Verification Continuum™

# ZeBu® Server Debug Guide

**Version Q-2020.03-1, July 2020**

SYNOPSYS®

## Copyright Notice and Proprietary Information



## Destination Control Statement

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

## Disclaimer

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

## Trademarks

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

## Free and Open-Source Software Licensing Notices

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

## Third-Party Links

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

*www.synopsys.com*

# Contents

**Preface..... 9**
About This Book ..... 9
Contents of This Book ..... 9
Related Documentation ..... 10

**1. Introduction to Debug Flow .....11**
**1.1. Debug Flow Overview..... 11**
**1.2. Glossary of Debug-Related Terms ..... 12**

**2. Waveform Capture and Expansion.....15**
**2.1. Debug Setup for Compilation..... 18**
2.1.1. Common Compilation Setup for Debug .....19
2.1.2. Compilation Setup for Dynamic-Probes.....20
2.1.3. Compilation Setup for Fast Waveform Capture (FWC) .....21
2.1.4. Compilation Setup for Quick Waveform Capture (QiWC).....25
2.1.5. Compilation Settings for Using Dynamic-Probes, FWC, and QiWC Simultaneously .....27
**2.2. Runtime Control for Outputting ZTDB Waveforms ..... 28**
2.2.1. Sampling Clock Selection.....29
2.2.2. ZTDB Slicing.....31
2.2.3. Using zConvertToFsdb with zSimzilla .....32
2.2.4. Using the ZeBu Runtime Control Interface (zRci) for Debug.....34
**2.3. Waveform Reconstruction and Expansion ..... 37**
2.3.1. Waveform Reconstruction .....37
2.3.2. Waveform Expansion .....38

**3. Recording and Replaying Stimuli (Snapshot) .....43**
**3.1. Compilation Setup for Stimuli Replay ..... 45**
**3.2. Initial Emulation Runtime for Sniffer..... 45**
**3.3. Replaying the Stimuli with the Injector..... 47**

**4. Capturing Waveforms Using $dumpvars System Task ..... 51**
**4.1. Compilation: FWC $dumpvars and $dumpports Common Syntax .... 51**
**4.2. Specifying Maximum Bits for $dumpvars/$dumpports ..... 52**

**5. Using Dynamic Trigger and Runtime Trigger ..... 55**
**5.1. Dynamic Trigger Technology ..... 55**
**5.2. Runtime Trigger ..... 56**
5.2.1. Using Runtime Trigger With a CEL Module .....57
5.2.2. Using Runtime Triggers .....59

**6. Debug-Related Limitations ..... 61**

# List of Figures

ZeBu Debug Flow Overview ..... 12
Waveform Viewing Methods ..... 16
Waveform Capture Technology X Compilation Impact ..... 17
Sampling Clock Effect on Waveforms ..... 30
ZTDB Slicing ..... 31
Waveform Capture and Expansion using -timescale ..... 33
Waveform Capture and Expansion using -timescale and --align-timescale 33
Waveform capture and expansion Using zConvertToFsdb and zSimzilla 34
nWave Interface ..... 38
Interactive Waveform Expansion With Verdi ..... 42
Global View During the Emulation Runtime ..... 44
Integrating Sniffer and Clock Connectivity ..... 45
Replaying the Stimuli with the Injector ..... 47

# List of Tables

**Top-Level Usage..... 51**
**Special Cases ..... 51**

# Preface

## About This Book

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

## Contents of This Book

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

| Chapter | Describes... |
|---|---|
| *Introduction to Debug Flow* | an overview of debug flow. |
| *Waveform Capture and Expansion* | waveform capturing mechanisms:<br>• Dynamic-Probes<br>• FWC<br>• QiWC |
| *Recording and Replaying Stimuli (Snapshot)* | the usage to record and replay the Stimuli sent to the design during the emulation. |
| *Capturing Waveforms Using $dumpvars System Task* | compilation tips. |
| *Using Dynamic Trigger and Runtime Trigger* | the usage of dynamic trigger and Runtime Trigger in the debug flow. |
| *Debug-Related Limitations* | the limitations of debug feature. |

# 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.<br>For VCS and Verdi, see the following:<br>- *Coverage Technology User Guide*<br>- *Coverage Technology Reference Guide*<br>- *Verification Planner User Guide*<br>- *Verdi Coverage User Guide and Tutorial*<br>For SystemVerilog, see the following:<br>- *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.<br>For SpyGlass, see the following:<br>- *SpyGlass Power Estimation and Rules Reference*<br>- *SpyGlass Power Estimation Methodology 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 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. 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 Server Debug Methodology Guide*.

