the context of the Runtime Trigger, the CEL file is the entry point for **zECO** to make FSM signals capturable with the FWC technology.

4.3.1 ECO-Aware Compilation

To enable ECO-aware compilation to reserve FWC in the original compilation for runtime trigger optimization by **zECO**, use the following UTF command:

```
eco -reserve_fwc 1024
```

4.3.2 Launching the ECO flow for Runtime Triggers

To launch **zECO** for runtime triggers, use the following command:

```
zECO -z zcui.work -do <incr_eco_for_cel.tcl> -dest new_zui.work
```

Where,

- `<incr_eco_for_cel.tcl>` contains the following:
  - `zeco_add_dbg_signal_file -ip fwc -cel_file <path of CEL file>`: Adds all the new debug signals that are declared directly under the CEL file and are inferred to FPGA FF/LD.
  - `zeco_config -no_dcp`: Disable writing the Vivado checkpoint.
  - `zeco_config -post_pnr_timing off`: Enables/disables the post-pnr **zTime** analysis.
  - `zeco_config -pack_vector true`: Enables/disables the vector packing with alignment with 32-bits word.
  - `zeco_config -grid "<qrsh/lsf>"`: Specifies the grid commands.

The signals on which **zECO** has been applied become available with the `ZECO_PRESERVE_FWC` Value Set.

The `ZECO_PRESERVE_FWC` Value Set (and potentially other Value Set targeting non ECOed signals) must be added to ZTDB attached to the runtime trigger.

4.3.3 zECO Tool Usage

4.3.3.1 Checking Signals ECOability

It is strongly recommended to use `-checkECO` before launching the ECO incremental flow, as follows:

```
zECO -checkECO -z <database> -signal <signal name or scope name> -
signal_file <filename> -depth <depth> -check_type <force/debug>
```

Where,

- `-checkECO`: Specify to run **zECO** in check ECOable mode.
- `-signal`: Signal or instance to check.
- `-depth`: Depth to query for instances; if it's not specified, default value is 1.
- `-check_type`: Specify the type to check; if it's not specified, default value is debug.
- `-signal_file`: A text file; each line contain signal name and depth. If depth is not specified, the default value is 1. Both comma separated or space separated is supported.
- `–Output`: All queried signals with ECOable information, output to console/ zECO.log.

Example 1

```
zECO -z zcui.work -checkECO -signal hwtop.dut -depth 2 -check_type force
```

The tool reports all signals within depth 2 of `hwtop.dut` with force ECOable information.

Example 2

```
zECO -z zcui.work -checkECO -signal_file siglist -check_type force
```

4.3.3.2 General Usage

To view the usage of the **zECO** general options and environment options, use the following command:

```
zECO  [<General options>] [<Environment options>] -z <dirname> [-do
<file>]
```

Where,

- `<General options>`
  - `-z | -zebu.work <dirname>`: Specifies the compiled ZeBu database; default is ./zebu.work
  - `-do < file>`: Specifies the ECO Tcl file for play
  - `-h`: Shows this help message
  - `-dryrun`: Reports the ECO summary and generates the related script without running
  - `-drycont`: Continues ECO processes of the ZeBu database generated in dry-run mode.
  - `-dest <dirname>`: Specifies the destination directory

    NOTE: *If this option is specified, the ECO result is saved in the specified directory.*
- `<Environment options>`
  - `-vivado <vivado executable>`: Specifies the Vivado executable. This can override the Vivado present in `ZEBU_ROOT` or specified by the environment variable, *`VIVADO`*.
  - `-grid <grid command>`: Specifies the grid commands

    NOTE: *This option can override the grid command specified by the environment variable, `REMOTECMD`.*

4.3.4 Limitations

The following are the various limitations of **zECO**.

- **zCui**: `RecompileOnError` in the non-overwriting mode is not allowed.
- **VCS**:
  - Only the following statements are parsed in the VCS stage for this flow:
    - `$dumpvar`
    - `$dumpports`

- **zECO**:
  - There are chances that the incremental Place and Route (PNR) might fail due to routing failures. If PNR runs more than 120 minutes, **zECO** treats it as a failure to prevent hanging situation.
  - Consider the routability of the design. The ECO signals of each chip are suggested to be <1000. If it exceeds this number, **zECO** displays the warning messages.
  - New signals are not ECOable.
  - Debug signal
    - Wildcard in signal/instance name is not supported.
    - Signal is not in the design.
    - Signal is optimized.
    - Fwc/QiWc resource is not sufficient.
  - Forced signal
    - Only the dynamic force is supported (`zforce` not `force_dyn`) and the static force/inject is not supported.
    - Signal is not in the design.
    - Signal is optimized.
    - Signal is a composite type.
    - Signal appears in multiple locations.
    - Force resources are not sufficient.

