

ZeBu[®] Server Debug Guide

Version O-2018.09-SP1, June 2019



Copyright Notice and Proprietary Information

©2019 Synopsys, Inc. All rights reserved. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

Destination Control Statement

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

Disclaimer

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

Trademarks

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

Free and Open-Source Software Licensing Notices

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

Third-Party Links

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

www.synopsys.com

Contents

Preface.....	9
About This Book	9
Contents of This Book	9
Related Documentation	10
 1. Debug Flow Introduction.....	 13
1.1. Debug Flow Overview.....	13
1.2. Glossary of Debug-Related Terms	15
 2. Waveform Capture	 17
2.1. Waveform Generation Mechanisms	19
2.2. Waveform Reconstruction	20
2.3. Debug Setup for Compilation.....	21
2.3.1. Common Compilation Setup for Debug	21
2.3.2. Compilation Setup for Dynamic-Probes.....	22
2.3.3. Compilation Setup for Fast Waveform Capture (FWC)	23
2.3.4. Compilation Setup for Quick Waveform Capture (QIWC).....	26
2.3.5. Compilation Settings for Using Dynamic-Probes, FWC, and QIWC Simultaneously	29
2.4. Runtime Control for Outputting Waveforms	29
2.4.1. Sampling Clock Selection.....	30
2.4.2. ZTDB Slicing	32
2.4.3. Using the ZeBu Runtime Control Interface (zRci) for Debug.....	33
2.4.4. Using zRun for Debug	35
2.4.5. Using C/C++ Testbench for Debug.....	38
2.5. Waveform Viewing and Analysis	42
2.5.1. Viewing Captured Signals	42
2.5.2. Waveform Reconstruction	44
 3. Recording and Replaying Stimuli (Snapshot)	 65
3.1. Compilation Setup for Stimuli Replay	67
3.2. Initial Emulation Runtime for Sniffer.....	67
3.3. Using zPostRunDebug (zPRD) to Replay Stimuli.....	69

3.3.1. Using the zPostRunDebug GUI.....	70
3.3.2. Creating a Task to Output Dynamic-Probes Waveforms and Run CSA Offline.....	71
3.3.3. Creating zPostRunDebug Tasks for QiWC Output and CSA.....	74
3.3.4. Creating zPostRunDebug Tasks for FWC Output.....	77
3.3.5. Using zRci UCLI Commands for Stimuli Replay	77
4. Capturing Using System Tasks	79
4.1. Compilation: FWC \$dumpvars and \$dumpports Common Syntax	79
4.2. Specifying Maximum Bits for \$dumpvars/\$dumpports	80
5. Using Dynamic Trigger and Runtime Trigger	83
5.1. Dynamic-Trigger Technology	83
5.2. Runtime Trigger	85
5.2.1. Using Runtime Trigger With a CEL Module	86
5.2.2. Using Runtime Trigger in C++ Testbench	87
5.2.3. Using Runtime Triggers With zRci.....	89
6. Debug-Related Limitations.....	91

List of Figures

ZeBu Debug Flow Overview.....	14
Waveform Viewing Methods.....	18
Waveform Capture Technology X Compilation Impact.....	19
Sampling Clock Effect on Waveforms	31
ZTDB Slicing	32
zRun Graphical Interface	36
FWC Window in zRun	37
nWave Interface	43
zCSA GUI.....	47
Working Directory in Project Properties Panel	48
Run Preferences in Project Properties Panel	49
Clock Preferences in Project Properties Panel	50
CSA Preferences in Project Properties Panel	51
Job Scheduling in Project Properties Panel	53
Interactive CSA With Verdi.....	64
Global View During the Emulation Runtime.....	66
Overview of the Debug With zPostRunDebug	66
zPostRunDebug Main Window	70
Dynamic-Probe Dump Task	72
Dynamic-Probes Output with CSA	73
QiWC Dump Task.....	74
QiWC Dump.....	75
Activate CSA for QiWC Dump.....	76

List of Tables

Working Directory Options and Tcl Commands..... 48

Run Preferences Options and Tcl Commands..... 49

CSA Preferences Options and Tcl Commands 51

Options Available in the Job Scheduling Section 53

Top-Level Usage..... 79

Special Cases 79

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...
Debug Flow Introduction	an overview of debug flow.
Waveform Capture	Dynamic-Probes, FWC and QiWC mechanisms.
Recording and Replaying Stimuli (Snapshot)	the usage of <code>zPostRunDebug</code> .
Capturing Using System Tasks	compilation tips.
Debug-Related Limitations	the limitations of debug feature.

Related Documentation

Document Name	Description
<i>ZeBu Server 4 Site Planning Guide</i>	Describes planning for ZeBu Server 4 hardware installation.
<i>ZeBu Server 3 Site Planning Guide</i>	Describes planning for ZeBu Server 3 hardware installation.
<i>ZeBu Server Site Administration Guide</i>	Provides information on administration tasks for ZeBu Server 3 and ZeBu Server 4. It includes software installation.
<i>ZeBu Server Getting Started Guide</i>	Provides brief information on using ZeBu Server.
<i>ZeBu Server User Guide</i>	Provides detailed information on using ZeBu Server.
<i>ZeBu Server Debug Guide</i>	Provides information on tools you can use for debugging.
<i>ZeBu Server Debug Methodology Guide</i>	Provides debug methodologies that you can use for debugging.
<i>ZeBu Server Unified Command-Line User Guide</i>	Provides the usage of Unified Command-Line Interface (UCLI) for debugging your design.
<i>ZeBu Server Functional Coverage User Guide</i>	<p>Describes collecting functional coverage in emulation.</p> <p>For VCS and Verdi, see the following:</p> <ul style="list-style-type: none">- Coverage Technology User Guide- Coverage Technology Reference Guide- Verification Planner User Guide- Verdi Coverage User Guide and Tutorial <p>For SystemVerilog, see the following:</p> <ul style="list-style-type: none">- SystemVerilog LRM (2017)
<i>ZeBu Server Power Estimation User Guide</i>	<p>Provides the power estimation flow and the tools required to estimate the power on a System on a Chip (SoC) in emulation.</p> <p>For SpyGlass, see the following:</p> <ul style="list-style-type: none">- SpyGlass Power Estimation and Rules Reference- SpyGlass Power Estimation Methodology Guide- SpyGlass GuideWare2018.09 - Early-Adopter User Guide
<i>ZeBu Verdi Integration Guide</i>	Provides Verdi features that you can use with ZeBu. This document is available in the Verdi documentation set.
<i>ZeBu Server LCA Features Guide</i>	Provides a list of LCA features available with ZeBu Server.
<i>ZeBu Server Release Notes</i>	Provides enhancements and limitations for a specific release.

1 Debug Flow Introduction

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

- 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 methodologies when using the ZeBu debug technologies, see the *ZeBu Server Debug Methodology Guide*. The guide also provides a debug planning checklist.

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 section [Runtime Control for Outputting 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.

