ZeBu® Debug Guide

Version V-2024.03-1, July 2024

SYNOPSYS®

Feedback

Contents

Preface

This chapter has the following sections:

About This Book

The *ZeBu® Debug Guide* describes the ZeBu Server debug features and technologies to emulate your design with ZeBu Server.

Contents of This Book

The *ZeBu® Debug Guide* has the following chapters:

Chapter	Describes...
Introduction to Debug Flow	Provides an overview of the debug flow.
Waveform Capture and Expansion	Describes waveform capturing mechanisms:Dynamic-ProbesFWCQiWC
Recording and Replaying Stimuli (Snapshot)	Describes the usage to record and replay the Stimuli sent to the design during the emulation.
Capturing Waveforms Using $dumpvars System Task	Provides compilation tips.
Using Static Trigger, Dynamic Trigger, and Runtime Trigger	Describes the usage of dynamic trigger and Runtime Trigger in the debug flow.
Debug-Related Limitations	Lists the limitations of debug feature.

Feedback

Related Documentation

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

Typographical Conventions

This document uses the following typographical conventions:

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

The following table describes the syntax used in this document:

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

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

Feedback

1

Introduction to Debug Flow

The ZeBu environment provides several technologies to debug designs depending on your requirements. Therefore, debug planning is recommended to choose a debug technology or a combination of the debug technologies based on your requirements. For example, ZeBu provides the following waveform capture mechanisms, which have distinct capabilities:

- Fast Waveform Capture (FWC): Use for essential signals or instance ports
- FWC or Quick Waveform Capture (QiWC): Use for instances
- Dynamic-Probes: Use for full SoC

With any of these mechanisms, you can capture ZTDB waveforms and then use Verdi to view the ZTDB waveforms. The ZTDB format has been updated to include improvements for Post-Processing engines. For more information on methods to view ZTDB waveforms, see Waveform Viewing and Analysis.

This section describes the following subtopics:

- Debug Flow Overview
- Glossary of Debug-Related Terms

To use common ZeBu debug methodologies and for debug planning tips, see the *ZeBu Debug Methodology Guide*.

Debug Flow Overview

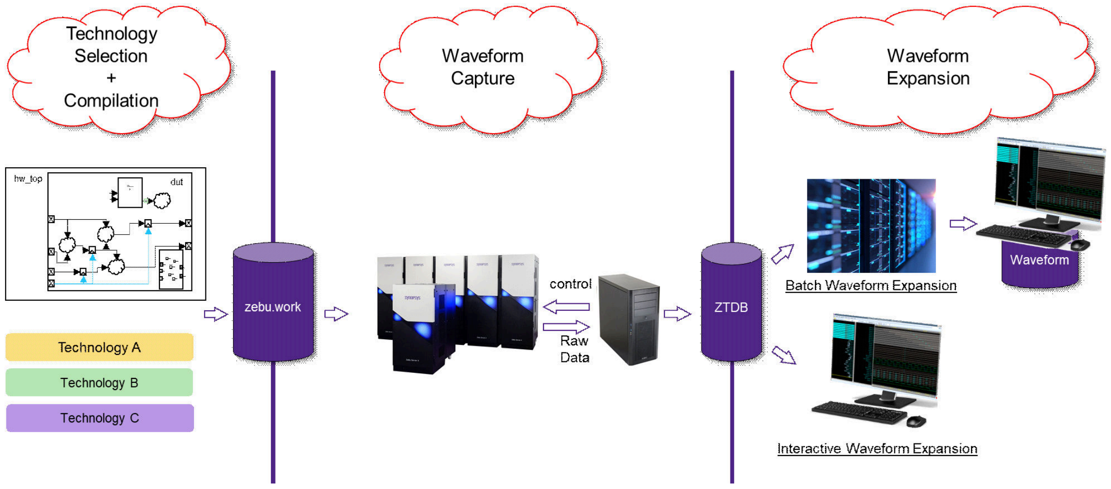
The flow for debug contains the following high-level steps:

1. **UTF Compilation**: Compile your design with appropriate UTF commands and optionally a list of signals to be captured (as described in Debug Setup for Compilation)
2. **Emulation Runtime**: Run emulation and capture ZTDB waveforms (as described in Runtime Control for Outputting ZTDB Waveforms)
3. **Waveform Analysis**: Reconstruct waveforms either interactively or in post processing (as described in Waveform Viewing and Analysis), and use Verdi to view them.

Feedback

The following figure shows the overall debug flow. The figure shows how the ZeBu debug technologies interact with each other.

Figure 1 ZeBu Debug Flow Overview



Glossary of Debug-Related Terms

Term	Definition
Composite Clock	Internal clock used to sample data on all the edges of all primary clocks
Support Signals	Signals such as flip-flop outputs required by the expansion engine to reconstruct the waveforms of combinational logic
Dynamic-Probes	Method to collect register data using Xilinx read-back mechanism
Essential Signals	Signals selected to perform a first level analysis
FSDB	Fast Signal Database – Verdi waveform format
KDB	Verdi Knowledge Database that contains design information
FWC	Fast Waveform Capture
QiWC	Quick Waveform Capture
SoC	System-On-Chip
ZTDB	Native ZeBu Database
ZWD	ZeBu Waveform Database - Viewable format

Feedback

2

Waveform Capture and Expansion

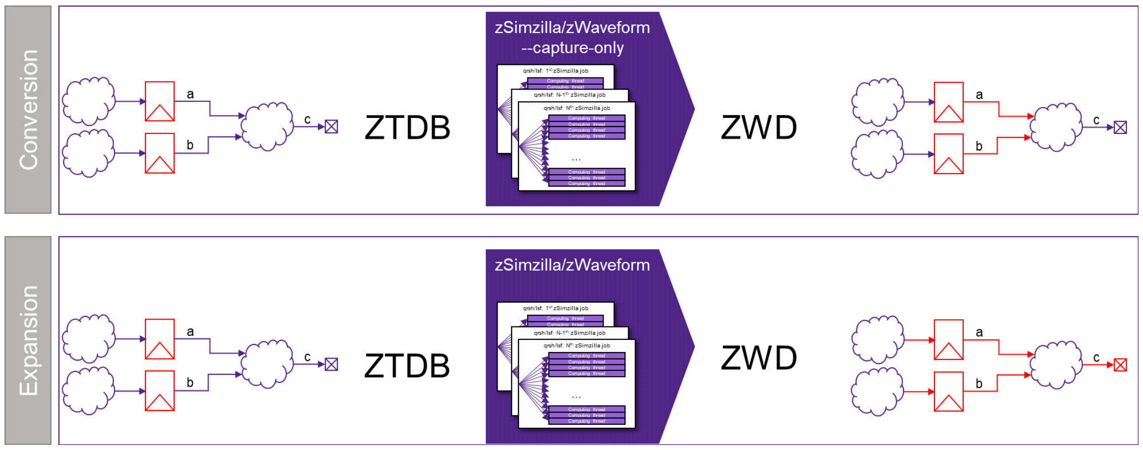
The Dynamic-Probes, FWC, and QiWC mechanisms share the same emulation flow. It consists of the following steps:

1. **Preparation**: Define essential signals and design instances to capture. For FWC and QiWC, these are specified using standard Verilog tasks
2. **Compilation: Compile your design with appropriate UTF commands**
3. **Runtime**: It consists of the following:
   - Writing a testbench
   - Running the emulation
   - Dumping the ZTDB waveforms
4. **PostRun**: There are three methods to view waveforms:
   - **Reconstruction:** `zSimzilla --capture-only`
   - **Expansion:**
     - **Batch mode:** `zSimzilla`
     - **Interactive mode:** `verdi`(running expansion engine)

The debug methodologies associated with waveform capture, expansion, and reconstruction are described in the *ZeBu Debug Methodology Guide*.

The following figure summarizes the waveform conversion and expansion methods:

Figure 2 *Waveform Viewing Methods*



Note:

In the above figure, the signals (highlighted in red) on the left correspond to ZTDB and the ZWD signals (highlighted in red) on the right can be viewed in Verdi waveform.

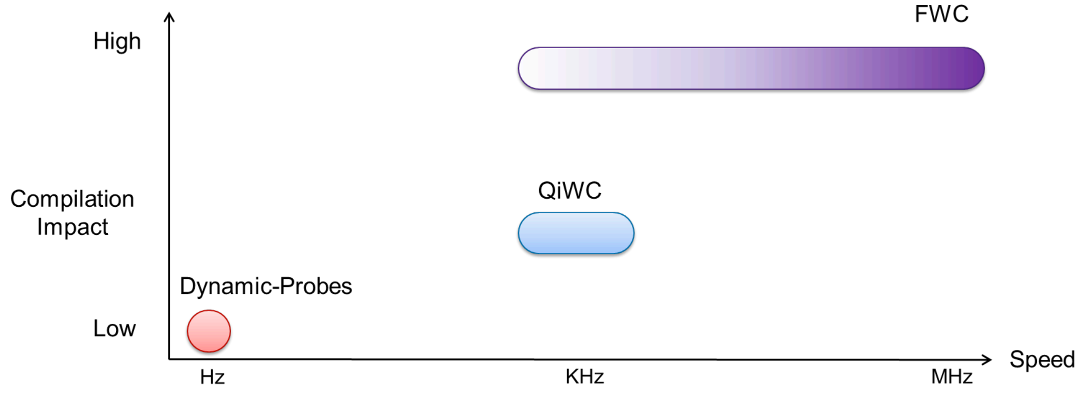
Waveform Capturing Mechanisms

There are three mechanisms for capturing waveforms during emulation runtime:

- Dynamic-Probes
- QiWC
- FWC

The following figure compares the three technologies to capture ZTDB waveforms:

Figure 3 Waveform Capture Technology X Compilation Impact



Dynamic-Probes

Dynamic-Probes use the Xilinx internal scan-chain to capture the design registers, that is:

- Any latch output
- Any register output

To capture combinational signals (non-registers), you must specify them at compilation. For more information, see Debug Setup for Compilation.