## 1.1 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.

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

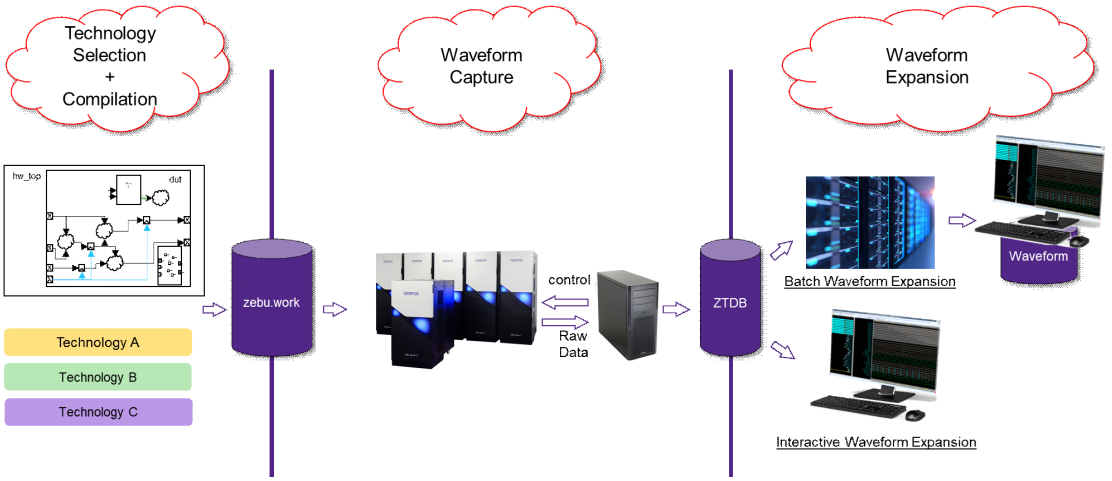


**FIGURE 1.** ZeBu Debug Flow Overview

# 1.2 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 |

| Term | Definition |
|---|---|
| ZTDB | Native ZeBu Database |
| ZWD | ZeBu Waveform Database - Viewable format |

# 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:
     - ◆ `zWaveform/zSimzilla --capture-only`
   - ❒ Expansion:
     - ◆ Batch mode: `zWaveform/zSimzilla`
     - ◆ Interactive mode: **`verdi`** (running expansion engine)

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

The following figure summarizes the waveform conversion and expansion methods:

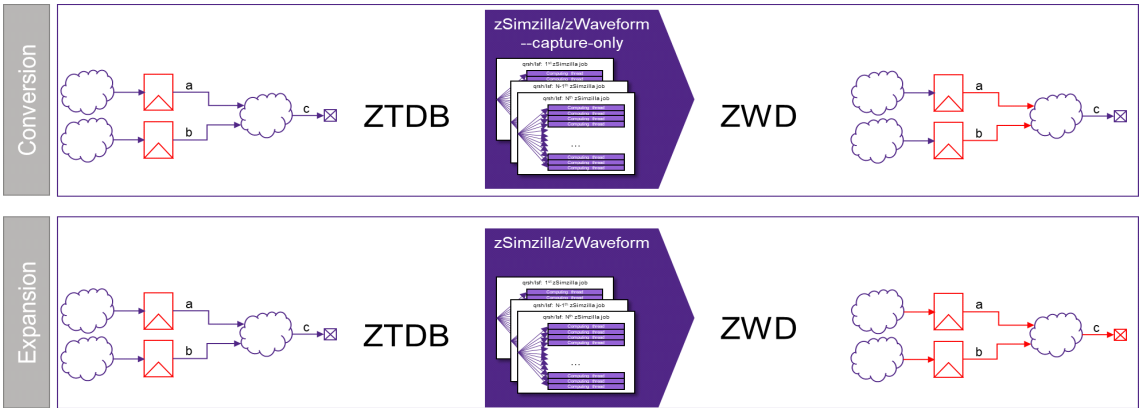


**FIGURE 2.** Waveform Viewing Methods

**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:

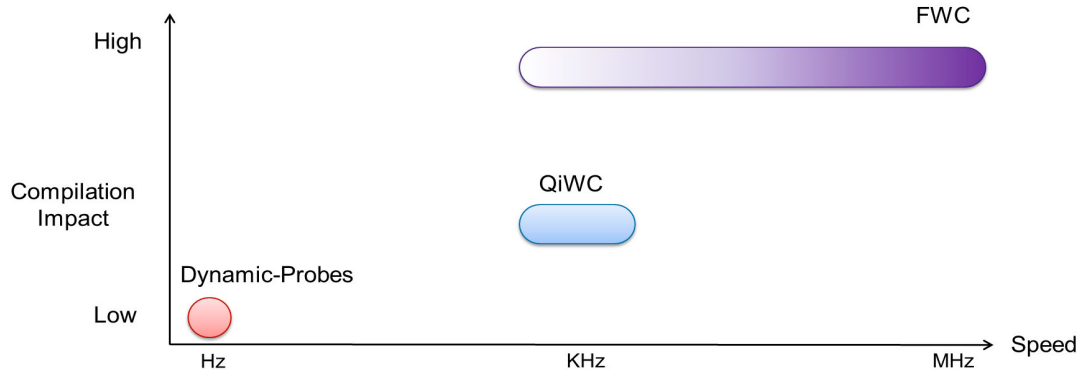


**FIGURE 3.** Waveform Capture Technology X Compilation Impact

## 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 zWaveform

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.