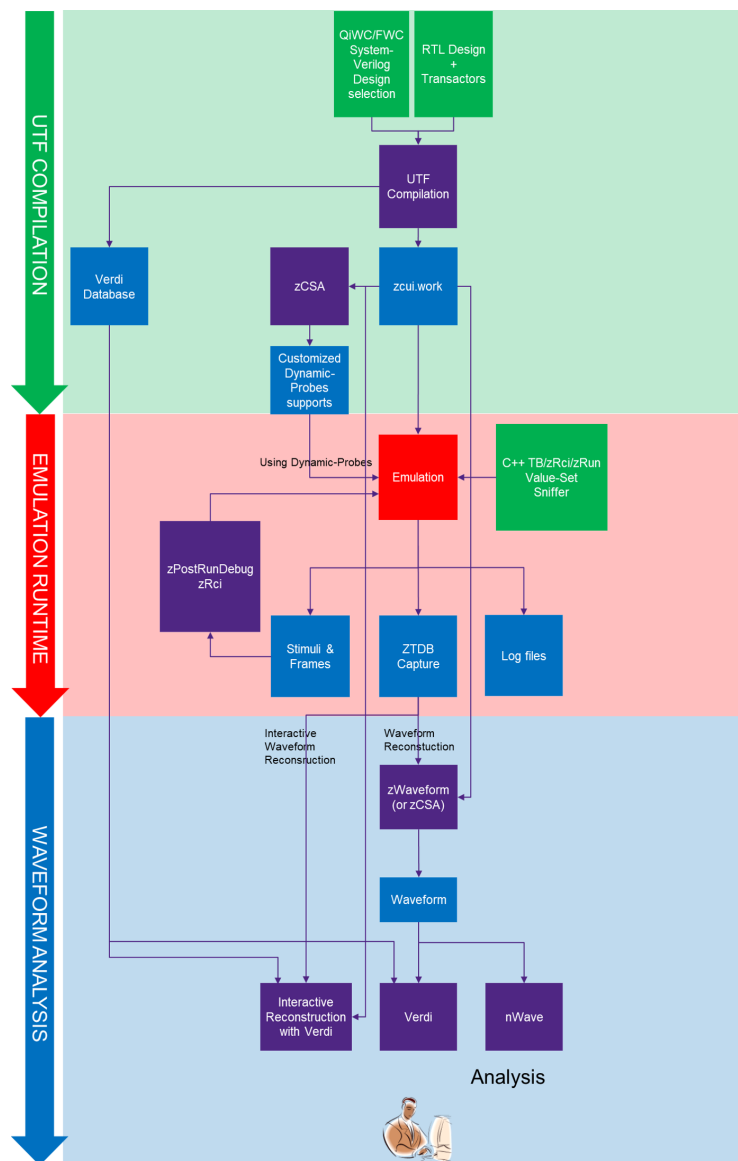


FIGURE 1. ZeBu Debug Flow Overview

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
CSA Support Signals	Signals such as flip-flop outputs required by the CSA engine to reconstruct the waveforms of combinational logic
CSA	Combinational Signal Accessibility: name of the software Waveform Reconstruction engine
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 Dump
ZWD	Zebu Waveform Database - Viewable format

2 Waveform Capture

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**
3. **Runtime:** It consists of the following:
 - ☐ Writing a testbench
 - ☐ Running the emulation
 - ☐ Dumping the waveforms
4. **PostRun:** There are several methods to view waveforms:
 - ☐ **zConvertToFbdb + nWave**
 - ☐ Reconstruct Waveforms with CSA engine. Waveform Reconstruction can be performed:
 - ♦ With a GUI or in batch: **zCSA + Verdi**
 - ♦ Completely interactively: **Verdi**
 - ☐ New Waveform Reconstruction, **zWaveform** to replace **zCSA** and **zConvertToFbdb**

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

The following figure summarizes the Waveform Viewing methods:

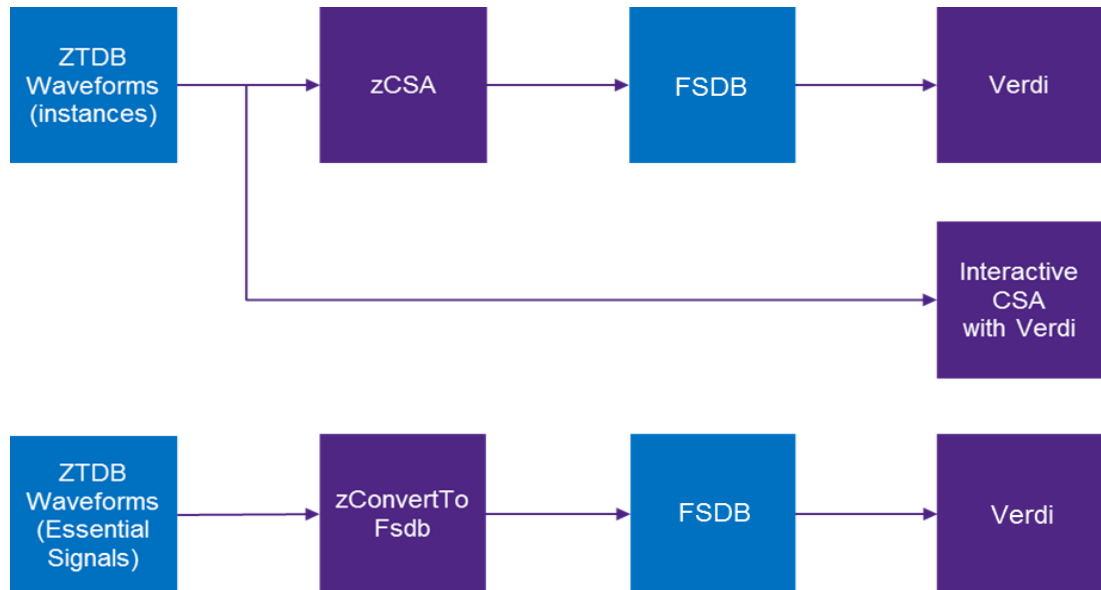


FIGURE 2. Waveform Viewing Methods

For more information, see the following subsections:

- [Waveform Generation Mechanisms](#)
- [Waveform Reconstruction](#)
- [Debug Setup for Compilation](#)
- [Runtime Control for Outputting Waveforms](#)
- [Waveform Viewing and Analysis](#)

2.1 Waveform Generation 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:

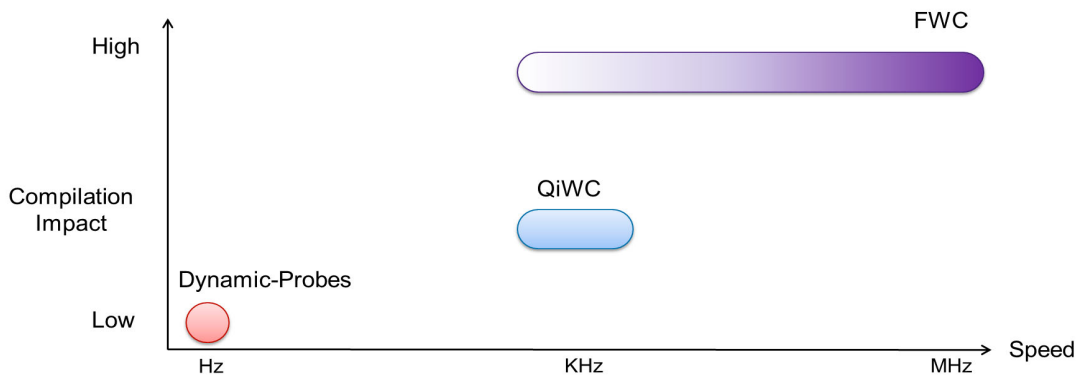


FIGURE 3. Waveform Capture Technology X Compilation Impact

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 CSA. QiWC allows runtime control over the dumping of any individual signal.