	Current Support (2018.09-SP1)
`zforce`	Yes
`-fnmatch`	No
`-mode` *`DYNAMIC`*	Yes
`-mode` *`STATIC`*	No
`-module <string>`	No
`-object_not_found` *`fatal`*	Yes
`-object_not_found` *`warning`*	No

	Current Support (2018.09-SP1)
-pin <list>	Yes
-pin_file <file>	Yes
-pin_only	No
-rtlname <list>	Yes
-rtlname_file <file>	Yes
-signal <list>	No
-sync_enable	Yes
-wire <list>	No
-wire_file <file>	No
force_dyn	No

  - ❒ For dumpvar instance, only signals in the instance that is ECOable are added.
  - ❒ The environment variable ZEBU_DV_QA_ID_CHECK is not supported.
  - ❒ Only signal addition is supported and the removal of probed signal is not supported.
  - ❒ Value-Set:

    Due to resource limitation, dumpvar signals are added to implicit Value-Sets (different from ones specified in the dumpvar statement). These implicit Value-Sets are automatically added at runtime when you add explicit Value-Set. You do not have to add implicit Value-Sets at runtime.
- The newly added debug signals are supported only if they can be mapped from RTL to gate, are inferred to FPGA FF/LD, and can be located in routed.dcp.
- If the newly added debug signals are inferred to memory, they are not supported.

5 SimXL

This section provides information about SimXL. See the following subsections:

5.1 Introduction to SimXL

The usage of SimXL 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.

SimXL enables communicating between SW and HW parts using SystemVerilog mechanisms, such as ports on a module or interface and calling of subroutines implemented in an interface.

Intended Audience

This SimXL documentation is written for design engineers to assist them with using SimXL in ZeBu.

These engineers should have knowledge of the following Synopsys tools:

- ZeBu Server
- VCS
- Verdi

Related Documentation

The following documents provide additional information related to this manual:

- The ***ZeBu Server User Guide*** provides information on ZeBu Server and its components to emulate a design.
- The **Verdi Coverage User Guide and Tutorial** document provides information on using Verdi for visualizing coverage data.

5.2 SimXL Cosimulation Flow

You can deploy the SimXL cosimulation flow on a working VCS test environment. The following modes are used in the SimXL cosimulation flow:

- *V2VX Mode* is used to compile and run a test case in the environment using VCS only to validate the readiness of environment for cosimulation.
  - 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 SimXL 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.
- *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 SimXL Transformations and Mapping
    - Incorrect RTL code generated by Behavioral Compiler.
    - Usage of unsupported features.
    - 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

- SIMV is used to run the cosimulation environment to generate waveforms and log files.

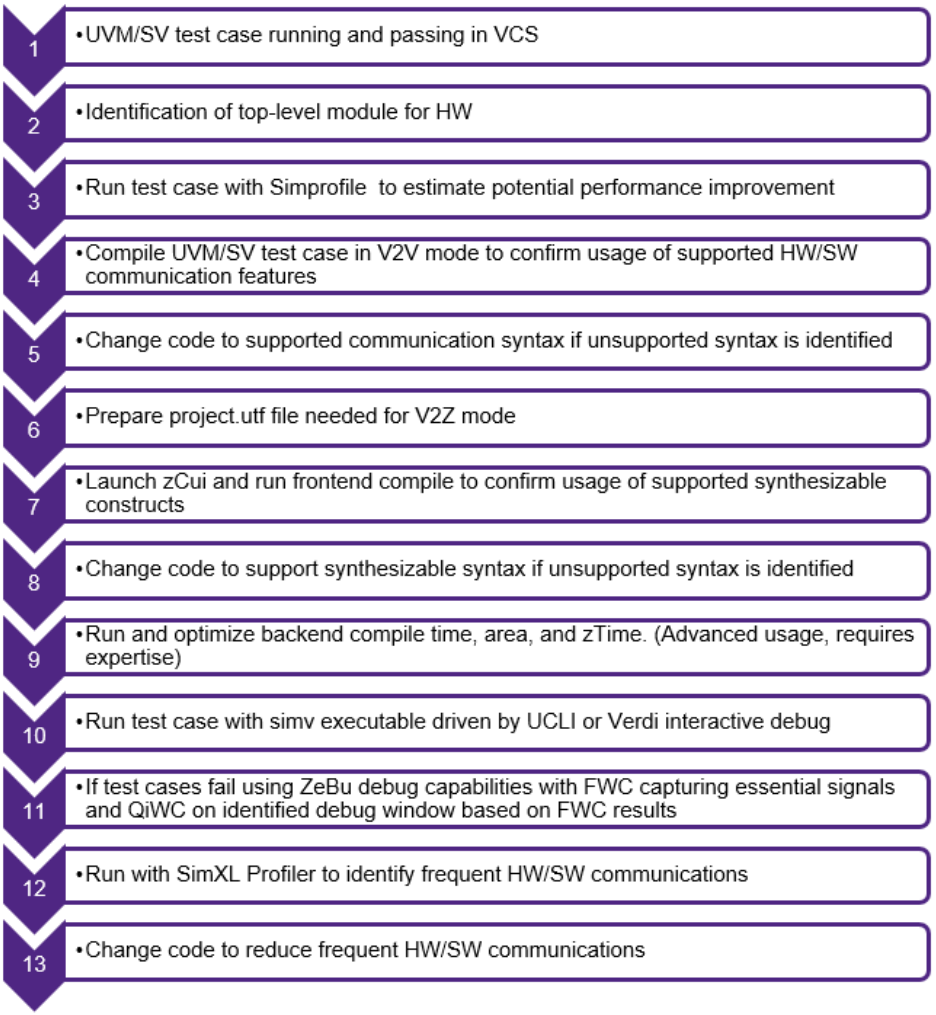


FIGURE 5. SimXL Cosimulation Flow

5.3 SimXL Compile and Runtime Options

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

- *V2VX Mode*
- *V2Z Mode*

V2VX Mode

- 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: `-Xhwcosimtest=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

- Uses ZeBu for HW compilation and VCS for SW compilation.
- The ZeBu `zCui` tool does the full compile with a UTF file as input and the UTF file pointing to the VCS compile using the `VcsCommand` UTF command.
- For compile, add the following UTF options and VCS compile options:
  - UTF command to enable SimXL: `simxl –enable true`
  - UTF command to specify HW top-level instance: `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, it is the name of the module.