You can use the `zSimzilla` tool directly to run waveform expansion/reconstruction and Conversion. You can use `zWaveform`, which is the wrapper that launches **Simzilla** engine with a set of options used by default.

This section discusses the following subsections:

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

# 2.1 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

## 2.1.1 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.*

### 2.1.1.1 Generating the Verdi Database

To view and analyze your design using Verdi, you must generate the Knowledge Database (KDB). You can generate the KDB by adding the following UTF command and then compile your design with ZeBu. The KDB is generated in the ZeBu compilation directory.

To generate the KDB, add the following UTF command to your UTF file:

```
debug -verdi_db true
```

The Verdi KDB is generated in the following directory:

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

### 2.1.1.2 Enabling Waveform Expansion

The Dynamic-Probe, FWC, and QiWC waveform capture mechanisms require a common setup to enable waveform reconstruction (see *Waveform Expansion Using zWaveform*).

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

```
debug -waveform_reconstruction true
```

This controls the activation of waveform expansion.

## 2.1.2 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.

### Generating the Support File

A support file with a list of default support signals is automatically generated at compilation. It allows you to expand waveforms on all DUTs when using the dynamic-probes `ztdb` database.

This support file is automatically generated at:

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

Also, to chose combinational signals to capture with Dynamic-Probes technology, use the following UTF command:

```
probe_signals -type dynamic -rtlname hw_top.top.core1.comb_signal
```

### Custom Support File

The support file can be manually generated for any specified design instances or signals in the .zrdb extension type.

To generate the Support File, use the `zExportSupport` tool.

- For Dynamic-Probes Support file, use the following command:

```
zExportSupport -c -p <zebu.work> -z <DynProbes_selection.zrdb> -f
<list.tcl>
```

- For QiWC Support file, use the following command:

```
zExportSupport -k -q -p <zebu.work> -o <QiWC_selection.zrdb> -f
<list.tcl>
```

In both the cases, the <list.tcl> file is defined as following:

- -i <hierarchy waveforms to capture>
- -x <hierarchy to exclude from capture>
- -s <signal to capture>

## 2.1.3 Compilation Setup for Fast Waveform Capture (FWC)

This section describes the following subtopics:

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

### 2.1.3.1 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
```

### $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.

## 2.1.3.2 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.*

## 2.1.3.3 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.

## 2.1.4 Compilation Setup for Quick Waveform Capture (QiWC)

This section describes the following subtopics:

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

### 2.1.4.1 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.

### 2.1.4.2 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.*

### 2.1.4.3 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.

## 2.1.5 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
```

# 2.2 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).*

This section contains the following subsections:

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

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

## 2.2.1 Sampling Clock Selection

The Composite clock is a union of all the clocks' edges of the primary clocks, which are directly generated by ZeBu and sent to the DUT. On ZeBu Server 3, FWC and QiWC use the Composite clock. The sampling clock definition is ignored.

On ZeBu Server 4, FWC and QiWC both use the Composite clock by default. However, if the sampling clock is specified, the captured data is sampled only on the specified sampling clock.

When the Composite clock is used, the waveforms are automatically aligned with the Design clocks' edges.

When using Dynamic-Probes, you must choose an appropriate sampling clock.

The following figure illustrates the impact of an incorrect sampling clock.

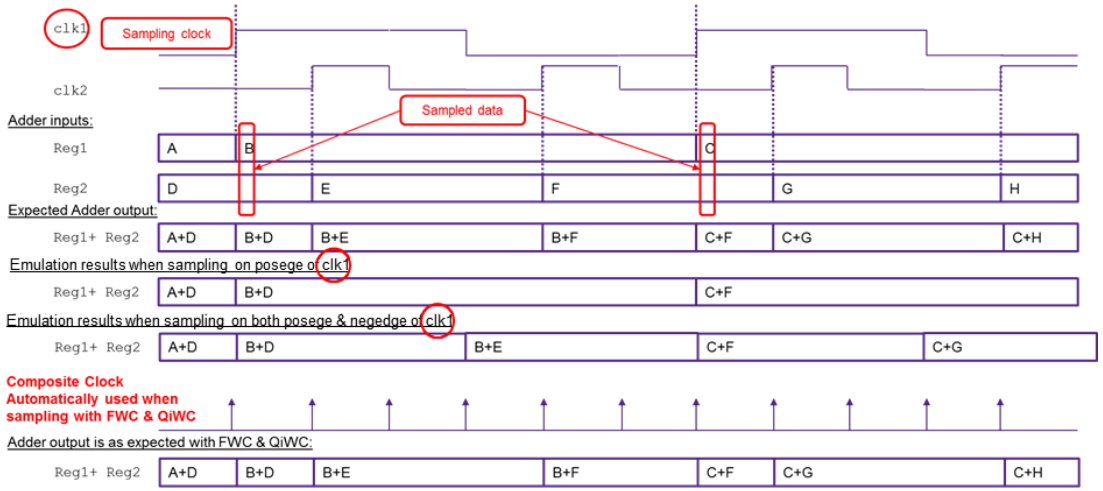


**FIGURE 4.** Sampling Clock Effect on Waveforms

When the clocks have complex timing waveforms, you can create a dummy clock that is twice as fast as the fastest primary clock in your `designFeatures` file, and use this clock as a Sampling clock.