QiWC requires very few hardware resources.

The emulation runtime speed while dumping is on 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. Use **WITH CAUTION** on entire design instances.

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

Note

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

2.2 Waveform Reconstruction

There are two methods used for Waveform Reconstruction are as follows:

- **zCSA**
- **zWaveform**

zCSA

zCSA is a software engine that reconstructs the waveforms of combinational logic signals using the waveforms of the Support Signals. The engine uses information gathered during compilation.

Dynamic-Probes, FWC, and QiWC automatically select the "Support Signals" when you designate design instances.

zWaveform

zWaveform is the new software engine that also reconstructs the waveforms of

combinational logic signals. **zWaveform** is faster than **zCSA**, and its scalability resides in ZTDB slicing, which requires you to define an interval between each ZTDB slice when capturing ZTDB waveforms.

2.3 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 Combinational Signal Accessibility (CSA)
- [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

2.3.1 Common Compilation Setup for Debug

The common compilation setup consists of:

- [Generating the Verdi Database](#)
- [Enabling Combinational Signal Accessibility \(CSA\)](#)

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

2.3.1.1 Generating the Verdi Database

To view and analyze your design using Verdi, the Knowledge DataBase (KDB) needs to be generated. 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
```

2.3.1.2 Enabling Combinational Signal Accessibility (CSA)

The Dynamic-Probe, FWC, and QiWC waveform capture mechanisms require a common setup to enable Waveform Reconstruction (see [Waveform Reconstruction](#)).

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

```
debug -waveform_reconstruction true
```

This controls the activation of **zCSA** and **zWaveform**.

2.3.2 Compilation Setup for Dynamic-Probes

Generally, dynamic-probes require no UTF command. However, if you use a C-COSIM transactor and you plan 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 CSA Support Signals.

Generating the CSA Support File

The list of CSA Support Signals is automatically generated at compilation. This list is the **CSA Support file**.

This CSA Support File is automatically generated at:

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

To also capture combinational signals, use the following UTF command:

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

Custom CSA Support File

The CSA Support File can be manually generated for any specified design instance.

To generate the **Support File** manually for any instance, use the **zCSA** tool as follows:

```
zCSA \
  --zebu-work path_to/zcui.work/zebu.work \
  --hier-path hw_top.top.core1 \
  --gen-zrdb-selection <support_for_core1.zrdb>
```

Where, `support_for_core1.zrdb` is the customized **Support File** for the `hw_top.top.core1` instance.

2.3.3 Compilation Setup for Fast Waveform Capture (FWC)

For more information, see the following subsections:

- [RTL Preparation for FWC](#)
- [Compilation](#)
- [VCS Compilation and Elaboration Script](#)

2.3.3.1 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 might be named through the label associated with its corresponding Verilog `initial` block.

All un-named blocks become part of one 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 CSA 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: DUT_wo_PLL
  (* qiwc *) $dumpvars (0, hw_top.dut);
  (* qiwc *) $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 (`$dumpvars`, `$dumpports` ...) 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 designated 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.
- `Essential_Signals_NIRQ_HANDLER` and `Essential_Ports_MEM_CORE1` are the names of the Value-Sets.

2.3.3.2 Compilation

No extra command is required 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.

2.3.3.3 VCS Compilation and Elaboration Script

The VCS compilation and elaboration script needs to 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.

2.3.4 Compilation Setup for Quick Waveform Capture (QiWC)

For more information, see the following subsections:

- [RTL Preparation for QiWC](#)
- [Compilation](#)
- [VCS Compilation and Elaboration Script](#)

2.3.4.1 RTL Preparation for QiWC

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

A Value-Set might 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 CSA 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_FWC.sv

```
module DUT_QiWC();
  initial begin : csa_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.
- **csa_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**: CSA 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, CSA can be on all logic in their corresponding hierarchies. Only the Support Signals are captured. The CSA engine can reconstruct the waveform of the entire instance tree.

2.3.4.2 Compilation

No extra command is required 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.

2.3.4.3 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.

2.3.5 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 : csa_CORE1_CORE2
    (* qiwc *) $dumpvars(0, hw_top.top.core1);
    (* qiwc *) $dumpvars(0, hw_top.top.core2);
  end

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

2.4 Runtime Control for Outputting Waveforms

During runtime, you can perform several activities, such as select a sampling clock and use the ZeBu Runtime Control Interface (**zRci**) to capture raw data from ZeBu Server and save the captured information in the ZTDB database file. **zRci** provides a Tcl interface that you can use to interact with the emulation at runtime.

Apart from **zRci**, you can capture waveforms at runtime using:

- zRun
- C/C++ testbench

Note

*When using the `-testbench` option with **zRci** and **zRun**, only one of the methods must activate waveform dumping, not all (**zRci**, **zRun**, and `testbench`).*

This section contains the following subsections:

- [Sampling Clock Selection](#)
- [ZTDB Slicing](#)
- [Using the ZeBu Runtime Control Interface \(zRci\) for Debug](#)
- [Using zRun for Debug](#)
- [Using C/C++ Testbench for Debug](#)

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

2.4.1 Sampling Clock Selection

On ZS3, FWC and QiWC both use the Composite clock. The sampling clock definition is ignored.

On ZS4, 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.

Runtime Control for Outputting Waveforms

The following figure illustrates the impact of an ill-chosen sampling clock.