- 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>`.

When your intention is to have multiple HW top instances using the commands above, note the following restrictions 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.
- HW module cannot be instantiated in another HW 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 DUT instance.
- HW top instance cannot be within a hierarchy that has been moved to SW.
- HW top instance cannot be an instance-array of size more than 1. For example:

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

- If one specifies "HWTOP" as hwtop module, an NYI error is reported.

Specification of waveform dumping using `$dumpvars` can be done in any of the hwtop modules or you can have a hwtop module just with the `$dumpvars` specifications.

SystemC specific limitations are as follows:

- Two inout ports of DUT 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 above scenario.

- DUT ports connected through SystemC would not be optimized (only impacts Performance, no funtional 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 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
  ```

- VCS additional compile option to partition the cosim controller:

  ```
  -Xhwcosim=xtor_split
  ```

  Sometimes, the cosim controller is large and cannot fit in a single FPGA. A good strategy to overcome this issue is to use this option to partition the module in the frontend.

- VCS additional compile option to specify the UTF file:

  ```
  -emulation_config=project.utf
  ```

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

  ```
  -Xhwcosim=diag
  ```

  To analyze diagnostic data, perform the following:

  - Open `zCui` compilation GUI is opened using the following: `zCui -u project.utf`.
  - Push the **Make Target** button in the `zCui` GUI to start the compile process.

- Run using the following SIMV additional runtime option to point to `zcui.work` for HW database:
  `+zcui.work=<zcui.work directory generated by zCui>`.

5.4 Supported Syntax for Using Signals as the Cosimulation Communication Mechanism

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*
- *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*
- *Limitations With Signal-Based Communication*

5.4.1 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
```

5.4.2 Signal Communication Using Aggregate Data Types

Signal communication using the following aggregate data types:

- Packed arrays
- Unpacked arrays
- Multidimensional arrays
- Packed structs
- Unpacked structs
- Unions

Any composition of these data types is supported in SimXL.

Example

```
typedef struct {
    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
```

5.4.3 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
```

5.4.4 Signal Communication Using XMRs

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.	`module dut(input a);`
	`endmodule`
	`module tb;`
	`assign dut.a = 1'b1;`
	`endmodule`

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 SimXL.	`module dut(input a);`
	`reg b;`
	`endmodule`
	`module tb;`
	`reg c;`
	`initial`
	`  c = dut.b;`
	`endmodule`

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 SimXL.
	`module dut(input a);`
	`reg b;`
	`endmodule`
	`module tb;`
	`reg c;`
	`initial`
	`  dut.b = c;`
	`endmodule`

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.	`module dut(input a);`
	`reg b;`
	`endmodule`
	`module tb;`
	`reg c;`
	`initial`
	`  force dut.b = c;`
	`endmodule`

Signal communication using XMRs that require named block as a segment in the hierarchical access.	`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`

5.4.5 Optimizing Performance for Clock XMRs From SW to HW

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 iteration of clocks completes. To enable an optimization such that there are not frequent synchronizations, use the following VCS elaboration command:

```
-Xhwcosim=clkopt -Xhwcosim=clkopt1
```

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

5.4.6 SW Accessing HW Memory Using $readmem and $writemem System Tasks

SW accessing HW memory using `$readmem` and `$writemem` system tasks is supported in SimXL. All variations of these tasks are supported:

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

Start and finish address is 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
```

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

The following system tasks are supported in SimXL:

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

$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. 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. You can specify an absolute path or a relative path to the object.

$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 quotes. The specified value must be appropriate for the type and be specified in valid syntax for any radix, as supported by the language.

5.4.8 Signal Communication Through External Subroutines

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

Declare signals in software to ensure the type of access is known before compile time. Use the `$hw_read`, `$hw_write`, and `$hw_force` VCS built-in system tasks.

Example

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

5.4.9 Limitations With Signal-Based Communication

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

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

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

  ```
  module dut(input 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 interfaces have the following limitations on those interfaces:
  - Should not contain clocking blocks
  - Should not have subroutines that are called from HW
  - Should not have signals driven simultaneously by both SW and HW
  - 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
endmodule
```

```
module tb;
   initial
      top_legal.illegal.loop3 = 0; // illegal
endmodule
```

5.5 Supported Syntax for Using Subroutines as the Cosimulation Communication Mechanism

For more information on the supported syntax for using subroutines as the cosimulation 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 are Supported*

5.5.1 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 subroutine cannot be called concurrently from two different places, regardless of whether it is automatic or not.
- 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 Cosimulation 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
```

5.5.2 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 Cosimulation 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
```

5.5.3 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 Cosimulation Communication Mechanism*.
- Ref modifier is not supported.

5.5.4 Streaming Export Methods

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

```
-Xhwcosim=streaming_sw2hw.
```

With this option, if a HW task/function has no return-value/output-port/delay, the SW2HW call of that task/function is made streaming. Therefore, the SW thread would not suspend.