The emulation runtime speed while capturing dynamic-probes data is in the order of Hz.

QiWC

QiWC enables dumping design registers only; the combinational signals can be reconstructed using the expansion engine. QiWC allows runtime control over the dumping of any individual signal.

QiWC requires a few hardware resources. The emulation runtime speed while dumping is in the order of KHz.

FWC

FWC enables dumping at moderately high speeds of any signal: register and

combinational.

FWC requires significant hardware resources, which might impact capacity. Therefore, FWC is recommended for essential signals only.

Note:

Use this with caution on entire design instances.

The emulation runtime speed while dumping is in the order of MHz.

Note:

If too many signals are dumped with FWC, the emulation runtime speed can decrease to KHz.

Waveform Expansion Using zSimzilla

The **Simzilla** engine is scalable and its scalability resides in ZTDB slicing, which requires you to define an interval between each ZTDB slice when capturing ZTDB waveforms. Additionally, having threads running in parallel for each job allows large parallel simulations.

This section discusses the following subsections:

- Debug Setup for Compilation
- Runtime Control for Outputting ZTDB Waveforms
- Waveform Reconstruction and Expansion

Debug Setup for Compilation

This section describes the activities to compile your design with appropriate UTF commands. In addition, it describes how to capture a list of signals.

For more information, see the following sections:

- Common Compilation Setup for Debug: Describes UTF commands to generate the Verdi database and enable waveform expansion
- Compilation Setup for Dynamic-Probes: Describes the commands required to capture combinational signals and how to enable the dual-edge sampling for C-COSIM transactors
- Compilation Setup for Fast Waveform Capture (FWC): Describes how to specify Value-Sets for FWC

- Compilation Setup for Quick Waveform Capture (QiWC): Describes how to specify Value-Sets for QiWC
- Compilation Settings for Using Dynamic-Probes, FWC, and QiWC Simultaneously: Describes the settings required for using dynamic-probes, FWC, and QiWC simultaneously

Common Compilation Setup for Debug

The common compilation setup consists of:

- Generating the Verdi Database
- Enabling Waveform Expansion

Note:

After these steps, specific settings are required for FWC and QiWC.

Generating the Verdi Database

To view and analyze your design using Verdi, you must generate the Knowledge Database (KDB). Now Verdi KDB is automatically generated in in the ZeBu compilation directory. You can use the KDB to compile your design with ZeBu.

The Verdi KDB is generated in the following directory:

```
zcui.work/vcs_splitter/simv.daidir
```

Enabling Waveform Expansion

Waveform expansion is enabled by default for the Dynamic-Probe, FWC, and QiWC waveform capture mechanisms.

To disable waveform expansion, add the following UTF command in your UTF file:

```
debug -waveform_reconstruction false
```

Compilation Setup for Dynamic-Probes

Generally, dynamic-probes do not require a UTF command. However, if you use a C-COSIM transactor and you want to reconstruct waveforms with memories, you must enable the dual-edge sampling with the following command in your UTF file:

```
set_dualedge -instance {hw_top.top_ccosim}
```

For more information, see Waveform Viewing and Analysis.

By default, dynamic-probes are available only on the Support Signals.

Compilation Setup for Fast Waveform Capture (FWC)

This section describes the following subtopics:

- RTL Preparation for FWC
- Compilation
- VCS Compilation and Elaboration Script

RTL Preparation for FWC

The FWC mechanism requires the specification of Value-Sets. A Value-Set is a collection of objects such as:

- Signals
- Ports
- Instances

A Value-Set can be named through the label associated with its corresponding Verilog `initial` block.

All unnamed blocks are part of a Value-Set named `default`.

$dumpvars Task

The `$dumpvars` task is defined in the Verilog LRM; it designates at least two arguments:

- First argument: instance's depth (levels of hierarchy below instance)
- Succeeding arguments: signals, instance names, and signals and instance names

When the second parameter is an instance, the FWC mechanism targets only the Support Signals of the given instance.

`$dumpvars_suppress` excludes hierarchies from being captured. The commands take effect in the order in which they are executed.

Example

```
initial begin: ClockGen_wo_PLL
   $dumpvars (0, hw_top.dut.clk_gen.pll_core);
   $dumpvars_suppress (0,hw_top.dut.clk_gen.pll_core);
end
```

When specifying the VHDL path in `$dumpvars`, the design object name/VHDL path must be specified with double quotes. Therefore, the tool does not invoke costly XMR matching algorithm of VCS to find the object and it uses a specialized algorithm to match design objects in `$dumpvars`.

Example

```
(* fwc *) $dumpvars (1, "path.to.my.vhdl.signal");
(* fwc *) $dumpvars(1, "dut_tb.inst_ifx_pc_monitor0_if.pc_o");
```

$dumpports Task

The `$dumpports` task is defined in the Verilog LRM and can be used to designate a depth and followed by one or more instance names.

```
initial begin: DUT_wo_PLL
     $dumpports (hw_top.dut.core1);
     $dumpports (3, hw_top.dut.core1);
     $dumpports (2, hw_top.dut.clk_gen);
End
```

Syntax Rules

The Value-Sets are specified in a top-level SystemVerilog module, separate from the design. The arguments of the Verilog tasks also support wildcards and strings.

All these system tasks (`$dumpvars, $dumports...`) accept strings that might contain:

- *: matches any sequence
- *?*: matches one character

Wildcards do not match the hierarchy separator (.):

- Wildcard matches are anchored at a hierarchical level

  Example: `$dumpvars(1, "top.core*", "top.mem?.clk"`);

Recommendations:

- Limit the size of the targeted instances to avoid capacity issues.
- Define all the Value-Sets in one SystemVerilog module.

Default Behavior of $dumpvars and $dumpports

By default, these tasks designate the FWC mechanism.

Therefore, `$dumpvars(1, hw_top.top.core1.nirq);` is identical to: `(* fwc *) $dumpvars(1, hw_top.top.core1.nirq);`

Example

Here is an example of a SystemVerilog file that specifies two FWC Value-Sets:

SystemVerilog file: `DUT_FWC.sv`

```
module DUT_FWC();
  initial begin : Essential_Signals_NIRQ_HANDLER
    // Essential signal: nirq
    $dumpvars(1, hw_top.top.core1.nirq);
    // Essential signal: all signals for module
    (* fwc *) $dumpvars(1, "hw_top.top.core1.handler_l1.*",
                           "hw_top.top.core1.handler_l2.*");
  end
  initial begin : Essential_Ports_MEM_CORE1
    // Essential signal: all ports of memory
    (* fwc *) $dumpports(hw_top.top.core1.memory);

    // Essential signal: all ports of controller
    $dumpports(hw_top.top.core1.controller);
  end
endmodule
```

where:

- `DUT_FWC` is the name of the top-level module that defines all the Value-Sets.
- `Essential_Signals_NIRQ_HANDLER` and `Essential_Ports_MEM_CORE1` are the names of the Value-Sets.

Compilation

The compilation does not require any additional command in the UTF file.

Note:

If you reuse a UTF file that did not include FWC but did apply manual partitioning, the new compilation might fail. In that case, you need to update your partitioning commands.

VCS Compilation and Elaboration Script

The VCS compilation and elaboration script must specify analysis and elaboration of the additional top-level modules, such as `DUT_FWC` in the preceding example.

```
vlogan -sverilog DUT_FWC.sv
vlogan  …/…
vcs     -sverilog hw_top DUT_FWC
```

Where:

- `hw_top`: Specifies the top module of the design.
- `DUT_FWC`: Specifies the additional top-level module defined above.

Accessing Encrypted Signals at Runtime or with Waveform

Public encrypted signals can be viewed in Verdi while applying FWC and using ZWD waveform through the following UTF command:

```
debug -encryption_support true
```

This UTF command also allows you to access the encrypted registers made public while running emulation with the `get` UCLI command. This provides runtime access to encrypted signals.

Note:

Interactive waveform expansion is not supported for these encrypted signals.

While using encrypted signals with transactors and memory models, MuDb size might increase significantly if unconnected memories are present in the code (RTL encrypted or not).

Compilation Setup for Quick Waveform Capture (QiWC)

This section describes the following subtopics:

- RTL Preparation for QiWC
- Compilation
- VCS Compilation and Elaboration Script

RTL Preparation for QiWC

The QiWC mechanism requires the specification of Value-Sets. Each Value-Set is a collection of instances.

A Value-Set can be named through the label associated with its corresponding Verilog initial block.

$dumpvars Task

The `$dumpvars` task is defined in the Verilog LRM; it designates at least two arguments:

- First argument: instance's depth (levels of hierarchy below instance)
- Succeeding arguments: signal and/or instance names

The QiWC mechanism limits the `$dumpvars` task to instance names only, and it only targets the Support Signals of the given instances.

Note:

QiWC does not support the `$dumpports` task.