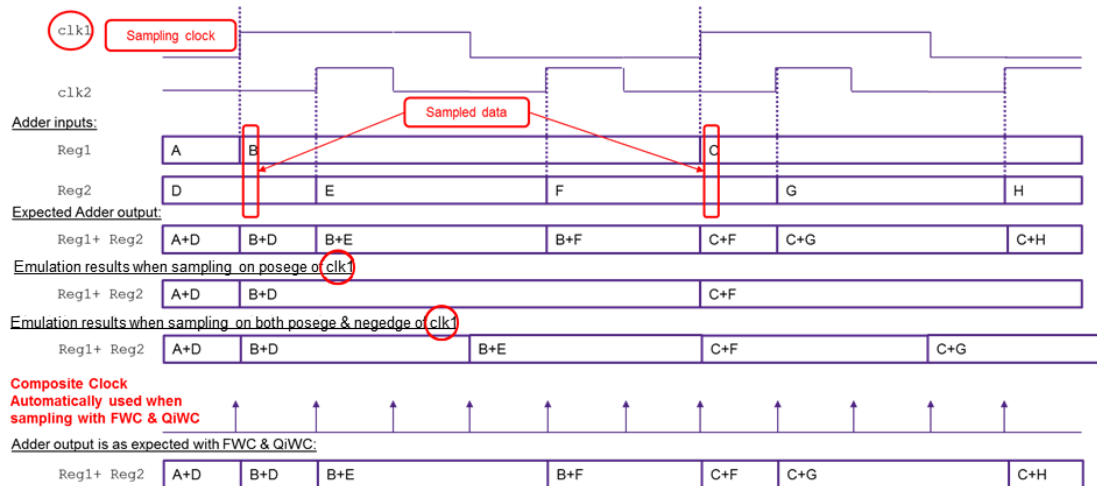


FIGURE 4. Sampling Clock Effect on Waveforms

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";
```

2.4.2 ZTDB Slicing

To reduce the time to Waveform, the **zWaveform** waveform reconstruction 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 into the disk.

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

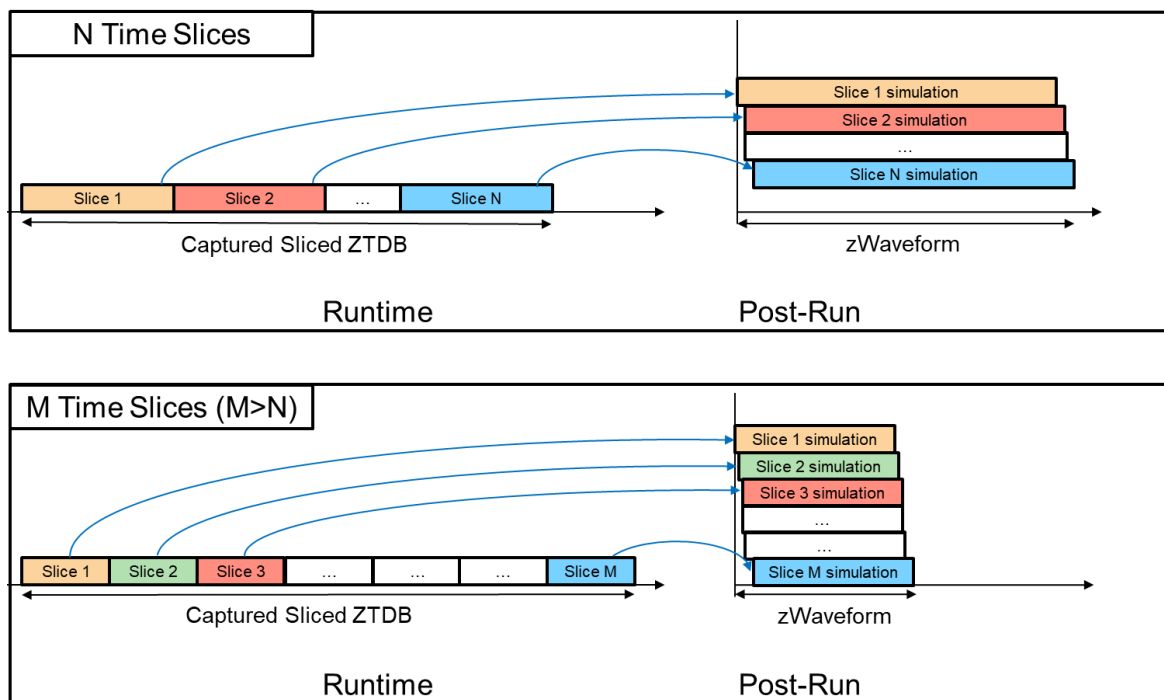


FIGURE 5. ZTDB Slicing

Auto-Slicing Support for QiWC

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

For best time to 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,50slice}
```

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

Limitation

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

The number of planned slices is limited to about 250.

2.4.3 Using the ZeBu Runtime Control Interface (zRci) for Debug

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

This section describes the following:

- [Using zRci to Capture Waveforms for Dynamic-Probes](#)
- [Using zRci to Capture FWC and QiWC Waveforms](#)

2.4.3.1 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
```

2.4.3.2 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

```
#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
```

2.4.4 Using zRun for Debug

zRun is a Tcl-based program that controls the ZeBu system emulation runtime. For more information on controlling runtime with zRun, see the ZeBu Server User Guide.

This section describes the following:

- [Using zRun for Dynamic-Probes](#)
- [Using zRun for FWC and QiWC](#)

2.4.4.1 Using zRun for Dynamic-Probes

1. Launch **zRun** with the following command:

```
zRun
-selectionDB path_to/zcui.work/zebu.work/zrdb/csa_supports.zrdb
[options]
```

Where, `-selectionDB` defines the path to the CSA support file for Dynamic-Probes.

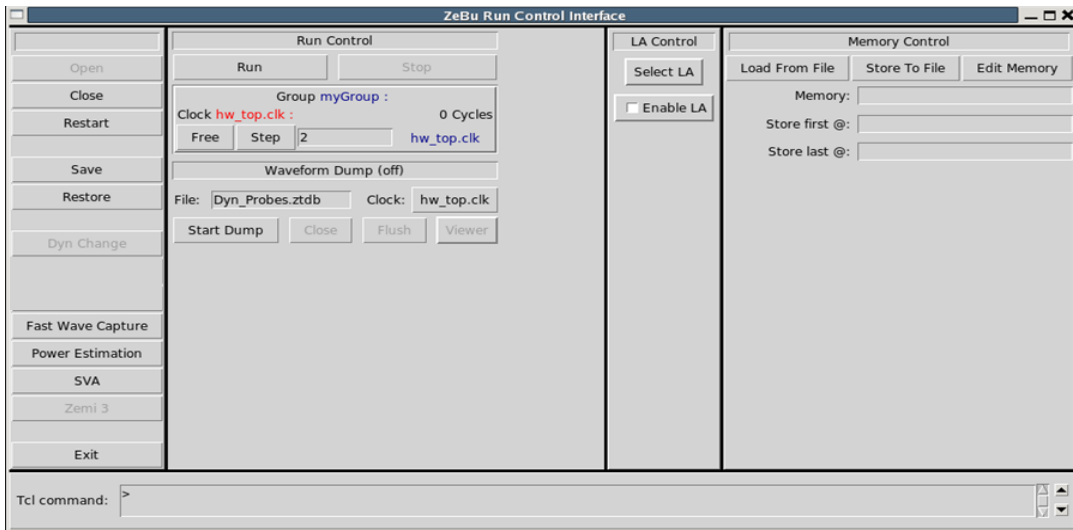


FIGURE 6. zRun Graphical Interface

2. Click **Open** to connect to the emulator.
3. Enter the waveform file name in the **File** field (**Waveform Dump** panel).
4. Specify the name of the sampling clock in the **Clock** field.
5. Click **Start Dump** to capture the ZTDB waveform. When the waveform capturing starts, the **Start Dump** is renamed **Stop Dump**, which allows you to stop the dump.
6. When finished dumping, click **Stop Dump**, and then click **Flush**.
7. Click **Close** to finish the waveform capture.

Alternatively, you can control the capture using a Tcl script that is sourced from the internal **zRun** shell or through the **zRun -do <script>**.

The Tcl API to control the waveform capture is the following:

```
ZEBU_Dump_file Dyn_Probes.ztdb hw_top.clk
ZEBU_Dump_on
...
ZEBU_Dump_off
ZEBU_Dump_flush
ZEBU_Dump_close
```

2.4.4.2 Using zRun for FWC and QiWC

You can activate FWC and QiWC waveform outputting at runtime using the **zRun** graphical interface or the Tcl API.

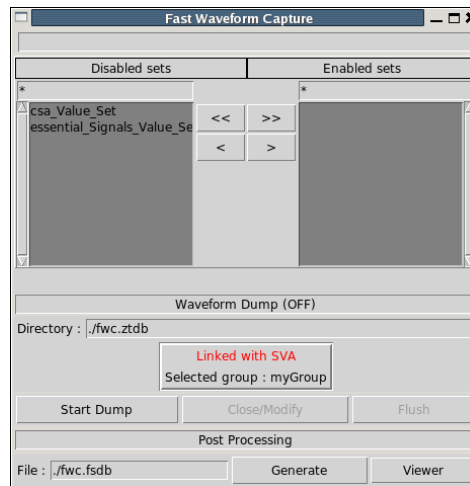



FIGURE 7. FWC Window in **zRun**

1. In the main **zRun** window, click **Fast Wave Capture** to open the **Fast Waveform Capture** window (shown above).
2. Activate Value-Set by moving it from the **Disabled Sets** column to the **Enabled Sets** column using .
3. If necessary, rename the destination directory in the **Directory** field.
4. Click **Start Dump** to capture the waveform.
5. When the dump starts, the **Start Dump** is renamed **Stop Dump**, which allows you to stop capturing the waveform.
6. When finishing the waveform capture, click **Stop Dump** > **Flush** > **Close**.
7. Click **Generate** to view the waveforms.
8. Click **Viewer** to display the waveform in Verdi's **nWave**.

Alternatively, you can control the FWC or QiWC capture using a Tcl script. The Tcl API to control the ZTDB waveform capture is the following:

```
ZEBU_Fwc_isDefined
ZEBU_Fwc_init
ZEBU_Fwc_addValueSet csa_CORE1_CORE2
ZEBU_Fwc_setOutputDir ./zrun_CSA.ztdb
ZEBU_Fwc_dumpOnOff "on"
...
ZEBU_Fwc_dumpOnOff "off"
ZEBU_Fwc_flush
ZEBU_Fwc_close
```

To enable ZTDB auto-slicing for QiWC, use the following command only in batch mode:

```
ZEBU_Fwc_setSlicingParameters (-total_samples=NUM) (-slices=NUM)
```

2.4.5 Using C/C++ Testbench for Debug

This section describes the following:

- [Controlling Dynamic-Probes Waveforms Using C/C++](#)
- [Controlling FWC and QiWC Using C/C++](#)

2.4.5.1 Controlling Dynamic-Probes Waveforms Using C/C++

When using a C-COSIM transactor, the Dynamic-Probes waveforms can be controlled through the C and C++ Board API.

Example

```
//some code in Main function
Board *zebu    = NULL;
try {
    zebu = Board::open(ZEBUWORK);
    zebu->loadSelectionDB("path_to/csa_supports.zrdb");
    zebu->dumpfile("dynamic_probes.ztdb");
    zebu->dumpvars();
    // Enable ZTDB Slicing for Waveform Capture with zWaveform
    zebu->dumpSetRegularZtdbFullFrameInterval(200);
    zebu->dumpon();
//some code
    zebu->dumppoff();
//some code
    zebu->dumpon();
//some code
    zebu->dumppoff();
    zebu->dumpclosefile();
}
//board closed
```

If the design was compiled in dual-edge mode, the waveforms are sampled on both rising and falling edges of the C-COSIM clock.

For more details, see the ZEBU_Board.h or Board.hh header files.

When NOT using a C-COSIM transactor, the Dynamic-Probes waveforms can be controlled through the C and C++ WaveFile APIs.

Example

```
//some code in a thread
    rb0 = new WaveFile(zebu,
                      "dynamic_probes.ztdb",
                      "hw_top.clk", 0);
    rb0->dumpvars();
    // Enable ZTDB Slicing for Waveform Capture with zWaveform
    rb0->setRegularZtdbFullFrameInterval(200);
    rb0->dumpon();
    rb0->dumppoff();
    rb0->flush();
//some code
    rb0->dumpon();
    rb0->dumppoff();
    rb0->flush();
//some code
    rb0->close();
    delete(rb0);
}
//board closed
```

In this example, the waveforms are sampled on the rising edge of "hw_top.clk".

You can indicate dual-edge sampling by specifying "posedge hw_top.clk or negedge hw_top.clk" to the Wavefile constructor.

For more details, see the ZEBU_WaveFile.h or WaveFile.hh header files.

2.4.5.2 Controlling FWC and QiWC Using C/C++

Both mechanisms use the C and C++ FastWaveformCapture APIs.

Example

```
FastWaveformCapture * waveform;
//some code
    waveform = new FastWaveformCapture;
    //board opened
    waveform->initialize (zebu);
    waveform ->add ("csa_CORE1_CORE2");
    // Specify how to apply ZTDB Slicing mechanism for Waveform
    Reconstruction with zWaveform:
        // ZTDB Slice every 4 seconds:
        // waveform->SetRegularZtdbFullFrameInterval(4);
        // or apply auto-slicing:
        ZEBU_SlicingParameters Params;
        params.totalEdgeCount = 3000000;
        params.maxSliceCount = 50;
        fwc_obj->setSlicingParameters(params);
        // The dumpFile method starts dump.
        waveform ->dumpFile ("CSA.ztdb");
//some code
    waveform ->disable(); // The disable method stops the dump.
//some code
    // The enable method restarts the dump.
    waveform ->enable();
//some code
    // The close file method stops and closes the ztdb waveform.
    waveform ->closeFile();
    //board closed
    delete (waveform);
```

The waveforms are sampled on all edges of all primary clocks (composite clock).

For more details, see the `ZEBU_FastWaveformCapture.h` or `FastWaveformCapture.hh` header files.

2.5 Waveform Viewing and Analysis

If the waveforms are generated with Value-Sets that specify signals, such as Essential signals, it is recommended to use the Direct Viewing method. For more information, see [Viewing Captured Signals](#).

If the waveforms are generated with Value-Sets that specify design instances, Waveform Reconstruction using the **zCSA** or **zWaveform** reconstruction engine is required. For more information, see [Waveform Reconstruction](#).

It is recommended to use Interactive Waveform Reconstruction to reconstruct the waveforms within the Verdi GUI when you display a signal's waveform of about a 100,000 cycles. Otherwise, you can use Batch CSA or **zWaveform** to reconstruct all waveforms.

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

2.5.1 Viewing Captured Signals

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

1. Launch **zConvertToFsdB** or **zWaveform** as follows:

```
zConvertToFsdB \  
--ztdb Essential_Signals.ztdb \  
--zebu-work path_to/zcui.work/zebu.work \  
--fsdb Essential_Signals \  
--timescale 2ps \  
--command "<remote command>" \  
--jobs <User entered specified values, ex: 40> \  
--log Essential_Signals_zConvertToFsdB.log
```

Waveform Viewing and Analysis

```

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

```

Note that the generated FSDB file `Essential_Signals.vf` and the `Essential_Signals` directory.

For best performance:

- ☐ Set the number of FPGAs used at compilation as the value for the `-j` option.
- ☐ Use the `--command` option with the `qsh` or `lsf` commands to use a compute farm.

For more details, use the `-h` option of **zConvertToFsd** and **zWaveform**.

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