5.5.5 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 support UVM messaging string support and system tasks $sformatf/ $sformat can be used to compose message context.

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:

-Xhwcosim=UVM_COMPILE_VERBOSITY+<verbosity-level>

SimXL will compile out the UVM message calls whose verbosity level cannot be met.

If there is no verbosity level provided (for example, -Xhwcosim=UVM_COMPILE_VERBOSITY), SimXL 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 the user provides -Xhwcosim=UVM_COMPILE_VERBOSITY+UVM_MEDIUM, SimXL 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.

5.5.6 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 time-consuming export task or function. VCS runtime allows import task to call time-consuming export task or function.
- When an import task or function calls an export, the import must be a task. SimXL 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
```

5.5.7 $test$plusargs, $value$plusargs are Supported

$test$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;
initial begin
   if ($test$plusargs("HELLO"))
     $display("Hello argument found.");
   if ($value$plusargs("TEST=%d", i1))
     $display("value was %d", i1);
end
```

5.6 Other Supported Features in SimXL

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

- *Interface Appearing in Both SW and HW*
- *Moving a HW Module/Interface to Execute in SW*
- *SVA Assertions are Supported in SW and HW That Is Not Mapped to IF FPGAs*
- *Miscellaneous Features*

5.6.1 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
```

5.6.2 Moving a HW Module/Interface 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 SystemVerilog Attributes or Comment Pragmas Embedded in Source Code*
- *Using an Embedded Source Code SystemVerilog Attribute, Comment Pragma, or UTF Command*
- *Using SystemVerilog Attributes or Comment Pragmas Embedded in Source Code*
- *Using an Embedded Source Code SystemVerilog Attribute, Comment Pragma, or UTF Command*

5.6.2.1 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*, the following needs to be added to the VCS compile-time options:

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

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

- For *V2Z Mode*, 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.

5.6.2.2 Using SystemVerilog Attributes or Comment Pragmas Embedded in Source Code

The module or interface immediately followed by pragma or attribute would be moved to SW.

If pragma is used at any other place in Verilog code, except for just before module or interface definition, it would be ignored.

Example

```
// Example using comment pragma
// synopsys simxl_move_to_tb
module mod;
   // module contents
endmodule
```

```
// Example using SV attribute (* simxl_move_to_tb *)
module mod;
   // module contents
endmodule

// Example using comment pragma
// synopsys simxl_move_to_tb
interface intf;
   // interface contents
endinterface

// Example using SV attribute (* simxl_move_to_tb *)
interface intf;
   // interface contents
endinterface
```

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

Tasks or functions implemented in modules or interfaces that are in HW can be moved to execute in SW no matter where they appear in the calling chain.

Moving tasks or functions 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).

Examples for modules provided also apply for interfaces.

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
```

5.6.2.4 Using SystemVerilog Attributes or Comment Pragmas Embedded in Source Code

The module or interface immediately followed by pragma or attribute is moved to SW.

If pragma is used at any other place in Verilog code, except for just before module or interface definition, it is ignored.

Example

```
// Example using comment pragma
// synopsys simxl_move_to_tb
module mod;
   // module contents
```

```
endmodule

// Example using SV attribute (* simxl_move_to_tb *)
module mod;
   // module contents
endmodule

// Example using comment pragma
// synopsys simxl_move_to_tb
interface intf;
   // interface contents
endinterface

// Example using SV attribute (* simxl_move_to_tb *)
interface intf;
   // interface contents
endinterface
```

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

Tasks or functions implemented in modules or interfaces, which are in HW, can be moved to execute in SW no matter where they appear in the calling chain.

Moving tasks or functions from HW to SW is useful for the following 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)

Examples for modules provided also apply for interfaces.

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
```

5.6.3 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 (like in ZS3). Use the following SimXL VCS runtime options for special control on the HW side. Use regular VCS features for assertions executing on SW side.

- Enable HW SVAs:

  `-simxl=enable_dut_sva`

- Disable specific HW SVAs:

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

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

- Disable all assertion in DUT:

  ```
  -Xhwcosim=disable_assertion_in_dut
  ```

5.6.4 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 HW

5.7 Other SimXL Limitations and Temporary Known Limitations

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

- *Other Limitations That Do Not Include Signals or Subroutines*
- *Temporary Known SimXL Limitations*

5.7.1 Other Limitations That Do Not Include Signals or Subroutines

See the following SimXL 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
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 $dumports in SW referencing HW hierarchies**
- **No support for -simprofile option in VCS compile**
- **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
   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
```

- **$stop is not supported**
- **$test$plusargs, $value$plusargs are not supported**

```
integer i1;
initial begin
   if ($test$plusargs("HELLO"))
      $display("Hello argument found.");
   if ($value$plusargs("TEST=%d", i1))
      $display("value was %d", i1);
end
```

- **Usage of Clock Delay Ports in SimXL flow is not recommended.** Clock Delay Ports timescale by default is 1ps/1ps and cannot be changed. This does not work well with other timescales.
- **Tasks defined in HW top level cannot be accessed like transactor tasks using XMRs from SW.** It is recommended to instantiate an interface in HW top level that contains the task functionality.