Example

Here is an example of a SystemVerilog file containing two QiWC Value-Sets:

SystemVerilog file: `DUT_QiWC.sv`

```
module DUT_QiWC();
  initial begin : CORE1_CORE2
    (* qiwc *)  $dumpvars(0, hw_top.top.core1);
    (* qiwc *)  $dumpvars(0, hw_top.top.core2);
  end
  initial begin : Essential_mem_controller
    (* qiwc *)  $dumpvars(1, hw_top.top.core3.memory);
    (* qiwc *)  $dumpvars(0, hw_top.top.core3.controller);
  end
endmodule
```

Where:

- `DUT_QiWC`: Specifies the module name for QiWC technology.
- `CORE1_CORE2` and `Essential_mem_controller`: Specifies the QiWC Value-Set.

The `$dumpvars` tasks are applied on instances with the specified depth.

- *1* for `hw_top.top.core3.memory`: Waveform expansion can be run only on "`hw_top.top.core3.memory`" and not on its instantiated modules.
- *0*for other hierarchies: The `$dumpvars` tasks are applied with unlimited depth. That is, waveform expansion can be done on all logic in their corresponding hierarchies.

Only the Support Signals are captured. The Expansion engine can reconstruct the waveform of the entire instance tree.

QiWC Support on Mixed-HDL

Use the following to enable QiWC on mixed-HDL:

```
(qiwc) $dumpvars(1, "zebu_rcu_top.INST1.DUT")
```

Note:

You must use the double quotes for mixed-HDL.

Compilation

The compilation does not require any additional command in the UTF file.

Note:

If you reuse a UTF file that did not include QiWC but did apply manual partitioning, the new compilation might fail. In that case, you need to update your partitioning commands.

VCS Compilation and Elaboration Script

The VCS compilation and elaboration script is the same as the one for the FWC technology:

```
vlogan -sverilog DUT_QiWC.sv
vlogan  …/…
vcs     -sverilog hw_top DUT_QiWC
```

Where:

- `hw_top`: Specifies the top module of the design.
- `DUT_QiWC`: Specifies the module that contains the Value-Sets specification.

Compilation Settings for Using Dynamic-Probes, FWC, and QiWC Simultaneously

The three waveform capture mechanisms can coexist and be used at the same time. The following example shows FWC and QiWC specified in the same top-level Verilog module.

Example

```
module DUT_Dump();
  initial begin : Essential_Signals_NIRQ_HANDLER
    $dumpvars(1, "hw_top.top.core1.handler_l1.*",
                  "hw_top.top.core1.handler_l2.*");
    $dumpvars(1, hw_top.top.core1.nirq);
  end
  initial begin : Essential_Ports_MEM_CORE1
    (* fwc *)   $dumpports(hw_top.top.core1.memory);
    (* fwc *)   $dumpports(hw_top.top.core1.controller);
  end
  initial begin : full_CORE1_CORE2
    (* qiwc *)  $dumpvars(0, hw_top.top.core1);
    (* qiwc *)  $dumpvars(0, hw_top.top.core2);
  end

  initial begin : full_MEMORY_CONTROLLER
    (* qiwc *)  $dumpvars(2, hw_top.top.core3.memory);
    (* qiwc *)  $dumpvars(0, hw_top.top.core3.controller);
  end
endmodule
```

Runtime Control for Outputting ZTDB Waveforms

During runtime, you can perform several activities, such as capturing raw data from ZeBu Server and saving the captured information in the ZTDB database with the ZeBu Runtime Control Interface (zRci). `zRci` provides a Tcl interface to interact with the emulation at runtime.

Note:

- When using the `--testbench` option with `zRci`, only one of the methods must activate waveform dumping, not all (`zRci` and testbench).
- Waveforms are always sampled on `tickClk`.

This section contains the following subsections:

- ZTDB Slicing
- Using the ZeBu Runtime Control Interface (zRci) for Debug
- Waveform Viewing and Analysis

For more information, see the *ZeBu User Guide*.

ZTDB Slicing

To reduce the time to capture waveform, the **zSimzilla** engine launches jobs in parallel in the compute farm.

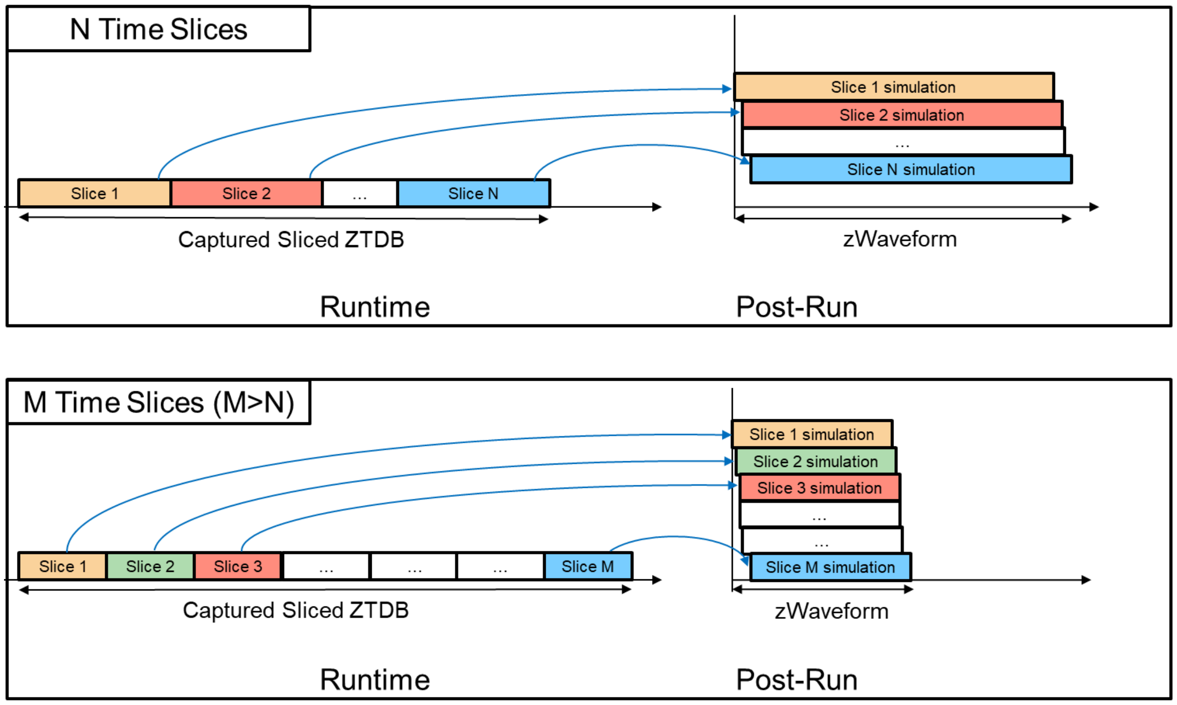
Each job consumes one or several ZTDB slices to reconstruct the combinational logic and write waveform data onto the disk.

zSimzilla reconstructs the waveforms based on the ZTDB slices. If more number of ZTDB slices are present, **zSimzilla** is faster.

For each capture technology, an interval can be defined to enable the ZTDB slicing and the time between each ZTDB slice.

The following figure reflects the impact of ZTDB slicing on the waveform reconstruction time.

Figure 4 ZTDB Slicing



Auto-Slicing Support for QiWC

The **Simzilla** engine parallelism is based on the number of ZTDB slices.

For best time to capture waveform, the number of ZTDB jobs running in parallel to the compute farm should be close to the number of ZTDB slices.

To automate the number of ZTDB slices to be received, use the `config` UCLI command.

Example

To run 50 jobs in parallel and capture 3 million samples, specify the following `config` command:

```
config waveform_capture_slicing {3000000total_samples,50slices}
```

For details, see the “**Config Commands”** section in the *ZeBu Unified Command-Line User Guide*.

Limitations

- The number of planned samples is limited to about 1 million.
- The number of planned slices is limited to about 250.

Aligning timescale with zSimzilla

When using the `zCeiClock` module to control the emulation runtime and capture a ZTDB, the waveform conversion with `zSimzilla --capture-only` requires you to use the `--timescale` option.

The `--align-timescale` option must be used simultaneously with the `--timescale` option.

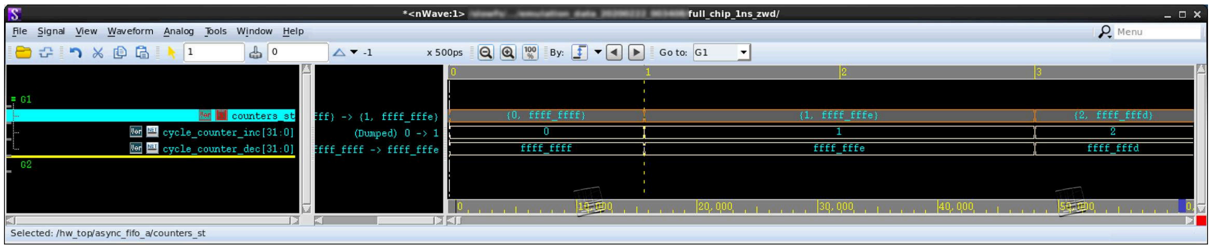
In this scenario, when using the `--align-timescale` option (without any parameter), the timescale is embedded with the value-change in the ZWD waveform format.

This is required for power estimation tools.

Example

The following figure shows the waveform capture when `--timescale` is 1ns (without `--align-timescale`).