## Example

```
$newClock = "U0.M0.hw_top.dummy_clock";
$U0.M0.hw_top.dummy_clock.Waveform = "_-_-";
$U0.M0.hw_top.dummy_clock.VirtualFrequency = 1;
$U0.M0.hw_top.dummy_clock.GroupName = "myGroup";
$U0.M0.hw_top.dummy_clock.Tolerance = "no";
```

## 2.2.2 ZTDB Slicing

To reduce the time to capture waveform, the `zSimzilla/zWaveform` 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/zWaveform` reconstructs the waveforms based on the ZTDB slices. If more number of ZTDB slices are present, `zSimzilla/zWaveform` 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.

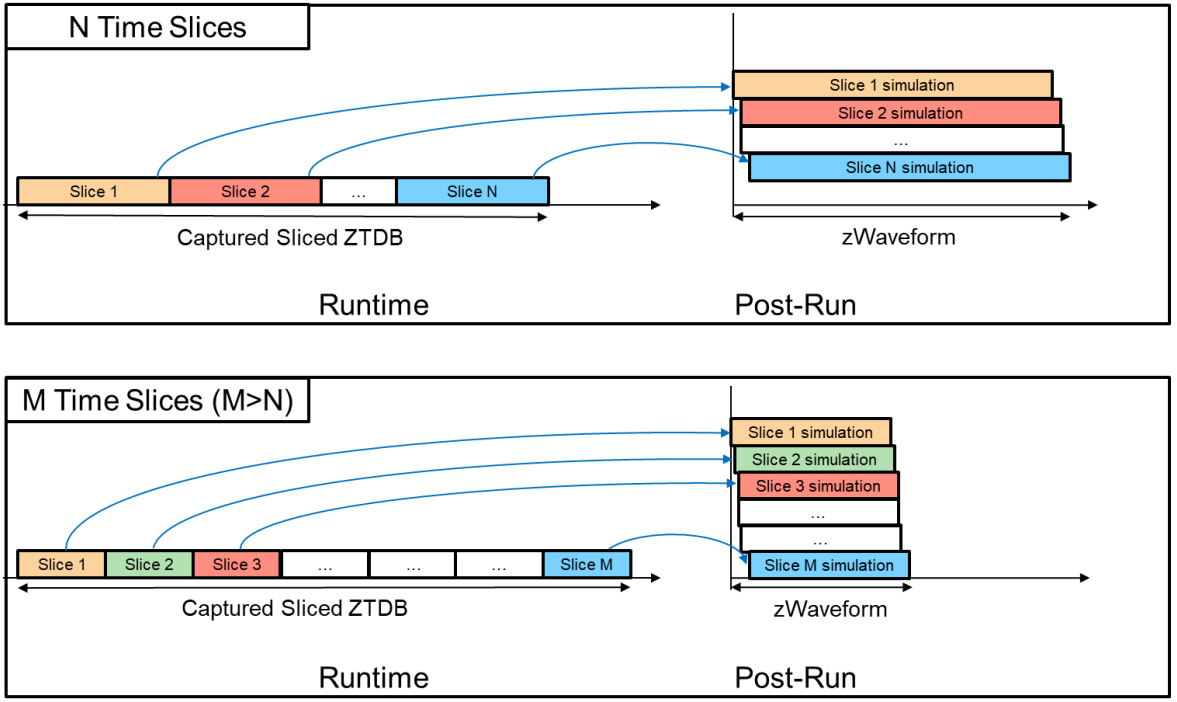


**FIGURE 5.** ZTDB Slicing

### 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 Server 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.

## 2.2.3 Using zConvertToFsdb with zSimzilla

When using the `zCeiClock` module to control the emulation runtime and capture a ZTDB, the waveform conversion with `zConvertToFsdb` and expansion with `zSimzilla` 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 FSDB/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`).