5.7.2 Temporary Known SimXL Limitations

Following are known limitations with the current release of SimXL:

- No support for SVA control residing in HW.
- No support for SVA in interfaces that have subroutines used to communicate between SW and HW.
- Usage of DSP Xilinx components is not recommended due to timing issues. To disable DSP components, use the following UTF command:

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

- **zPRD** is not supported in SimXL. To enable debug for SimXL, use the following UTF command:

  ```
  debug -verdi_db true
  ```

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

5.8 Supported and Unsupported Behavioral Compiler Syntax

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

- *Temporary Compile-Time Options*
- *Supported Behavioral Compiler Syntax in SimXL*
- *Not Supported Behavioral Compiler Syntax in SimXL*

5.8.1 Temporary Compile-Time Options

The following VCS compile-time options are required to enable SimXL's Behavioral Compiler's functionality:

```
-Xhwcosim=halt
```

```
-Xhwcosim=blocking_event
```

5.8.2 Supported Behavioral Compiler Syntax in SimXL

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*
- *System Tasks*
- *SystemVerilog Strings, $sformatf, %m are Supported*
- *Enabling concurrent export task/function calls*
- *Specifying a Module Not to be Processed by Behavioral Compiler That is Instantiated in a Transactor*
- *Number of Threads Used to Manage Transactors at Runtime*
- *Usage of Memory in Behavioral Code*

5.8.2.1 # 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
```

5.8.2.2 Clocking Edge in for Loops

See the following example:

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

5.8.2.3 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
```

5.8.2.4 Complex Edge Combination

See the following example:

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

5.8.2.5 Combinations of Edge and Level Events

See the following example:

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

5.8.2.6 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.

5.8.2.7 Unbounded Loops

See the following example:

```
initial
while (1) begin
```

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

5.8.2.8 Wait Statements

To enable correct wait behavior, add the following temporary VCS compile option: `-Xhwcosim=insert_delay`.

See the following example:

```
wait(clk);
```

5.8.2.9 Named Events

See the following example:

```
event my_event;
initial -> my_event;
always begin
    @(my_event);
    a <= b;
end
```

5.8.2.10 Behavioral Compiler Supports all SystemVerilog Data Types

Behavioral Compiler supports all SystemVerilog data types specified above.

5.8.2.11 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
```

5.8.2.12 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 temporary VCS compile option: `-Xmshroff5=0x800`.

Example

```
fork
   do_A();
```

```
    do_B();
join
```

5.8.2.13 System Tasks

The following system tasks are supported:

- `$display`, together with: `$displayh`, `$displayb`, `$displayo`
- `$fdisplay`, together with: `$fdisplayh`, `$fdisplayb`, `$fdisplayo`
- `$write`, together with: `$writeh`, `$writeb`, `$writeo`
- `$fwrite`, together with: `$fwriteh`, `$fwriteb`, `$fwriteo`
- `$fopen`
- `$fclose`

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

SimXL 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, add the following option to VCS compile that disables the SimXL implementation:

```
-Xhwcosim=disable_systasks_utf
```

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.*

5.8.2.14 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.

- `-Xhwcosim=<systasks or the corresponding entry in the UTF file.>`
- `-zebu_string=<MAX_STRING_LENGTH>`

If `-zebu_string` is not provided, SimXL reports an error stating that the `-zebu_string` should be enabled.

5.8.2.15 Enabling concurrent export task/function calls

Currently SIMXL allows only one call to a TB->DUT TF (task/function) at any given time. If multiple calls to the same TF are made at the same time, it results in an error, and stops SIMXL execution.

SIMXL adds instrumentation for a TF to both the TB side and the DUT side, and both need to be enhanced to support concurrent calls. Given the inherent difficulty of adding instrumentation on the DUT side in a way that it can be synthesized, the support will have a limitation, at least to start with: The user will provide an input to the tool to indicate which TFs can be called in parallel, and how many instances can be active at any given time.

Let us say that a maximum of N calls to a TF may be active in parallel. SimXL will create N copies/instances of the TF with unique idb-Ids corresponding to each of these calls. The HW module will have N copies of the auxiliary DS for the TF, referring to these idb-Ids (it is okay to have only one copy of the actual function, since it will be inlined). The TB side will allocate one of the free idb-Ids corresponding to this TF to the requesting call. If all the ids are busy, an error will be generated. Once the call is completed, the instance is freed for use by another call.

Use-Model

V2Z Mode (UTF Option):

```
simxl_allow_concurrent -module <module name> -tf {<TF list>} [-count
<n>]
```

Examples

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

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

V2VX Mode (VCS option):

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

Examples (equivalent to above mentioned UTF commands)

```
-Xhwcosim=concurrent+subdut+dutTaskOne
```

```
-Xhwcosim=concurrent+sub2+T1+2 -Xhwcosim=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.