Figure 5 Waveform Capture and Expansion using -timescale

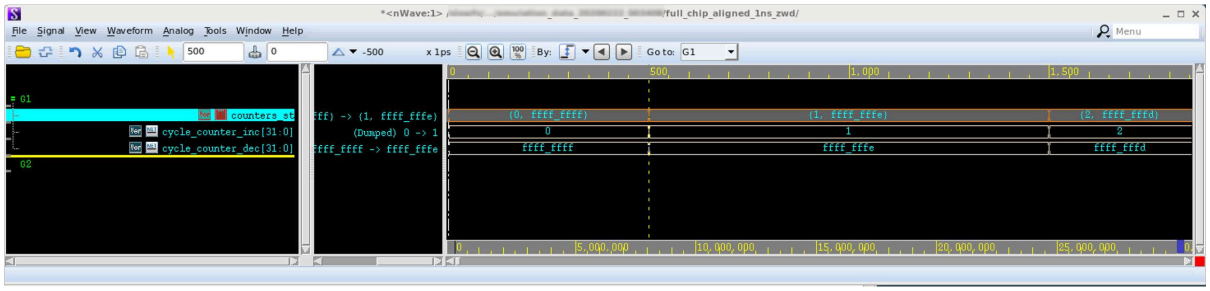


*<nWave:1> [redacted] full_chip_1ns_zwd/

Selected: /hw_top/async_fifo_a/counters_st

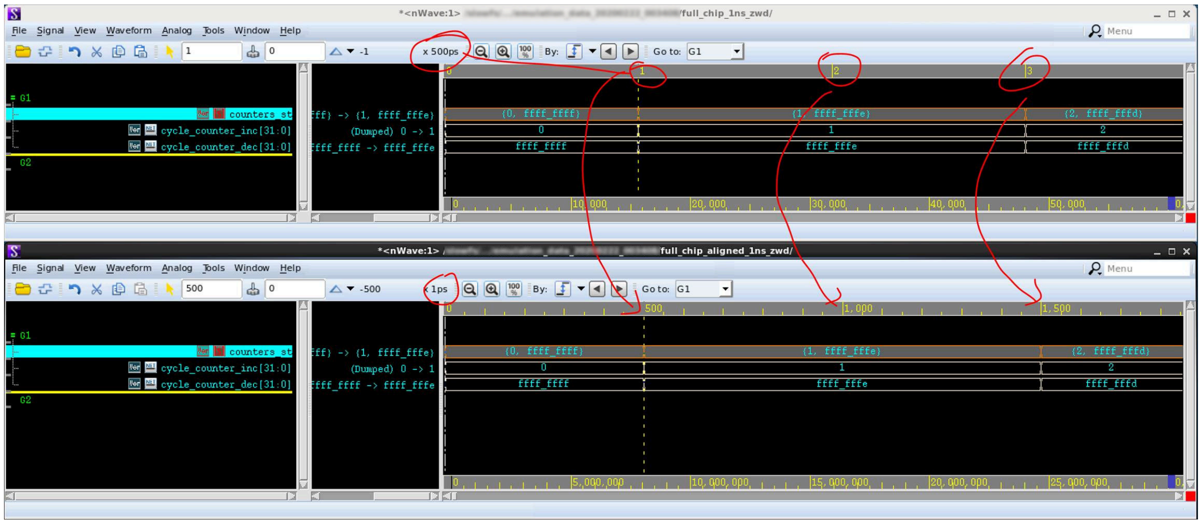
The following figure shows the waveform capture when `--timescale` is 1ns and `--align-timescale` is 500ps.

Figure 6 Waveform Capture and Expansion using -timescale and --align-timescale



The following figure depicts the comparison between the waveforms captured as highlighted 0 -> 0ps, 1 -> 500ps, 2 -> 1000ps, 3 -> 1500ps.

Figure 7 Waveform capture and expansion Using zSimzilla and zSimzilla --capture-only



Using the ZeBu Runtime Control Interface (zRci) for Debug

This section describes the following:

- Using zRci to Capture Waveforms for Dynamic-Probes
- Using zRci to Capture FWC and QiWC Waveforms

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

Using zRci to Capture Waveforms for Dynamic-Probes

To capture waveforms using Dynamic-Probes, use the two commands displayed in the following example:

```
#File Definition and Technology selection:
set fid [dump -file Dynamic_Probes.ztdb -dynamic]
#Specify the interval in seconds to enable
#ZTDB slicing for Waveform Reconstruction with zSimzilla:
dump -interval 200 -fid $fid

dump -enable -fid $fid
…
dump -disable -fid $fid
dump -flush -fid $fid
dump -close -fid $fid
```

Using zRci to Capture FWC and QiWC Waveforms

When using `zRci`, to capture waveforms, use the dump command. For more details, see the *ZeBu Unified Command-Line User Guide*.

The following code is an example for both QiWC and FWC technologies.

- For FWC, use the `-fwc` option
- For QiWC, use `-qiwc` option

```
#File Definition and Technology selection:
set fid [dump -file full_chip.ztdb -qiwc]

#QiWC value-set selection:
dump -add_value_set {full_chip_qiwc} -fid $fid
#Specify how to apply ZTDB Slicing mechanism for Waveform
 Reconstruction #with zSimzilla:
#ZTDB Slice every 4 seconds:
#dump -interval 4 -fid $fid
# or apply auto-slicing:
dump -interval 3000000total_samples,50slicess -fid $fid
#Start capture:
dump -enable  -fid $fid
…


#Stop capture and closing file
dump -disable -fid $fid
dump -flush   -fid $fid
dump -close   -fid $fid
```

If no value set is added before enabling FWC or QiWC capture, all relevant Value-Sets are automatically added and a warning is printed. If no Value-Sets are available, an error is reported.

Note:

Automatically adding Value-Set is only supported when the ZTDB is opened.

Waveform Viewing and Analysis

If the captured ZTDB contains raw data of a small set of signals or hierarchies, or if the ZTDB is captured on a small emulation runtime window (that is, less than 300 thousands samples), you can use Direct Viewing method. For more information, see Waveform Reconstruction.

If the waveforms are generated with Value-Sets that specify design instances, waveform expansion with **zSimzilla engine** tool is required. For more information, see Waveform Expansion.

It is recommended to use interactive waveform reconstruction to reconstruct the waveforms within the Verdi GUI when you display a signal's waveform of approximately 100,000 cycles. Otherwise, you can use **zSimzilla** to reconstruct all waveforms.

For more information about reconstructing the waveforms using Verdi GUI, see the *ZeBu-Verdi Integration Guide*.

Note:

Use the following APIs for controlling the emulation runtime with C/C++ testbench:

- `WaveFile`: Used to control the Dynamic-Probe ZTDB capture
- `FastWaveformCapture`: Used to control FWC and QiWC ZTDB capture

Waveform Reconstruction and Expansion

This sections provides information on the following:

- Waveform Reconstruction
- Waveform Expansion

Waveform Reconstruction

Emulation runtime generates ZTDB waveforms, but **nWave** operates on ZWD waveforms. Therefore, before viewing, the waveform must be generated:

1. To launch **zSimzilla, perform the following:**

```
zSimzilla --capture-only\
--ztdb Key_Signals.ztdb \
--zebu-work path_to/zcui.work/zebu.work \
--zwd Key_Signals \
--timescale 1ps \
--command "<remote command>" \
--jobs <User entered specified values, ex: 40> \
--log zSimzilla_capture_only.log
```

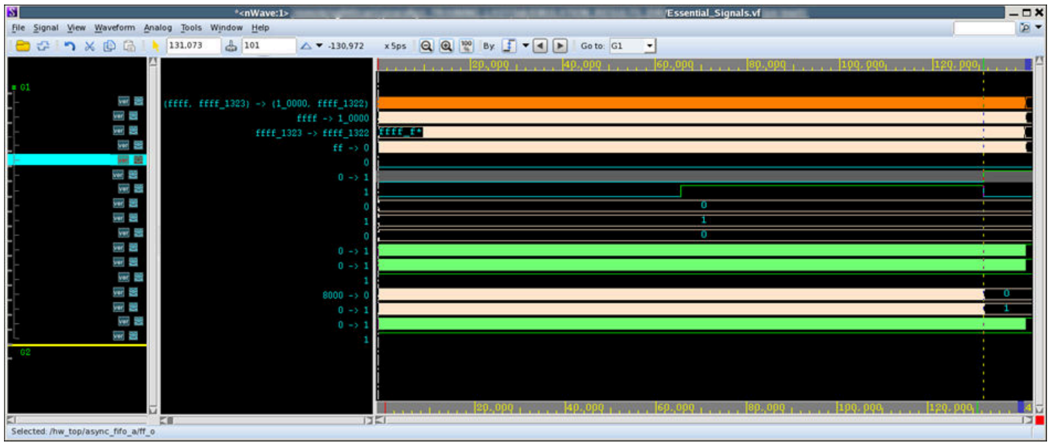
For best performance, use **zSimzilla** on a sliced ZTDB and use the `--command` option with the `qrsh` or `lsf` commands to use a compute farm.

For more details, use the `-h` option of `zSimzilla`.

1. Launch **nWave** to view the waveform:

```
nWave -ssf $(RUNDIR)/Key_Signals
```

Figure 8 nWave Interface



Waveform Expansion

There are two modes to view waveforms:

- **Batch mode with** `zSimzilla`

  `zSimzilla` is a tool to run waveform expansion/reconstruction in batch mode.

The waveform expansion consists of expanding the capture FPGA signals from the ZTDB to DUT signals.

It is available on dynamic-probes, FWC, and QiWC captured ZTDBs. It is applicable on all the DUT signals (Registers, Latches, Combinational signals, and so on) or a subset of the DUT in the ZWD/FSDB waveform.

Its scalability resides in ZTDB slicing, which requires you to define an interval between each ZTDB slice when capturing ZTDB waveforms.

`zRci` automatically launches waveform expansion (`zSimzilla`) as soon as a ZTDB slice is created by the host PC. For details, see the *ZeBu Unified Command-Line User Guide*.