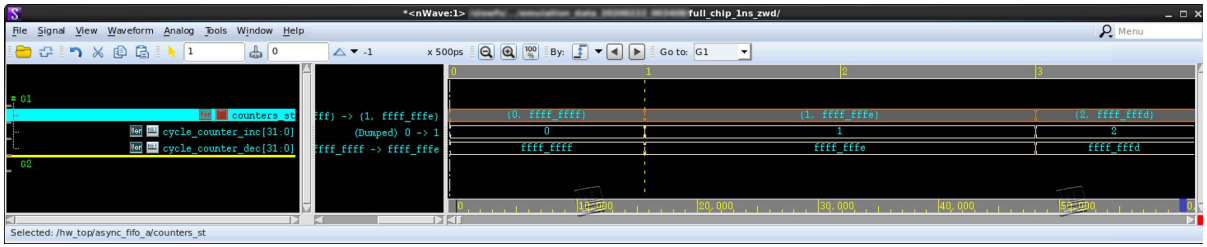


**FIGURE 6.** Waveform Capture and Expansion using -timescale

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

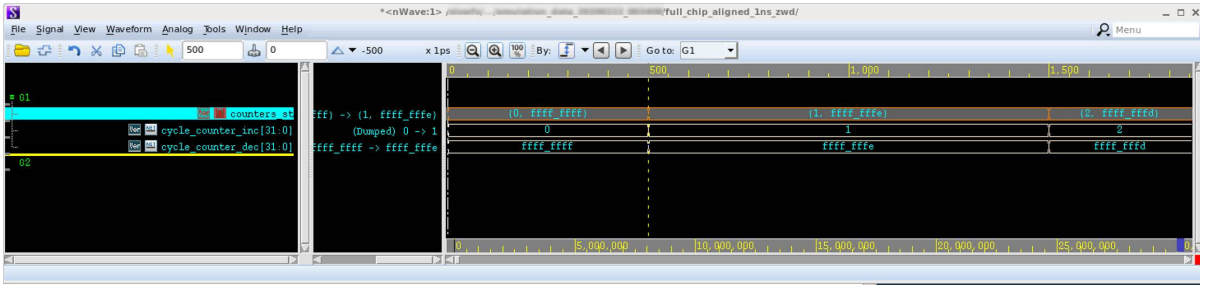


**FIGURE 7.** Waveform Capture and Expansion using -timescale and --align-timescale

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

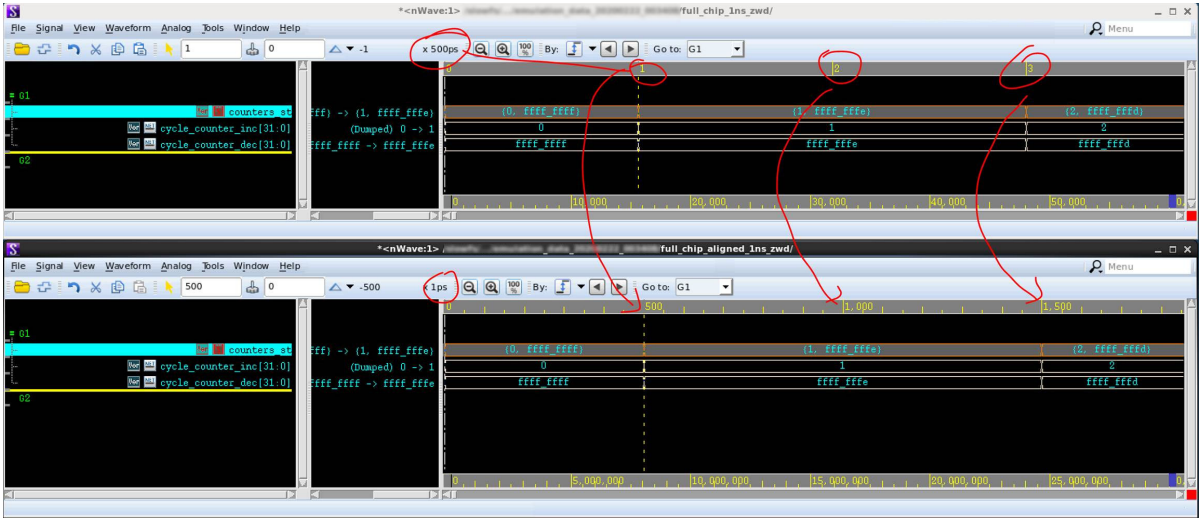


**FIGURE 8.** Waveform capture and expansion Using zConvertToFsdb and zSimzilla

## 2.2.4 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 *ZeBu Server Unified Command-Line User Guide*.

### 2.2.4.1 Using zRci to Capture Waveforms for Dynamic-Probes

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

```
#select the selection file, below is the default one you need to select
#config selection_file zcui.work/zebu.work/zrdb/csa_supports.zrdb

#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 zWaveform:
dump -interval 200 -fid $fid

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

## 2.2.4.2 Using zRci to Capture FWC and QiWC Waveforms

When using **`zRci`**, to capture waveforms, use the dump command. For more details, see *ZeBu Server 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

Waveform Viewing and Analysis