5.8.2.16 Specifying a Module Not to be Processed by Behavioral Compiler That is Instantiated in a Transactor

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

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
```

5.8.2.17 Number of Threads Used to Manage Transactors at Runtime

It is recommended to set the environment variable `ZEMI3_XTOR_NTHREADS` to `16`. The engine that manages the communication on the host with the transactor allocates a thread per transactor. Maximum number of threads that can be allocated is 256. Thread context switching can overload the host and significantly reduce performance.

The `ZEMI3_XTOR_NTHREADS` environment variable will be set automatically in the future to a default value.

5.8.2.18 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.

5.8.3 Not Supported Behavioral Compiler Syntax in SimXL

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

- SystemVerilog semaphores are not supported
- SystemVerilog mailboxes are not supported
- 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 messages are not supported in subroutines mapped to HW.
- Usage of UVM config_db to assign physical interfaces to virtual interfaces is not supported in HW.

5.9 SystemC Support

The `-sysc=show_sc_main` option is automatically enabled for the SystemC top designs and the user is just required to pass the DUT path that starts with "sc_main"

("sc_main" being for the top of the design).

On 2018.09.ZEBU, 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 SimXL flow.

5.10 Scheduling Semantics and Race Conditions

The SimXL 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

See the following subsections:

- *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*

5.10.1 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.

5.10.2 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.

5.10.3 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:

   `-Xhwcosim=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*.

5.10.4 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 SimXL 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), SimXL 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.

5.11 Debug Facilities for Communication Mechanisms

To enable logging of tasks and function call between SW and HW, use the following compile-time option:

```
-Xhwcosim=enable_log_tf
```

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

It is recommended to use these facilities before going to debug through waveform signals. This provides a high-level view of the interactions going on between HW and SW.

TABLE 2 Options for Logging the Communication Transactions Between SW and HW

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

5.12 Functional Debug in SimXL

SimXL 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 SimXL debug-related information:

- *Enabling UCLI Signal Access for ZeBu Compile*
- *Enabling UCLI Waveform Output for ZeBu Compile*
- *Supported UCLI commands in SimXL*
- *Verdi Interactive Debug for SimXL*
- *Waveform Outputting Using UCLI*

5.12.1 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.

5.12.2 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
```

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

5.12.3 Supported UCLI commands in SimXL

See the following subsections:

- *get*
- *memory*
- *stop*
- *run*
- *force, release, show*
- *restart*
- *System Information*
- *dump Commands*

5.12.3.1 get

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.

5.12.3.2 memory

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.

5.12.3.3 stop

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*.

5.12.3.4 run

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

Example

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

5.12.3.5 force, release, show

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

5.12.3.6 restart

Restart the simulation at time zero.

```
restart
```

5.12.3.7 System Information

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

TABLE 3 UCLI Commands to Retrieve System Information

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

5.12.3.8 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.

- `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 `$dumpvar` commands together with block labels.

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

  Enables and disables outputting before starting or continuing simulation.

- `dump -close <FID>`

  Closes the file associated with the specified FID.

- `dump -flush <FID>`

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

5.12.4 Verdi Interactive Debug for SimXL

Verdi interactive debug supports the SimXL flow. With SimXL interactive debug, you can debug both software (testbench) and hardware (DUT) in the Verdi interactive debug environment. SimXL 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***.

5.12.5 Waveform Outputting Using UCLI

- SimXL 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 FSDB 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 SimXL*

5.12.5.1 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 big design.
- UCLI Tcl outputting example:

```
set v_fid [dump -file  rwc_wave.ztdb -type dynamic_probe]
dump -add hw_top.ucli_tester -depth 0 -fid $v_fid
dump -enable -fid $v_fid
run 10us
dump -fid $v_fid -flush dump -fid $v_fid -close
```

5.12.5.2 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
```

5.12.5.3 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 zRun GUI. For details, see the ***ZeBu Server User Guide***. SimXL 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.ztdb -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 -iCSA --input <your dumped ztdb directory> --zebu.work
<your zebu.work directory>
```

5.12.5.4 Multi-ZTDB Output

SimXL 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
```

5.12.5.5 Using Verdi for Debug

- *Verdi Interactive Debug for SimXL* 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:

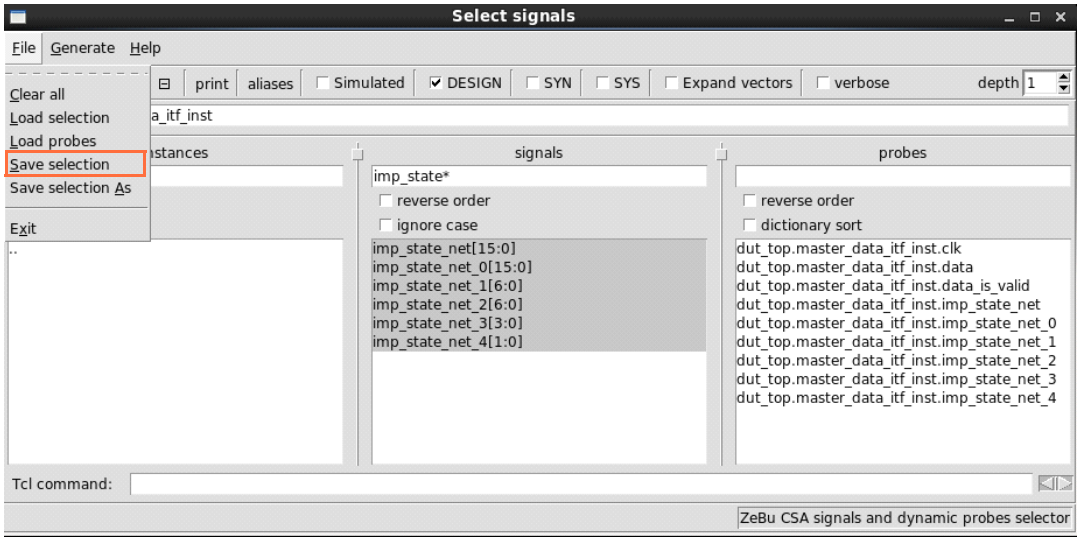
  ```
  verdi -nologo -emulation --zebu.work <ZEBU_WORK> --input
  <ZTDB_FILE> <other verdi command line options>
  ```

This opens Verdi to be run with the iCSA engine.

5.12.5.6 Output ZeBu XTOR Internal Signals

When using the subroutine-based SimXL 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 SimXL `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 `zRun`.

5.12.5.7 Waveform Expansion in SimXL

After the ZTDB file is outputted, you can use the ZeBu Waveform Expansion utility,

zWaveform, to get the FSDB file for debugging with Verdi. For more information, see the ***ZeBu Server 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.