- Interactive mode with `verdi`

  Interactive waveform expansion within Verdi allows you to reconstruct waveforms on-the-fly through drag and drop operations.

  For more information about interactive waveform reconstruction, see the *ZeBu-Verdi Integration Guide*.

This section describes the following subsections:

- zSimzilla
- Example
- Running Interactive Waveform Expansion

zSimzilla

The **zSimzilla** launches jobs on the compute farm using the `--jobs` option.

Each job runs using multiple CPU cores. All the jobs launched by **zSimzilla** reconstruct a combinational logic in parallel and their number is defined using the `--threads` option.

You need to align the compute farm command with the number of threads.

An example to specify the number of CPU cores is as follows:

- LSF: `-n <Number of Cores>`
- QRSH: `-pe mt <Number of Cores>`

If the number of jobs (limited by the number of ZTDB slices) and threads are more, **zSimzilla** is faster.

You can reconstruct a subpart of hierarchies and signals that needs to be included in the hierarchies captured in ZTDB.

You can then use the `zSimzilla` command as specified in the following example:

```
zSimzilla \
--ztdb captured_waveform.ztdb \
--zebu-work path_to/zcui.work/zebu.work \
--zwd my_waveform \
--timescale 1ps \
--command "<remote command specifying the number of CPU cores>" \
--threads <Number of CPU cores> \
--jobs <User entered specified values, ex: 40> \
--log my_waveform.log
```

Note:

`zSimzilla --help` can be used for viewing the options.

You can reconstruct a subpart of hierarchies and signals that needs to be included in the hierarchies captured in ZTDB.

For FWC and QiWC types of ZTDBs, only the signals or hierarchies selected with `$dumpvars` are honored by default and present in the waveforms.

To change the list of signals and hierarchies to be reconstructed, use the following `zSimzilla` option:

```
--zxf
```

It contains a list of `-i` or `-s` commands.

While specifying a hierarchy, if required, you can specify the depth with `-d` of the hierarchy to reconstruct and view in the waveform.

Example

- `-i <path> -d <depth>`: Path and depth of an instance to be computed
- `-s <path>`: Path of a signal to be computed

In the waveforms, simulation algorithm is added to resolve the X and NC propagation. Addition of this algorithm can impact the speed of waveform expansion.

The `--enable-xn-resolution` switch is added to the `zSimzilla` command to enable the X and NC resolution.

When **zSimzilla** finished the execution with some spawned jobs that are failed, you can relaunch only the failing jobs by adding the `--rerun-jobs` option to the original `zSimzilla` command.

Note:

- It is recommended to use the command line option as `--rerun-jobs`. If there is any change in the usage, `--rerun-jobs` options does not work.
- If the output waveform directory is deleted or moved, `--rerun-jobs` does not work because `--rerun-jobs`does not have access to the status of the previous run.

Running Interactive Waveform Expansion

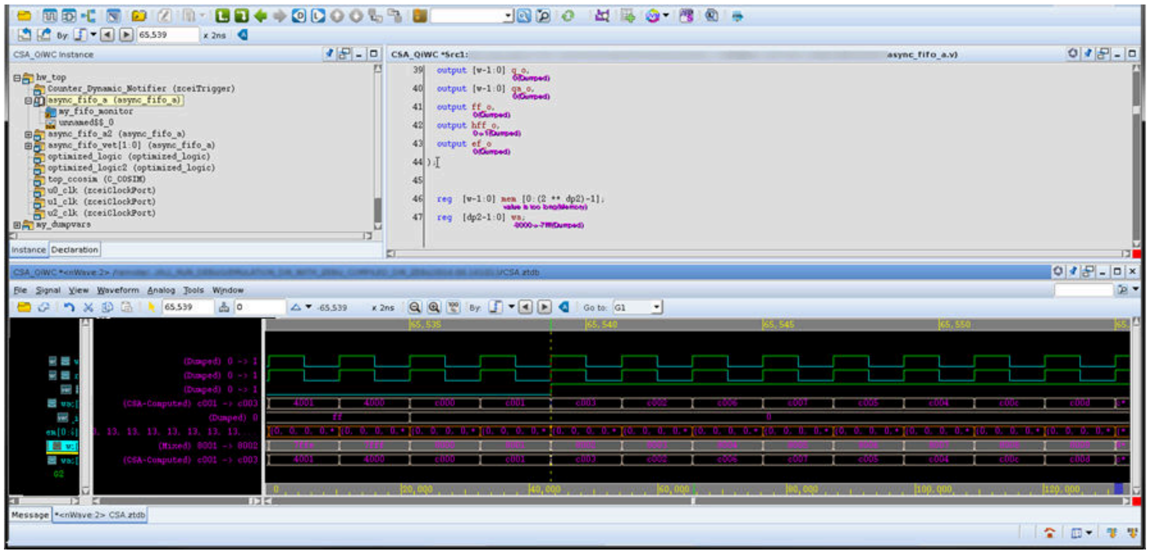
You can run Verdi and interactively reconstruct waveforms. The default command is as follows:

```
verdi -emulation \
-workMode hardwareDebug \
--input <path to the .ztdb waveform> \
--root <name of the hw_top> \
--zebu-work <path to zebu.work directory> \
--timescale 2ps
```

If required, you can also modify the command to specify the path to the KDB directory using `-dbdir` during compilation.

```
verdi -emulation \
 -simflow \
 -dbdir       <path to zcui.work/vcs_splitter/simv.daidir \
 -workMode    hardwareDebug \
 --root       <name of the top> \
 --input      <path to the .ztdb waveform> \
 --zebu-work  <path to zebu.work directory> \
 --timescale  2ps \
```

Figure 9 Interactive Waveform Expansion With Verdi



```
hw_top
  Counter_Dynamic_Notifier (zceiTrigger)
  async_fifo_a (async_fifo_a)
    my_fifo_monitor
    unnamed$$_0
  async_fifo_a2 (async_fifo_a)
  async_fifo_vet[1:0] (async_fifo_a)
  optimized_logic (optimized_logic)
  optimized_logic2 (optimized_logic)
  top_ccosim (C_COSIM)
  u0_clk (zceiClockPort)
  u1_clk (zceiClockPort)
  u2_clk (zceiClockPort)
my_dumpvars
```

```
output [w-1:0] q_o,
output [w-1:0] qa_o,
output ff_o,
output hff_o,
output ef_o
);

reg  [w-1:0] mem [0:(2 ** dp2)-1];
reg  [dp2-1:0] wa;
```

Support for FSDB Stitch in zSimzilla

Stitch Mode FSDB waveforms can be generated using **zSimzilla** when the ZTDB is based on Readback/Dynamic-Probe, FWC or QiWC technologies.

Use the following UTF commands during compilation:

```
debug -waveform_reconstruction true
debug -waveform_reconstruction_params {partitions=auto, FSDB=true}
```

When the ZTDB waveform is captured, use the `--fsdb` option to run waveform expansion as follows:

```
zSimzilla --ztdb wave.ztdb --fsdb wave_fsdb ...
```

Open the waveform with Verdi using the `.vf` Virtual File as follows:

```
verdi -emulation --zebu-work -ssf wave_fsdb.vf
```

3

Recording and Replaying Stimuli (Snapshot)

Use the Stimuli Replay technology to record and replay the Stimuli sent to the design during the emulation. The Sniffer captures emulator states and records Stimuli in frames. Multiple frames can be captured at intervals that you have specified. This is useful when your original emulation run is applied on billions of cycles because fewer cycles are replayed to capture a ZTDB waveform.

Using the Stimuli Replay technology, you can perform the following:

- Save the design stimuli starting from an arbitrary time using **Sniffer**.
- Create snapshots of the ZeBu system at any time; These snapshots are called **Frames**.
- Restore any snapshot/frame and rerun using the snapshot/Frame as a starting point. The rerun uses the **Injector** technology.

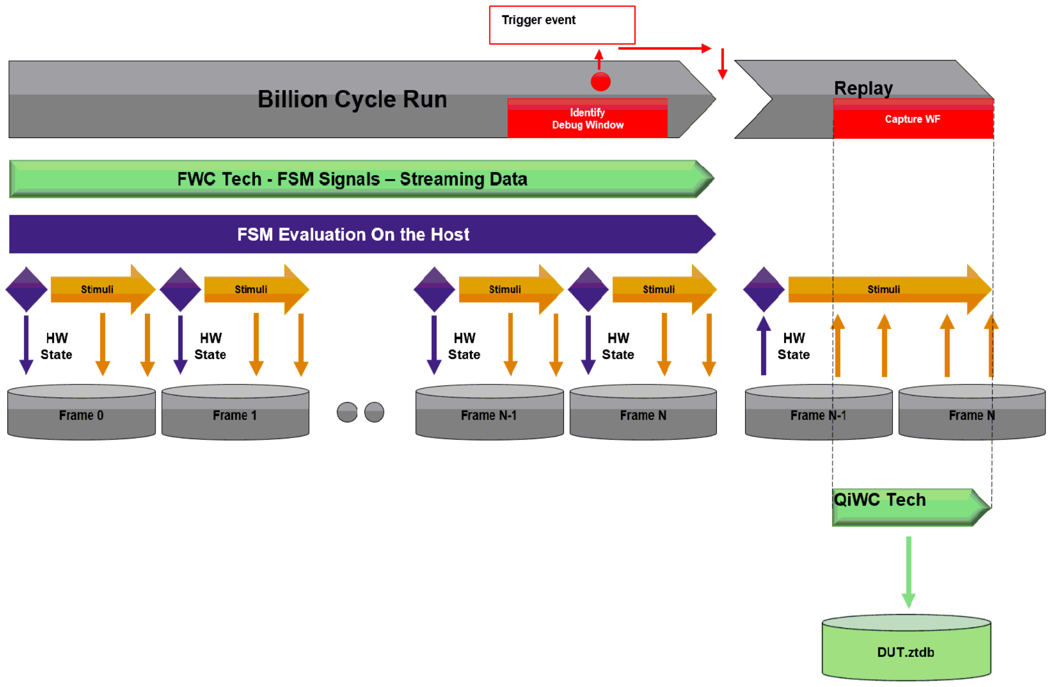
When rerunning the stimuli, the testbench driving the DUT is bypassed. You can enable any Streaming technology such as waveform capture, SVA, and zDPI (monitoring).