```

nWave -ssf $(RUNDIR)/Essential_Signals.vf

```

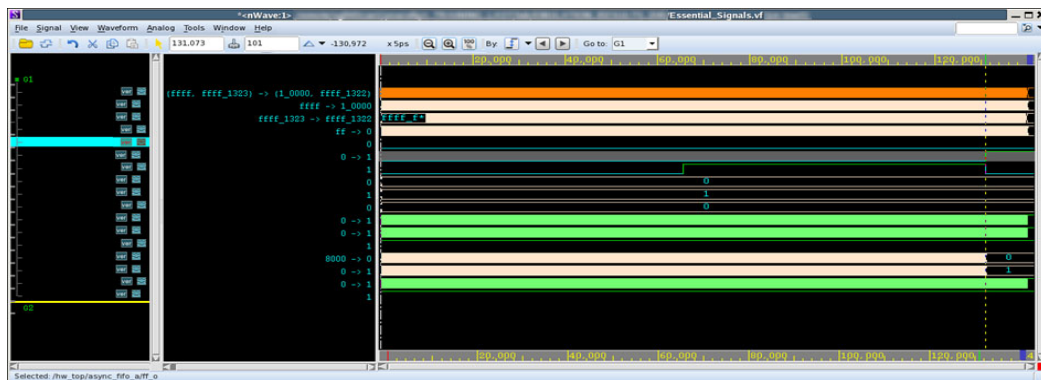


FIGURE 8. nWave Interface

It is possible to launch **zConvertToFbdb** and **zWaveform** while the waveform output is ongoing at Emulation runtime. For long emulation runtimes, this makes it possible to generate the FSDB to track the waveform progress.

2.5.2 Waveform Reconstruction

zWaveform is the new waveform reconstruction engine. It is **faster** and much more scalable than CSA. Its scalability resides the number of parallel jobs that can be up to the number of the ZTDB slices.

The legacy waveform reconstruction engine is **zCSA**.

Both waveform reconstruction engines are available for Dynamic-Probes, FWC or QiWC technologies. They reconstruct the waveforms of combinational signals contained within the captured instances.

There are two methods to reconstruct the waveforms:

- **zCSA** reconstructs the waveforms of signals or instances you specify. It provides both a GUI and a batch mode, using a project file or the command line.
- **zWaveform** reconstructs the waveforms of signals and/or instances. It is used with command line.
- Interactive Waveform Reconstruction 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*.

For more information, see the following subsections:

- [zWaveform](#)
- [Batch zCSA](#)
- [zCSA in GUI](#)
- [Reconstructing Waveforms](#)
- [Rerunning zCSA in Batch Mode](#)
- [Running Interactive CSA in Command Mode](#)

2.5.2.1 zWaveform

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>`
- QSSH: `-pe mt <Number of Cores>`

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

You can then launch the **zWaveform** command as specified in the following example:

```
zWaveform \
--ztdb captured_waveform.ztdb \
--work path_to/zcui.work/zebu.work \
--fsdb 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.

When using **zSimzilla** or **zWaveform**, you can reconstruct a subpart of hierarchies and signals that need to be included in the hierarchies captured in ZTDB.

In **zWaveform**, 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 **zSimzilla**, 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 `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.

2.5.2.2 Batch zCSA

You can launch **zCSA** from the command line as specified in the following example:

```
zCSA \  
--input <path to the .ztdb waveform> \  
--zebu-work <path to zebu.work directory> \  
--timescale <timescale, ex: 2ps> \  
--nogui
```

Note

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

2.5.2.3 zCSA in GUI

You can also reconstruct the waveforms using the **zCSA** GUI (displayed in the following figure when used in conjunction with Dynamic-Probes).

The following graphical interface appears.

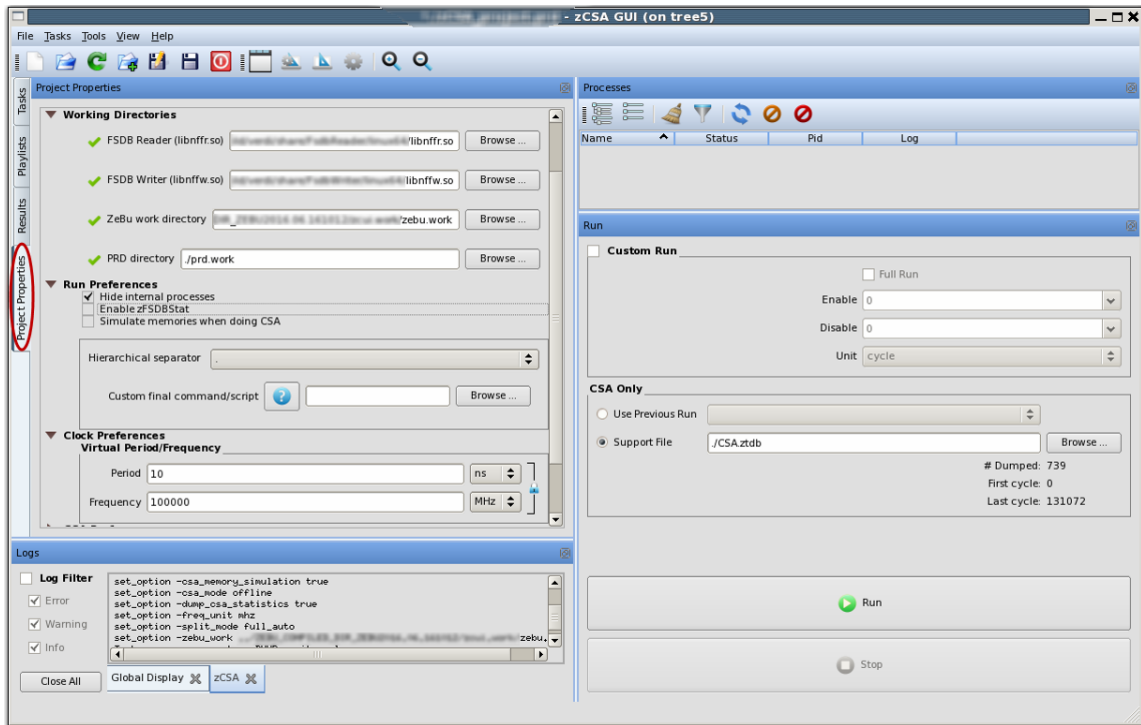


FIGURE 9. zCSA GUI

The **zCSA** GUI allows you customize the waveform reconstruction process, as follows:

- Working Directories
- Run Preferences
- Clock Preferences
- CSA Preferences
- Job Scheduling

Working Directory

The following figure displays the options available in the **Working Directories** section.

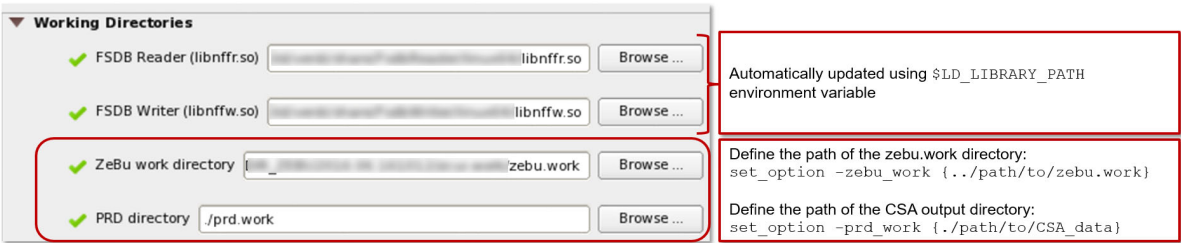


FIGURE 10. Working Directory in Project Properties Panel

The following table lists the options available in the **Working Directory** section.

TABLE 1 Working Directory Options and Tcl Commands

Options	Description
FSDB Reader/FSDB Writer	Automatically updated using \$LD_LIBRARY_PATH environment variable
ZeBu work directory	Allows you to define the path of the Zebu work directory, or use the following command to define the path. set_option -zebu_work {../path/to/zebu.work}
PRD directory	Allows you to define the path of the PRD directory, or use the following command to define the path. set_option -prd_work {../path/to/CSA_data}

Run Preferences

The following figure displays the options available in the **Run Preferences** section.