```
%zWaveform --work zebu.work --ztdb myqiwc.ztdb --fsdb 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 zWaveform command is as follows:

```
%zWaveform --capture-only --work zebu.work --fsdb ….
```

5.13 SVA Assertion and Coverage Support

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
```

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
```

5.14 Profiler

As you begin to achieve runtime functionality with a SimXL V2Z environment, 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.

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

SimXL 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 SimXL 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.

The report comprises the following sections:

- Header
  - Tool Version
  - Start/End time
  - Clock frequency and edge count
- Time consumption
  - SimXL Elapsed time
  - Communication time (Data Preparation + Synchronization + Coemulation)
  - Testbench time
- SW/HW Synchronization
  - Number of Timesteps requiring synchronization
  - Number of synchronization events
- Data transferred and event counts
  - Each type of signal/memory
  - Each transactor
- Details of data transfer (for individual signals, memories, tasks and functions)

- Histogram and Expensive time steps

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 can be more apparent in the translog report.

For more information, see the following subsections:

- *Enabling Profiler*
- *Example Report*

5.14.1 Enabling Profiler

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

TABLE 4 Runtime Options to Enable Profiler

Runtime option	Description
`-simxl=profile`	Enables Profiler
`-simxl=profile,profile_log:myrep.txt`	Renames the report generated. By default, the report is generated in a file called `simxlProfile.txt`.

TABLE 4 Runtime Options to Enable Profiler

Runtime option	Description
`-simxl=profile,nheavy:<n>`	Specifies the number of expensive steps. By default, 10 expensive steps are printed. Max value of `<n>` is 100.
`-simxl=profile,profile_verbosity:<value>`	[a]Verbosity control

a.This option controls the verbosity of the profiler report. The default is 0. When the default is used, the following information is not printed:
- Number of simulation time steps in HW streaming mode
- Number of synchronization events in HW streaming mode
- Number of coemulation synchronous events
- Number of synchronization events for SW time advance only
- Average number of SW/HW synchronization events per time step

Note that multiple options require only one specification of -simxl=profile.

There are no compile-time options needed.

5.14.2 Example Report

In the following example report, the following applies:

- The numbers in tables above are representative and not necessarily related.
- Information printed in verbose mode is highlighted in blue text.
- Comments added to the report are in orange text.

Example Report

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


+-----------------------------------------------------------+
```

```
|  VCS Version          | <Version>
|  VCS Build Date       | Build Date = Aug 16 2018 07:22:59
+-----------------------------------------------------------+
|  SimXL Start Time     | <Start date and time>
|  SimXL End Time       | <End date and time>
|  Mode                 | Zebu
|  VCS Simulation Time | 200
+-----------------------------------------------------------+
|  ZEBU Version            | O-2019.06-Alpha
|  ZRDB Id                 | 55124962 [Only printed in verbose mode]
|  Zebu work path          | [full path: zebu.work]
|  Driver Clock Frequency | 6250 kHz
|  Driver Clock Cycles     | 1124948
|  Timestamp Clock Cycles | 20
+-----------------------------------------------------------+


Summary:
--------

Elapsed time
------------
    %time is the percentage of Elapsed Time.
    The breakup of Communication Time indicates % of Communication Time/% of Elapsed Time.
+==============================+==============+============+
|                              |  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  |
+==============================+==============+

Synchronization events
----------------------
Number of simulation time steps requiring synchronization: 20
Number of SW/HW synchronization events: 53
Number of synchronization events for SW time advance only: 0

Average time spent per synchronization event: 0.013 ms
Average number of time steps per second: 4.98

Number of simulation time steps in HW time streaming mode: 0
Number of synchronization events in HW time streaming mode: 0
Number of coEmulation sync events: 0
Average number of SW/HW synchronization events per time step: 2.650000
```

```
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:
|  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


Details of Data Transfer:
-------------------------

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

Details of Task/Function communication:
NOTE: For internal communication:
      HW->SW calls have 32 bits added to Input size
```

```
      SW->HW calls have 32 bits added to Output size
     Class and Virtual interface method calls (indicated with '*') have 64 bits added to Input
size
Instance Name: 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

Instance Name:
| Total Calls| Direction| Input Size | Output Size | Task ID |    Task/Function name    |
|------------|----------|------------|-------------|---------|--------------------------|


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


Details of Data Transfer for SimXL infrastructure:
--------------------------------------------------

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 |

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 |

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 |

Details of Control communication:
|     Function Name     |  Type  | Size | Call count |
|-----------------------|--------|------|------------|
|    cosimCtrlOSync     | HW->SW | 32 b |         21 |
|    cosimCtrlISync     | SW->HW | 32 b |         32 |