Before using the stimuli replay technology, a compilation setup is required. For more information, see the Compilation Setup for Stimuli Replay section.

During the first emulation runtime, you can use the `sniffer` UCLI command from `zRci`, or eventually the Sniffer C++ API in a different runtime environment.

The following figure illustrates the data generated by the **Sniffer during the Billion Cycle Run and the Injector during the Replay**. For more information, see the Initial Emulation Runtime for Sniffer section.

Figure 10 Global View During the Emulation Runtime



The Stimuli Capture Replay technology must be enabled at compile time. For more information, see the Replaying the Stimuli With the Injector section.

This section describes the following subtopics:

- Compilation Setup for Stimuli Replay
- Initial Emulation Runtime for Sniffer
- Replaying the Stimuli With the Injector
- New Engine for Stimuli Capture and Replay

The *ZeBu Debug Methodology Guide*describes the “*Stimuli Record With zDPI and Replay With Waveform”* methodology.

Compilation Setup for Stimuli Replay

The Stimuli Capture and Replay feature is enabled by default.

To disable this feature, add the following UTF command in your UTF file:

```
debug -offline_debug false
```

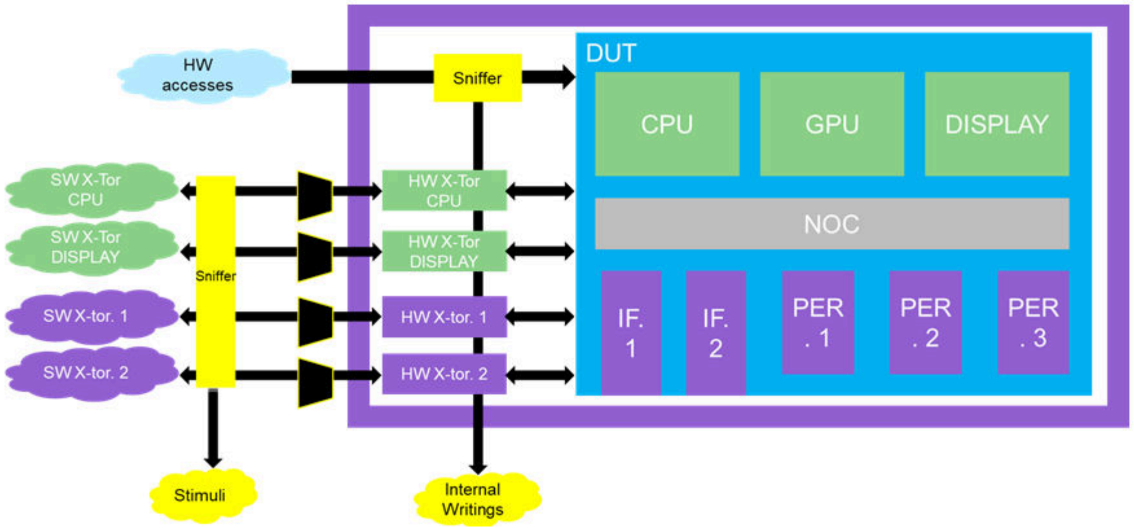
Where:

- `offline_debug`:(mandatory) Integrates the rerun capability

Initial Emulation Runtime for Sniffer

The Sniffer technology samples the stimuli on each clock edge of the clock groups containing the default UCLI clock as shown in the following figure.

Figure 11   Integrating Sniffer and Clock Connectivity



To control the Stimuli Capture and Replay feature, use the following `zRci` UCLI commands:

- `config`: Use to replay Stimuli recorded from a different runtime environment
- `sniffer`: Use to record state and save stimuli
- `replay`: Use to replay the stimuli and can be used only if a frame is restored

To set up the Sniffer configuration, use the following UCLI command:

```
sniffer -config [clock <clk_name>] [keep_frames <n>] [no_memory_copy]
```

where:

- `clock <clk_name>`: Sets the reference clock
- `keep_frames <n>`: Indicates the number of frames to be kept in the sniffer database
- `no_memory_copy`: Indicates no copy of memories while recording stimuli (by default all memories that are used are copied into sniffer folder)

For more details, see the *ZeBu Unified Command-Line User Guide*.

You can control the frames creation inside the UCLI code using the following command, if required:

```
sniffer -start_frame
```

You can automatically generate Sniffer frames periodically. The period can be defined in the following format:

- `Number of DUT clock cycles`: `sniffer -auto_create -cycle 1000000`
- `Time`: `sniffer -auto_create -runtime 123us`
- `In wall time`: `sniffer -auto_create 3600s`

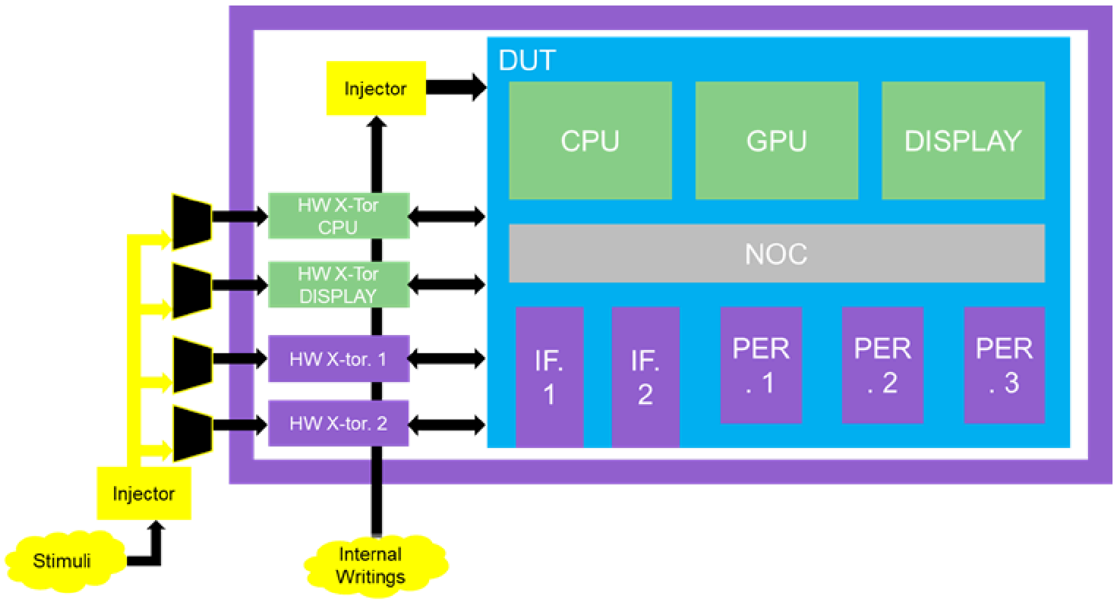
Note:

For C/C++ testbenches, Sniffer APIs are available to use the Sniffer technologies. For details, see the `ZEBU_Sniffer.h` and `Sniffer.hh` header files.

Replaying the Stimuli With the Injector

The following figure illustrates the steps to replay the stimuli with injector.

Figure 12 Replaying the Stimuli with the Injector



To set up the Sniffer configuration using the `replay -config` option, use the following UCLI command:

```
replay -config [reader <reader_name>] [tasks <n>]
```

where:

- `reader <smd||ztdb_threaded_scanner>`: Selects the appropriate reader for Replay. Default is `smd`.
- `tasks <n>`: Sets the maximum number of HW injectors for parallel injection.

The replay UCLI command of `zRci` allows you to control the injection of the Stimuli recorded in the Sniffer frames.

Prior to choosing which Sniffer frame to restore using the `sniffer -restore` UCLI command, you may need to identify the start and end cycle/time of all the Sniffer frames. Therefore, use the `sniffer -list` and `show -cycle` or `show -time` UCLI command. For details, see the following example.

During Replay, you can also perform the following:

- Capture the ZTDB waveform
- Run `CCall` commands (offline or on-the-fly)
- Read any registers and apply forces
- Read memories content and eventually update them
- Use the Runtime Trigger

Example

```
set start_of_dump_window [lindex $argv 1]
# zebu.work required for Replay
config zebu_work zcui.work/zebu.work
config default_clock   posedge hw_top.clk
replay -config -reader ztdb_threaded_scanner
config db_path  sniffer_w_zDpi_db
set replay_end 0
puts "Looking for the last frame containing cycle #$start_of_dump_window"
foreach frame [sniffer -list] {
   set res [show $frame -cycles]
   set start [lindex $res 0]
   set end [lindex $res 1]
   puts "$frame starts from $start and ends at $end"
   if {($start <= $start_of_dump_window) && ($end >=
 $start_of_dump_window)} {
       set replay_start $start
       set frame_to_restore $frame
   }
   if {  ($end >= $replay_end) } {
        set replay_end $end
    }
}

puts "Restoring: $frame_to_restore"
start_zebu -sniffer_restore $frame_to_restore
puts "$frame_to_restore restored"
puts "At cycle #[replay 0]"
replay [ expr {$start_of_dump_window - $replay_start} ]
puts "Starting Dump at cycle #[replay 0]"
set qiwc_fid [dump -file full_chip_at_replay.ztdb -qiwc]
dump -add_value_set {Full_Chip_VS} -fid $qiwc_fid
dump -interval 1 -fid $qiwc_fid
dump -enable -fid $qiwc_fid
replay 1010
puts "Closing Dump at cycle #[replay 0]"
dump -close -fid $qiwc_fid
```

For more details, see the *ZeBu Unified Command-Line User Guide.*

New Engine for Stimuli Capture and Replay

The Stimuli capture and replay technology is now applicable at the level of message ports of transactors. Since the messages are replayed, the transactor is now available in waveforms when they are not encrypted such as the ZEMI-3 transactors. This feature is called the Transactional Stimuli Capture and Replay technology.