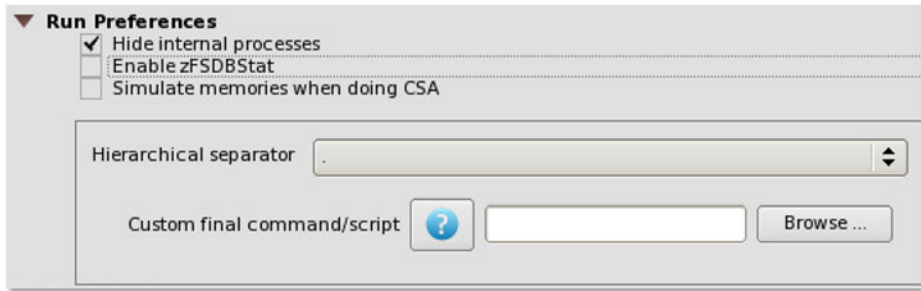


FIGURE 11. Run Preferences in Project Properties Panel

The following table lists the options available in the **Run Preferences** section.

TABLE 2 Run Preferences Options and Tcl Commands

Options	Description
Hide Internal Processes check box	Allows you to limit the verbosity. The Tcl command for this check box is: <code>set_option -hide_internal_processes {true}</code>
Enable zFSDBStat check box	Enables the generation of FSDB statistics for each run. The Tcl command for this check box is: <code>set_option -dump_csa_statistics {true}</code> .
Simulate memories when doing CSA check box	Enable the simulation of the memories (only reconstruct write accesses within the dumped r). The Tcl command for this check box is: <code>set_option -csa_memory_simulation true}</code> .

TABLE 2 Run Preferences Options and Tcl Commands

Options	Description
Hierarchical separator list	Allows you to select the separator (.) for all instance paths indicated in the tasks. The Tcl command for this option is: set_option -separator {.}.
Custom final command/script	Specifies a Custom final command or script file. Here is an example for this option. set_option -custom_final_script {/path/to/executable/file %r %z %t %pid %uid %verdi %vp %unit} Where: <ul style="list-style-type: none">• %r is the Run directory• %z is the zebu.work directory• %t is Top Level name• %pid is the pid• %uid is the uid• %verdi is the Verdi kdb

Clock Preferences

You can set the virtual period/frequency of the waveforms using the **Clock Preferences** section.

The following figure displays the options available in the **Clock Preferences** section.

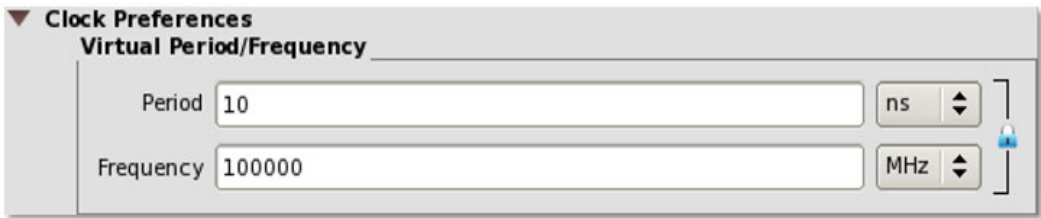


FIGURE 12. Clock Preferences in Project Properties Panel

You can type the virtual period of a waveform or use the following commands.

```
set_option -virtual_period {10}  
set_option -time_unit {ns}
```

CSA Preferences

The following figure displays the options available in the **CSA Preferences** section.

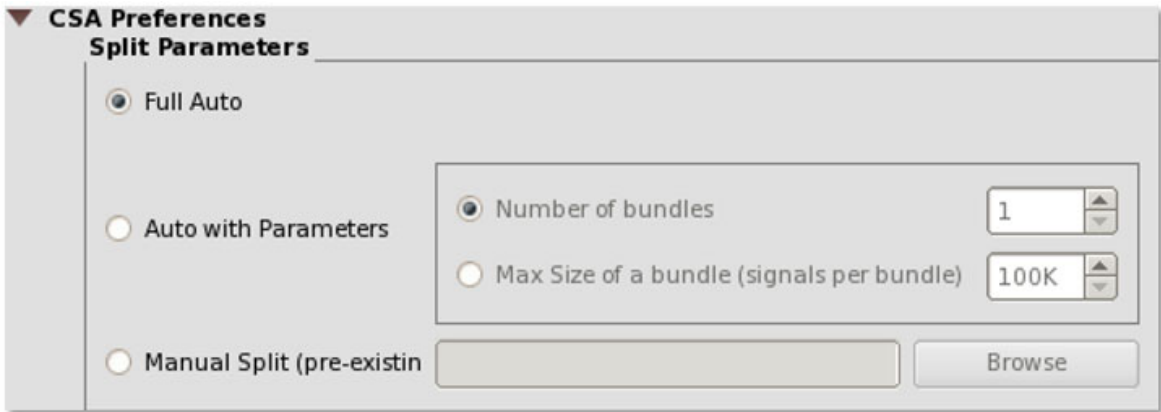


FIGURE 13. CSA Preferences in Project Properties Panel

You can use the following options to enable the CSA parallelism feature.

- Full Auto
- Auto with parameters
- Manual split (pre-existing)

The following table lists the options and the Tcl commands.

TABLE 3 CSA Preferences Options and Tcl Commands

Options	Description
Full Auto	Enables the CSA engine to automatically handle the split. This option is selected by default. The Tcl command for this option is: <code>set_option -split_mode {full_auto}</code>
Auto with Parameters	Handles the splits using the number of bundle or the size of each bundle. The Tcl command for this option is: <code>set_option -split_mode {auto_params}</code>

TABLE 3 CSA Preferences Options and Tcl Commands

Options	Description
Number of bundles	Allows you to specify the number of bundles. The Tcl command for this option is: <code>set_option -nb_split_csa {<number>}</code>
Max size of a bundle (signals per bundle)	Allows you to specify the size of a bundle. The Tcl command for this option is: <code>set_option -split_core_cost {<number>}</code>
Manual Split (preexisting)	Allows you to specify the bundle hierarchically to the CSA engine. The Tcl command for this option is: <code>set_option -split_file {/path/to/BundlesDefinedByHierarchies/file.lst}</code>

Example

Here is an example of a hierarchy file (`file.lst`).

```
hw_top
hw_top.async_fifo_a
hw_top.async_fifo_a2
```

The generated bundles contain "-i" for the included hierarchies and "-x" for the excluded hierarchies.

Bundle #0:

```
-i hw_top.async_fifo_a2
```

Bundle #1:

```
-i hw_top.async_fifo_a
```

Bundle #2:

```
-i hw_top
-x hw_top.async_fifo_a
-x hw_top.async_fifo_a2
```

Job Scheduling

The following figure displays the options available in the **Job Scheduling** section.