```
#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 zWaveform:
#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
```

**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

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 `zWaveform`/`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 `zWaveform` to reconstruct all waveforms.

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

# 2.3 Waveform Reconstruction and Expansion

## 2.3.1 Waveform Reconstruction

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

1. To launch `zWaveform`, perform the following:

```
zWaveform \
--capture-only
--ztdb Key_Signals.ztdb \
--work path_to/zcui.work/zebu.work \
--zwd Key_Signals \
--timescale 2ps \
--command "<remote command>" \
--jobs <User entered specified values, ex: 40> \
--log Key_Signals_zConvertToFsdb.log
```

   For best performance, use zWaveform 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 `zWaveform`.

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

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

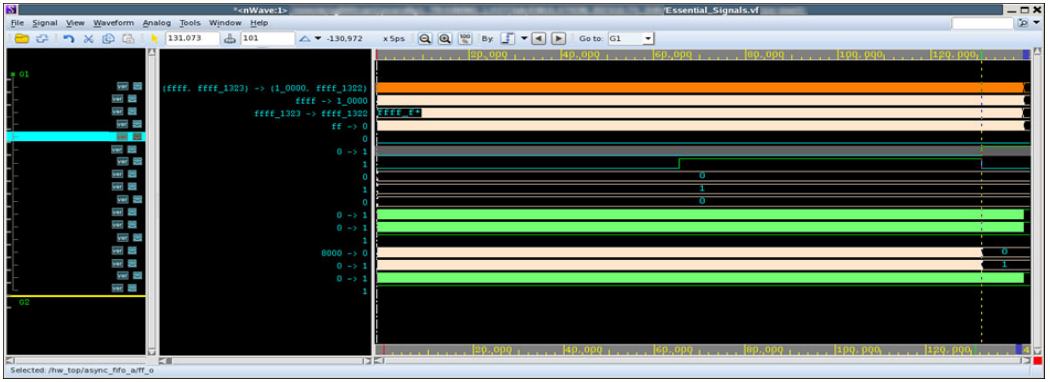

**FIGURE 9.** nWave Interface

## 2.3.2 Waveform Expansion

There are two modes to view waveforms:

- Batch mode with `zWaveform`

  `zWaveform` 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.

  It is based on the **Simzilla** engine, which is scalable. Its scalability resides in ZTDB slicing, which requires you to define an interval between each ZTDB slice when capturing ZTDB waveforms.

- 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 *ZeBu-Verdi Integration Guide*.

This section describes the following subsections:

- *zWaveform*

- *Running Interactive Waveform Expansion*

### 2.3.2.1 zWaveform

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

Each job runs using multiple CPU cores. All the jobs launched by `zWaveform` 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, `zWaveform` is faster.

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

```
zWaveform \
--ztdb captured_waveform.ztdb \
--work path_to/zcui.work/zebu.work \
--zwd my_waveform \
--timescale 2ps \
--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** `zWaveform --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.

The hierarchies and signals are passed as `--instance` and `--signal`.

### Example

```
zWaveform \
--instance <path1> --instance <path2> \
--signal <signal1> --signal <signal2> \
```

In `zSimzilla`, the hierarchies and signals to be reconstructed must be specified in the file passed using the `--zxf` option.

To specify the depth of a hierarchy to reconstruct, use the following option:

`-d <depth>`

### 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.

With `zWaveform`, this algorithm is disabled by default.

The `--xn-resolution` switch is added to the `zWaveform` command to control the X and NC resolution. By default, it is set to `true`. To disable, set this switch to `false`.