Transactional Stimuli Capture and Replay is now enabled by default at compile time.

For more information, see the following subsections:

- Sniffer Usage
- Replay Usage
- Identifying Usage of Stimuli Capture and Replay
- Stimuli Capture and Replay Support for Direct-ICE and Speed Adaptor

Sniffer Usage

The following sniffer operations are supported with stimuli capture and replay:

```
sniffer -auto_create <minutes>|<seconds>s|<minutes>m|<hours>h|-cycles
 <n>|-runtime <n>h|m|s|ms|us|ns|ps|fs [-prefix <prefix>]
sniffer -config <name> [<value>]
```

Options:

```
keep_frames <n>
output_record
no_memory_copy
   save_state_at_stop (legacy will accept this option but since we can
 only replay until last sampling edge - 1, we cannot do any comparisons)
sniffer -import <path> [<name>]
sniffer -convert <entry|path> [-frames <n>]
sniffer -info [<name>] [-clock <clock>] [-text]
sniffer -info [<name>] [-cycles|-time]
sniffer -reference_clock
sniffer -list [-dbs | <db_name> | -text]
sniffer -list -at <n>|<time><time_unit> [-db <db_name>] [-clock <clock>]
sniffer -list -event [-before <n>|<time><time_unit>]
sniffer -restore <name> [-frames <n>]
sniffer -restore -event [-before <n>|<time><time_unit>]
sniffer -restore -at <n>|<time><time_unit> [-db <db_name>] [-clock
 <clock>]
sniffer -start_frame [<name>]
sniffer -state_capture <on|off>
sniffer -status
sniffer -stop
```

For details, see the *ZeBu Unified Command Line User Guide*.

Replay Usage

The following replay operations are supported with stimuli capture and replay:

```
replay [<cycles|time>]
replay -end
replay -current
replay -status
replay -config <config> <value>
```

Options:

```
enable_state_checks <on>
enable_io_checks <both>
progress <on|off>
multi_host_command <remote_command> [<command>] (<remote_command> must
 contain %host)
```

For details, see the *ZeBu Unified Command Line User Guide*.

Identifying Usage of Stimuli Capture and Replay

To check whether stimuli capture and replay was enabled at compilation time, use the `sniffer -at`_speed UCLI command. If stimuli capture and replay is in use, it returns *1*. Otherwise, it returns *0*.

Note:

Previous `zebu_work` configuration is mandatory for this command.

Stimuli Capture and Replay Support for Direct-ICE and Speed Adaptor

Recording and replaying the Stimuli is supported with Direct-ICE, Ethernet Speed Adaptor, and PCIe Speed Adaptor for maximum of 24K direct-ICE signals.

Feedback

4

Capturing Waveforms Using $dumpvars System Task

This section describes the following tips:

- Compilation: FWC $dumpvars and $dumpports Common Syntax
- Specifying Maximum Bits for $dumpvars/$dumpports

Compilation: FWC $dumpvars and $dumpports Common Syntax

Table 1 Top-Level Usage

$dumpvars	**$dumpports**
`$dumpvars(1, "dut.inst0*");`	Capture all signals in this instance
`$dumpports("dut.inst0");`	Capture all ports of an instance
`$dumpvars(1, "dut.core.signal");`	Capture an individual signal

Table 2 Special Cases

$dumpvars	**$dumpports**
`$dumpvars(1, dut.core.signal);`	If signal does not exist, compile displays an error
`$dumpvars(1, "dut.core.signal");`	If signal does not exist, compile displays a warning
`$dumpvars(1, dut.core.\escape);`	Add the required space at the end
`$dumpvars(1, "dut.core.\\escape");`	Add space at the end and extra '\'
`$dumpvars(1, dut.core.signal[31:0]);`	
`$dumpvars(1, "dut.core.signal[31:0]")`	

Table 2 Special Cases (Continued)

$dumpvars	**$dumpports**
`$dumpvars(1, dut.core.\escape[0] [31:0]);`	Range is not part of the vector name
`$dumpvars(1, "dut.core.\\escape[0] [31:0]");`	Range is not part of the vector name
``$dumpvars(1, `MACRO_TOP.core.signal);``	Macro
``$dumpvars(1, `"`MACRO_TOP.core.signal`");``	Macro with quotation marks

Specifying Maximum Bits for $dumpvars/$dumpports

To limit diagnostics for `$dumpvars()` or `$dumpports()` with VCS, the following options allow you to specify the maximum number of bits to be captured by a single `$dumpvars()` or `$dumpports()` task:

- Setting Global Limits
- Setting Per-Command Limits
- Usage Information

Setting Global Limits

To set a global limit on the bits to capture, use the following UTF commands:

```
debug -dumpvars_maxbits <int>
debug -dumpports_maxbits <int>
```

In either case, the default is *0* (unlimited). If the specified limit is reached, an elaboration error is displayed.

Example

```
debug -dumpvars_maxbits 100
```

If the limit is exceeded, the following error message is displayed by VCS:

```
Error-[FS_MAXBITS_EXCEEDED] FSDB maxbits limit exceeded
vlog_top.v, 35
```

Setting Per-Command Limits

To set a per-command limit, add the `maxbits pragma` to the specific statements:

- `(*maxbits=<num>*) $dumpvars(…)`
- `(*maxbits=<num>*) $dumpports(…)`

Example

```
(*maxbits=256*) $dumpvars(1,top.a.my_inst);
```

The `pragma` to the left of the `$dumpvars` indicates that if more than 256 bits are to be captured by the task, an error should be reported.

Usage Information

The `per-command pragma` attributes override the global limits.

The `Error-[FS_MAXBITS_EXCEEDED]` can be downgraded to a warning using the `-error=noFS_MAXBITS_EXCEEDED` VCS switch.

When the error is downgraded, all the specified bits are captured (as if there were no limits).

5

Using Static Trigger, Dynamic Trigger, and Runtime Trigger

Dynamic triggers and runtime triggers are used in the debug flow to notify the impact of an event or a sequence of events on DUT. You can then take appropriate actions, such as capturing waveforms, restoring a saved state, and so on.

This section describes the following subtopics.

- Static Trigger
- Dynamic Trigger Technology
- Limitation on the Number of Static and Dynamic Triggers
- Runtime Trigger

Dynamic triggers and runtime triggers are used in the debug methodologies described in the *ZeBu Debug Methodology Guide*.

Static Trigger

A Static trigger compares two signals or compares a signal with a constant. Any scalar signal can be specified as a static trigger input. To declare a static trigger, use the following UTF command:

```
set_trigger -name <name> -hdl_path {<RTL path is bit signal>}
```

Dynamic Trigger Technology

Using the dynamic trigger technology, you can choose the signals to connect to a `zceiTrigger` Verilog module at compile time.

At runtime, you can define a combination of values for your signal and make your dynamic trigger stop the emulation. You have an option to decide the actions that must be taken.

The dynamic trigger technology is cycle-accurate because it is handled by the hardware.

For each dynamic trigger, only an AND operation can be performed while comparing signals with value.

When using multiple dynamic triggers, the OR operation between them is applied if they are active at the same time.

This section describes the following subtopics:

- Defining the Signals at Compile Time
- Defining a Value to Stop Runtime

Defining the Signals at Compile Time

At compile time, define the list of signals by connecting them to a `zceiTrigger` Verilog instantiated module.

Verilog Example

```
zceiTrigger Counter_Dynamic_Trigger (
.trigger_input(hw_top.async_fifo_a.counters_st.cycle_counter_inc[31:0])
);
```

Defining a Value to Stop Runtime

At runtime, you can dynamically program the dynamic trigger to decide when to stop the run. To program the dynamic trigger, set the value to each signal being used.

The following example explains how to stop the runtime using the `stop` UCLI command.

```
stop -expression {hw_top.async_fifo_a.counters_st.cycle_counter_inc[31:0]
 == 32'd8} hw_top.Counter_Dynamic_Trigger
stop -enable hw_top.Counter_Dynamic_Trigger -action <callback action>
run 2000
```

Note:

For C/C++ testbenches, you can use the `Sniffer` and `RunManager` API to control the dynamic trigger. For details, see the `ZEBU_Sniffer.hh` and `RunManager.hh` header files.

Assume the following declaration in the RTL:

```
zceiTrigger
 myDynTrigger( .trigger_input( hw_top.dut.g0.bk[0].b0.count.cnt ) );
```

where, `cnt` is a vector.