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 |


Histogram: [only for V2Z mode]
----------
| Sync Calls | Time Steps |
|------------|------------|
|         >8 |         14 |
|          6 |          8 |
|          3 |         21 |
|          1 |          1 |

Most expensive Time steps:
--------------------------
```

```
[Sorted by Elapsed time in V2Z, by no of calls (i/p + o/p + tf, sync) in V2VX]
[Elapsed time is available only in V2Z mode]
|   Time step   |Synchronization|     Input     |     Output    | Task/Function | Elapsed |
|               |     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|
====== End SimXL Communication Profiler Report ======
```

5.15 Incremental Compile in SimXL

Incremental compilation is implemented in the SimXL flow. With incremental compilation, SimXL compilation detects if the VCS and ZeBu interface database in the current compilation is a subset of the database in a previous SimXL compilation. If it is a subset, SimXL compilation skips the ZeBu compilation. Some cases where incremental compilation is useful are as follows:

- If there are changes that are local to the SW testbench that do not access any HW capabilities, there is no need to recompile HW.
- If XMRs used in SW to HW are removed from SW, there is no need for a recompile of HW as all the other needed accesses are still there.

To enable incremental compilation, specify the following VCS compilation-time option:

```
-Xhwcosim=incr_comp
```

You can also force testbench-only compilation through the UTF command if you are sure there is no change in SW accesses to HW and there is no change in the HW. The UTF command is as follows:

```
simxl_tb_compile_only -enable true
```

To enable incremental compile diagnostics, specify the following VCS compilation-time option:

```
-Xhwcosim=incr_diag
```

SimXL compilation generates a file called **IncrDiag.txt** that contains information about the changes to the SW or HW accesses.

5.16 Timing Analysis

SimXL 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, SimXL 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 to View Critical Output Routing Data Paths*
- *Analyzing a Critical Path Using zTime*

5.16.1 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
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.

5.16.2 Viewing the zTime Report to View 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 `<zcui.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 brownser, such as Mozilla FireFox. For example:

   ```
   file:///home/projects/zcui.work/backend_default/zTime_fpga.html
   ```

   The **zTime Report: Index** page appears.


zTime Report: Index

Date: Tue Jan 15 23:17:32 2019

Filter paths

Routing data paths

2. 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.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

The following table provides a description of the columns in the **zTime Report: Critical Output Routing Data Paths** report.

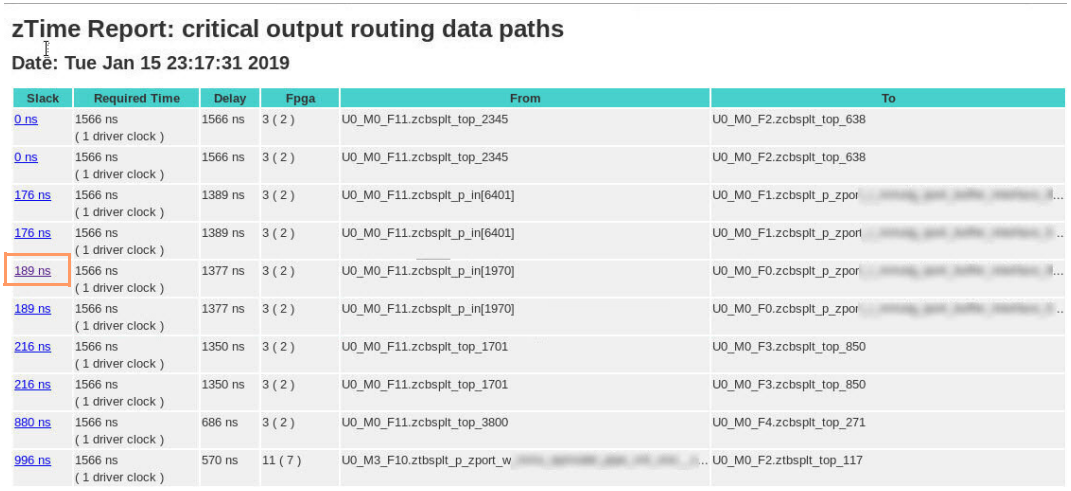
TABLE 5 Column Description in zTime Report: Critical Output Routing Data Paths

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.

5.16.3 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.

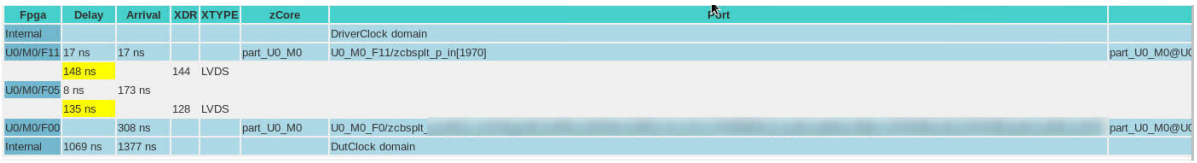


zTime Report: critical output routing data paths

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

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]	part_U0_M0@U0
	148 ns		144	LVDS			
U0/M0/F05	8 ns	173 ns					
	135 ns		128	LVDS			
U0/M0/F00		308 ns			part_U0_M0	U0_M0_F0/zcbsplt_...	part_U0_M0@U0
Internal	1069 ns	1377 ns				DutClock domain	

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 6 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.