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

## 2.3.2.2 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 \
```

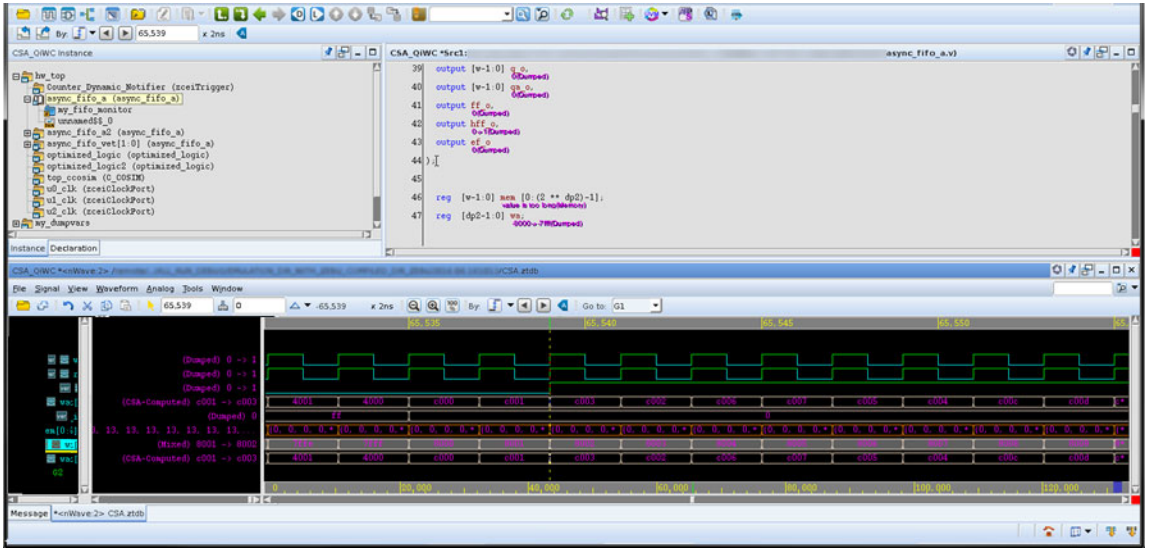

**FIGURE 10.** Interactive Waveform Expansion With Verdi

# 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.

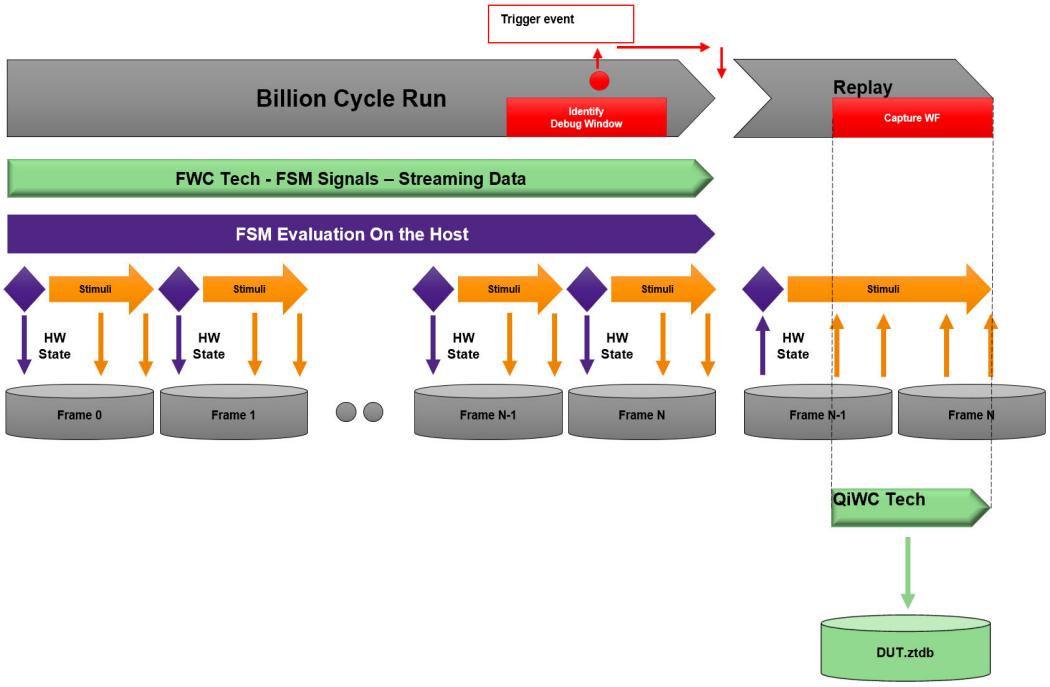


**FIGURE 11.** Global View During the Emulation Runtime

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*

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

# 3.1 Compilation Setup for Stimuli Replay

To activate the Snapshot features, add the following UTF commands in your UTF file:

```
debug -offline_debug true
debug -verdi_db      true
```

Where:

- `offline_debug`: (mandatory) Integrates the rerun capability
- `verdi_db`: (optional) Controls the generation of the Verdi KDB

# 3.2 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.

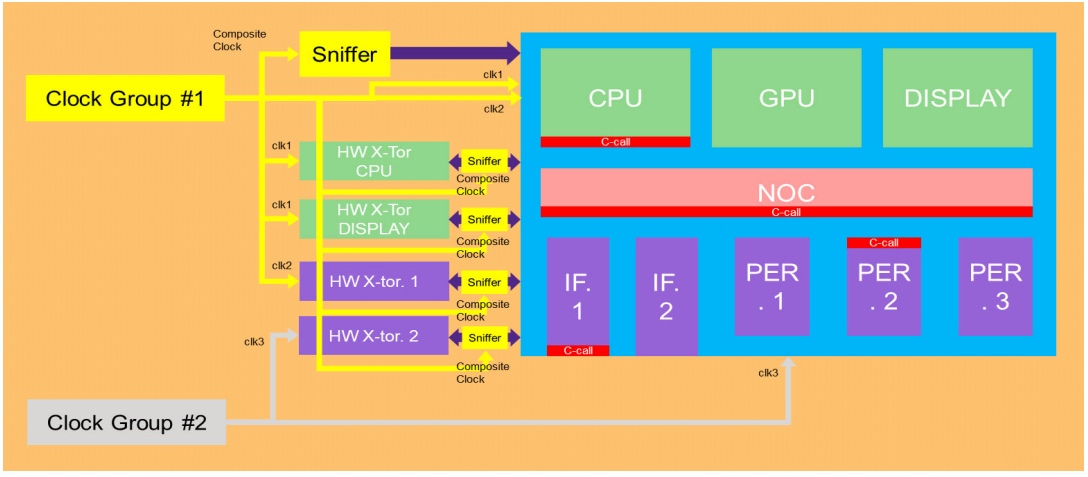


**FIGURE 12.** Integrating Sniffer and Clock Connectivity

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

For more details, see the *ZeBu Server 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.

# 3.3 Replaying the Stimuli with the Injector

Depending on the designs and amount of stimuli, you can inject the stimuli in ZTDB or in the SMD format.

The SMD format is a result of ZTDB stimuli conversion. By default, `zRci` automatically converts ZTDB to SMD as shown in the following figure. You can directly inject ZTDB with the `replay -config ztdb_threaded_scanner` UCLI command.

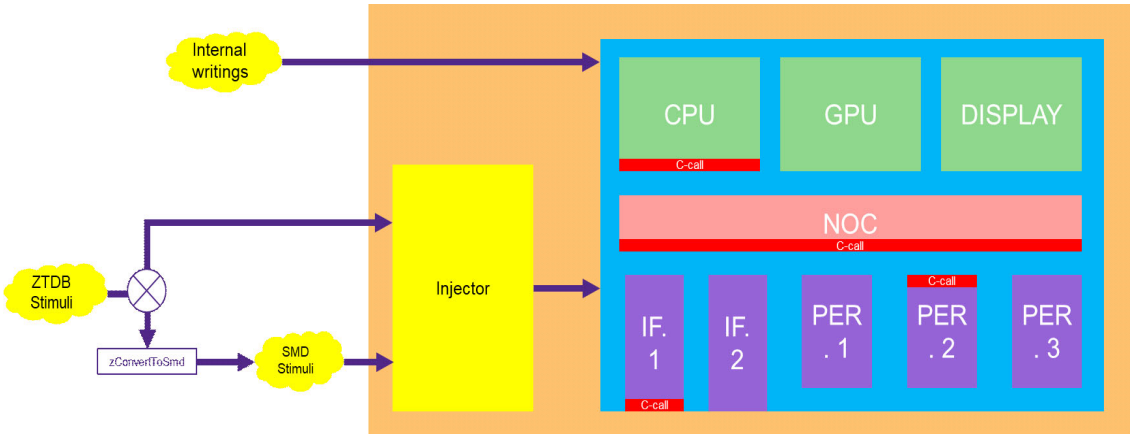


**FIGURE 13.** Replaying the Stimuli with the Injector

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 Server Unified Command-Line User Guide.*

# 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*

## 4.1 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

| $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 |

# 4.2 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 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.

- *Dynamic Trigger Technology*
- *Runtime Trigger*

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

## 5.1 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 cycle is accurate because it is handled by the hardware.

You can have up to 16 dynamic triggers on ZeBu Server 3 and 32 dynamic triggers on ZeBu Server 4.

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.

# 5.2 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 *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 Server Unified Command-Line User Guide*.

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

This section describes the following subtopics:

- *Using Runtime Trigger With a CEL Module*
- *Using Runtime Triggers*

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 Server Debug Methodology Guide*.

## 5.2.1 Using Runtime Trigger With a CEL Module

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
```

Sample number and notification time/cycle can be aligned with the clocks defined in the CEL file when using the following command:

```
stop -config cel_clock [on|off]
```

By default, it is set to `off`. To enable the alignment with clocks defined in the CEL file,

you must set to `on`.

To get the current status of this command, run `stop -config cel_clock`.

## 5.2.2 Using Runtime Triggers

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 ""
}
#UCLI commands to enable Runtime Trigger with zRci
set qiwc_id [dump -file captured_data_for_swn.ztdb -qiwc]
dump -add_value_set {my_qiwc_value_set} -fid $qiwc_id

stop -cel ../SRC/CEL/complex.cel -action SWN_callback -clock hw_top.clk
#if you do not need to capture QiWC waveform while using Runtime Trigger,
#you can comment the following line to increase runtime performance
dump -enable -fid $qiwc_id

# run emulation
```

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

# 6 Debug-Related Limitations

This section describes the following debug-related limitations:

- *Value-Sets*
- *Dynamic-Probes*
- *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`).

## Dynamic-Probes

When using Dynamic-Probes, the following `zConvertToFsdb` options are not available:

```
-j [ --jobs ] arg    Number of jobs (multi-process) (default: 1).
-c [ --command ] arg   Remote command (default: "").
-b [ --begin-cycle ] arg Begin cycle of the conversion (default: first dumped cycle).
-e [ --end-cycle ] arg   End cycle of the conversion (default: last dumped cycle).
```

## 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 dumping 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
…
```