Few examples depicting valid expressions for runtime are shown as follows:

- Compare one bit

```
hw_top.dut.g0.bk[0].b0.count.cnt[0] == 1
hw_top.dut.g0.bk[0].b0.count.cnt[7] == 1
```

- Compare to Verilog constants

```
hw_top.dut.g0.bk[0].b0.count.cnt[0] == 1'b1
hw_top.dut.g0.bk[0].b0.count.cnt == 16'hFF
hw_top.dut.g0.bk[0].b0.count.cnt == 8'hF
```

- Use the && operator

```
hw_top.dut.g0.bk[0].b0.count.cnt[0] == 1 &&
 hw_top.dut.g0.bk[0].b0.count.cnt[7] == 1
```

- Use braces

```
(hw_top.dut.g0.bk[0].b0.count.cnt[0] == 1) &&
 (hw_top.dut.g0.bk[0].b0.count.cnt[7] == 1)
```

- Add spaces around braces

```
( hw_top.dut.g0.bk[0].b0.count.cnt[0] == 1 ) &&
 ( hw_top.dut.g0.bk[0].b0.count.cnt[7] == 1 )
```

- Compare the full vector

```
hw_top.dut.g0.bk[0].b0.count.cnt == 1
hw_top.dut.g0.bk[0].b0.count.cnt == 0
hw_top.dut.g0.bk[0].b0.count.cnt == 16
```

- Remove spaces around the operator

```
hw_top.dut.g0.bk[0].b0.count.cnt==1
hw_top.dut.g0.bk[0].b0.count.cnt[0]==1
```

- Compare two parts selection of the same vector

```
hw_top.dut.g0.bk[0].b0.count.cnt[1:0] == 2'b11 &&
 hw_top.dut.g0.bk[0].b0.count.cnt[6:7] == 2'b11
```

Few examples depicting non-valid expressions for runtime are shown as follows:

- Constants
  - *1*
  - *0*
- Using EDIF names for signals

Feedback

```
hw_top.dut.g0_bk_0__b0.count.cnt[0] == 1
hw_top.dut.g0_bk_0__b0.count.cnt == 1
```

- Using the || operator

```
( hw_top.dut.g0.bk[0].b0.count.cnt[0] == 1 ) ||
 ( hw_top.dut.g0.bk[0].b0.count.cnt[7] == 1 )
```

- Using a signal that was not included at compilation time

```
hw_top.dut.g0.bk[1].b0.count.cnt == 1
hw_top.dut.g0.bk[0].b0.count.cnt == 1 &&
 hw_top.dut.g0.bk[1].b0.count.cnt == 1
hw_top.dut.g0.bk[0].b0.count.cnt == 1 ||
 hw_top.dut.g0.bk[1].b0.count.cnt == 1
```

- Using C constants

```
hw_top.dut.g0.bk[0].b0.count.cnt == 0xFF
hw_top.dut.g0.bk[0].b0.count.cnt == 0xF
```

- Using the != operator

```
hw_top.dut.g0.bk[0].b0.count.cnt != 16
```

- Using non Verilog-compliant constants

```
hw_top.dut.g0.bk[0].b0.count.cnt == FF
hw_top.dut.g0.bk[0].b0.count.cnt == F
```

Limitation on the Number of Static and Dynamic Triggers

There is a limit on the total number of Static and Dynamic triggers based on the hardware platform as listed in the following table:

Table 3 Maximum Number of Static and Dynamic Triggers

Hardware Platform	Maximum Number of Static and Dynamic Triggers
Other platforms (except when using C-Cosim)	32
Other platforms when using C-Cosim	16

Runtime Trigger

At runtime, the testbench can be notified using a procedure when a sequence of DUT events occurs.

The sequence of DUT events is a Finite State Machine (FSM) described using Complex Event Language (CEL). For more details, see the *ZeBu-Verdi Integration Guide*.

The signals used by the FSM are captured at runtime using FWC or QiWC technologies. The FSM transitions are based on the sampling clock that is used to capture FWC and QiWC data.

The notification from the emulator to the testbench is always delayed.

To activate the runtime trigger at runtime, see the *ZeBu Unified Command-Line User Guide*.

For details on CEL, see the *ZeBu-Verdi Integration Guide.*

This section describes the following subtopics:

- Setting Up CEL File for Runtime Trigger
- Using Runtime Triggers in zRci

The debug methodology associated with runtime trigger is best suited to monitor key signals to determine a debug window. To determine the root cause, you can capture and expand the waveforms of signals in the debug window. This methodology is described in the *ZeBu Debug Methodology Guide*.

Setting Up CEL File for Runtime Trigger

The sample number corresponds to the ZTDB sampling mechanism. By default, the sample number corresponds to the number of edges (both positive and negative edges) of all primary clocks.

Runtime trigger reports state transitions and notifications using:

- Sample number and DUT clock cycle if the compilation is done using ZCEI clocks and if DUT clock is specified while attaching the runtime trigger to the ZTDB capture
- Sample number and time if the compilation is done using RTL clocks

Here is an example to show the usage of runtime trigger with CEL.

Example

```
module complex_cel;

clock posedge hw_top.async_fifo_a.wclk_i;
```

```
reset negedge hw_top.rstn_i;

var ff          hw_top.ff_o;
var we          hw_top.we_i;

var dut_counter [31:0]
 hw_top.async_fifo_a.counters_st.cycle_counter_inc[31:0];

counter l_delay;
event e_ff := ff == 1'b1;
event e_we := we == 1'b1;

event e_00 := (dut_counter == 8'd00);
event e_01 := (dut_counter == 8'd01);
event e_02 := (dut_counter == 8'd02);
fsm my_FSM {
    initial S00;
    state S00 { if (e_00 occurs 1) then goto S01      }
    state S01 { if (e_01 occurs 1) then goto S02      }
    state S02 { if (e_02 occurs 1) then goto S03      }

    state S03 { if (e_18 occurs 1) then goto S_Notify }
    state S_Notify {
        notify
    }
}

endmodule
```

To align the sample number and notification time/cycle with the clocks defined in the CEL file, the CEL clock is enabled by default.

$callback Support in CEL

The `$callback` support with CEL enables you to execute Tcl callback in **zRci**.

CEL syntax in a State definition:

```
$callback( "This is a callback with %lu %s", p, "parameters.");
```

Example:

To enable `$callback` in **zRci**, use the `-cb_action` to define the Tcl command and execute as follows:

```
stop -cel /path/to/fsm.cel -action RT_callback -cb_action dollar_callback
```

where, `dollar_callback` is defined as:

```
proc dollar_callback {module sampleNumber userClock CEL_param} {
  puts "\$callback exected with module            $module"
  puts "\$callback exected with sampleNumber      $sampleNumber"
  puts "\$callback exected with userClock         $userClock"
}
```

Guidelines

- If the clock is provided in the CEL file, it can be used as FSM's sampling clock;
- If the clock is provided in the CEL file but, it is incorrect (bad name, or no underlying FWC co-ordinates), an error is displayed.
- If the clock is not provided in the CEL file, the sampling clock of the underlying FWC stream is used as FSM's sampling clock.

When the CEL clock is disabled, the CEL clock is not considered in any circumstance even if it is provided in CEL file. The FSM's sampling clock is the underlying FWC's sampling clock.

Note:

The CEL clocks are limited to the primary clocks (outputs of `zceiClockPort` instances).

Using Runtime Triggers in zRci

Here is an example to show the usage of runtime trigger with `zRci`. The `dump` and `stop` UCLI commands are used.

Example

```
#UCLI Callback for Runtime Trigger
proc SWN_callback {module sampleNumber ClockCycle isLastNotify} {
    global __sw_notifier_done

    puts "swn-test-callback: NOTIFICATION START"

    puts "swn-test-callback: module           = $module"
    puts "swn-test-callback: sampleNumber     = $sampleNumber"
    puts "swn-test-callback: ClockCycle       = $ClockCycle"
    puts "swn-test-callback: isLastNotify     = $isLastNotify"
    set __sw_notifier_done $isLastNotify
    puts "swn-test-callback: NOTIFICATION CALLBACK DONE"
    puts ""
}
stop -cel ../SRC/CEL/complex.cel -action SWN_callback -clock hw_top.clk

# run emulation
```

To use the runtime trigger in the C++ testbench, use the *SwNotifier.hh* API.

Note:

Only the required IPs are selected to capture the ZTDB and not the entire ValueSet.

6

Debug-Related Limitations

This section describes the following debug-related limitations:

- Value-Sets
- Emulation Runtime Speed While Capturing Streaming Raw Data
- Dynamic Trigger

Value-Sets

Value-Set labels must be different from the enclosing SystemVerilog module name (for example, `hw_top`).

Emulation Runtime Speed While Capturing Streaming Raw Data

The global emulation speed can decrease due to network traffic and disk access, which might impact any of the waveform capturing mechanisms.

Waveform capture performance depends largely on the number of signals displayed and their activity level.

The maximum number of Value-Sets for QiWC is 16.

Dynamic Trigger

While using several dynamic triggers simultaneously, an implicit OR is applied on all of them.

Example

```
stop -expression {hw_top.top.counter1 == 32'd8} hw_top.Dynamic_Trigger_1
stop -expression {hw_top.top.signal == 32'd12345}
 hw_top.Dynamic_Trigger_2
stop -enable hw_top.Dynamic_Trigger_1 -action Trigger_callback
stop -enable hw_top.Dynamic_Trigger_2 -action Trigger2_callback
…
```

```
puts "hw_top.Dynamic_Trigger_1: state = [stop -state
 hw_top.Dynamic_Trigger_1]"
puts "hw_top.Dynamic_Trigger_2: state = [stop -state
 hw_top.Dynamic_Trigger_2]"
…
stop -disable hw_top.Dynamic_Trigger_1
…
stop -disable hw_top.Dynamic_Trigger_2
…
```