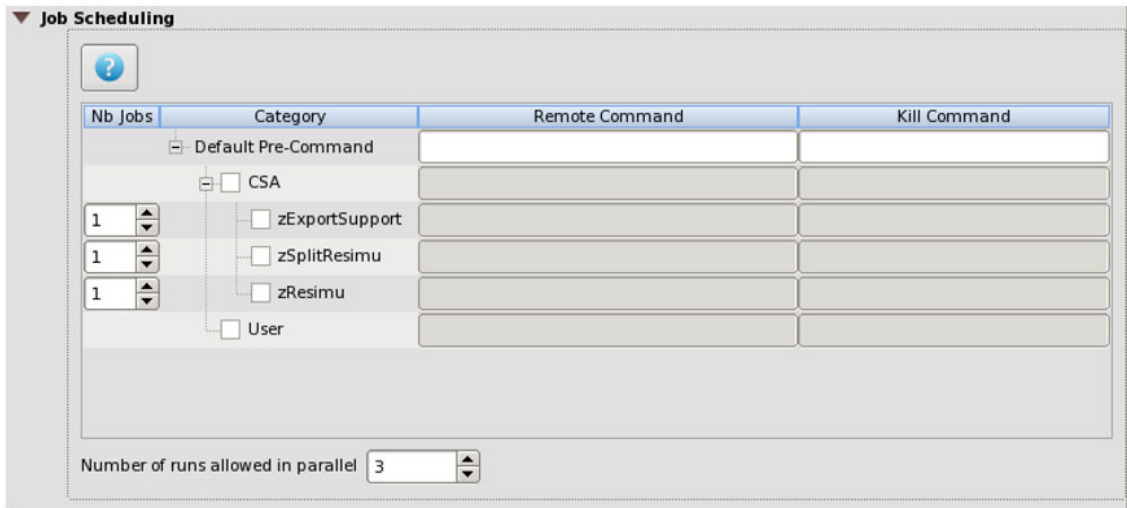


FIGURE 14. Job Scheduling in Project Properties Panel

The following table lists the options available in the **Job Scheduling** section.

TABLE 4 Options Available in the **Job Scheduling** Section

Options	Description
Nb Jobs column	Enables you to enter the maximum number of jobs launched in parallel. The Tcl command is: <pre>set_option -pre_zexport_cmd_nb_job {<maximum number of jobs launched in parallel>}</pre>
Category column	
Remote Command column	Allows you to define the remote Unix command setup. The Tcl command is: <pre>set_option -pre_global_cmd {<remote command: qysh/lsh/...>}</pre>

TABLE 4 Options Available in the **Job Scheduling** Section

Options	Description
Kill Command column	Allows you to define general kill command setup. The Tcl command is: <pre>set_option -pre_global_kill {<kill command: kill -9, ...>}</pre>
Default Pre-Command list	
CSA check box	Enables you to override the remote command: <pre>set_option -use_csa_cmd {true}</pre> <p>If this check box is selected, the Remote Command and the Kill Command columns are updated as follows:</p> <ul style="list-style-type: none"> • Remote Command column: <pre>set_option -pre_csa_cmd {<remote command: qrsh/lsf/...>}</pre> • Kill Command column: <pre>set_option -pre_csa_kill {<kill command: kill -9, ...>}</pre> <p>If this check box is not selected, the general remote command setup is used.</p>
zExportSupport check box	Enables you to override the remote command. The Tcl command is: <pre>set_option -use_zexport_cmd {true}</pre> <p>Supports Tcl way of defining RTL paths.</p> <p>If this check box is selected, the Remote Command column and the Kill Command column are updated as follows:</p> <ul style="list-style-type: none"> • Remote Command column: <pre>set_option -pre_zexport_cmd {<remote command: qrsh/lsf/...>}</pre> • Kill Command column: <pre>set_option -pre_zexport_kill {<kill command: kill -9, ...>}</pre> <p>If this check box is not selected, the general remote command setup is used.</p>

TABLE 4 Options Available in the **Job Scheduling** Section

Options	Description
zSplitResimu check box	<p>Allows you to perform CSA split for parallelism of bundles. The Tcl command is:</p> <pre>set_option -pre_zsplit_cmd_nb_job {<maximum number of jobs launched in parallel>} set_option -use_zsplit_cmd {true}</pre> <p>If this check box is selected, the Remote Command column and the Kill Command column are updated as follows:</p> <ul style="list-style-type: none"> • Remote Command column: <pre>set_option -pre_zsplit_cmd {<remote command: qrsh/lsf/...>}</pre> • Kill Command column: <pre>set_option -pre_zsplit_kill {<kill command: kill -9, ...>}</pre> <p>If this check box is not selected (Tcl command "use_zsplit_cmd {false}"), the following commands are used:</p> <ul style="list-style-type: none"> • pre_zsplit_cmd • pre_zsplit_kill
zResimu check box	<p>Allows you to perform CSA split for parallelism of bundles. The Tcl command is:</p> <pre>set_option -pre_resimu_cmd_nb_job {<maximum number of jobs launched in parallel>} set_option -use_resimu_cmd {true}</pre> <p>If this check box is selected, the Remote Command column and the Kill Command column are updated as follows:</p> <ul style="list-style-type: none"> • Remote Command column: <pre>set_option -pre_resimu_cmd {<remote command: qrsh/lsf/...>}</pre> • Kill Command column: <pre>set_option -pre_resimu_kill {<kill command: kill -9, ...>}</pre> <p>If this check box is not selected, the general remote command setup is used.</p>

TABLE 4 Options Available in the **Job Scheduling** Section

Options	Description
Users check box	<p>Allows you to launch user scripts/commands on the compute farm. The Tcl command is:</p> <pre>set_option -use_user_cmd {true}</pre> <p>If this check box is selected, the Remote Command and the Kill Command setups are used as follows:</p> <ul style="list-style-type: none"> • Remote Command column: <pre>set_option -pre_user_cmd {<remote command: qrush/lsf/...>}</pre> <ul style="list-style-type: none"> • Kill Command column: <pre>set_option -pre_user_kill {<kill command: kill -9, ...>}</pre> <p>If this check box is not selected, the general remote command setup is used.</p>
Number of runs allowed in parallel list box	<p>Allows you to limit the total number of jobs executing in parallel. The Tcl command is:</p> <pre>set_option -nb_run {<maximum of Total number of above jobs launched in parallel>}</pre>

2.5.2.4 Reconstructing Waveforms

To reconstruct the waveforms, perform the following steps:

1. Launch **zCSA** graphical interface with the following command:

```
zCSA
```

2. In the **Project Properties** panel, perform the following steps in the **Working Directories** section:
 - a. Observe that the **FSDB Reader** and **FSDB Writer** fields are automatically updated using \$LD_LIBRARY_PATH environment variable.
 - b. Click **Browse** to define the path of the **Zebu work directory** or use the following command to define the path.


```
set_option -zebu_work {../path/to/zebu.work}
```
 - c. Click **Browse** to define the path of the **PRD directory** or use the following command to define the path.


```
set_option -prd_work {../path/to/CSA_data}
```
3. In the **Project Properties** panel, perform the following steps in the **Run Preferences** section.

- a. Select the **Hide internal processes** check box.
This limits the verbosity.
- b. Select the **Enable zFSDBStat** check box.
This enables the generation of FSDB statistics for each run.
- c. Select the **Simulate memories when doing CSA** check box.
This enables the reconstruction of memories (only reconstructs write accesses within the displayed window).
- d. Select the **Hierarchical separator** as ". (DOT)" from the list.
This is separates instance paths.
- e. Click **Browse** to specify the **Custom final command/script** file.

NOTE: *The command/script file is executed at the end of each CSA run.*

4. In the **Project Properties** panel, perform the following steps in the **Clock Preferences** section.
 - a. Specify the waveform virtual period in the **Period** box, and then select the unit for virtual period to define the virtual period and its unit.
 - b. Observe that the frequency is automatically computed and updated in the **Frequency** box.
5. In the **Project Properties** panel, perform the following steps in the **CSA Preferences** section to enable the CSA parallelism feature.
 - ☐ Observe that the **Full Auto** is selected by default so that the CSA engine is enabled to automatically handle the split.
 - ☐ Click **Auto with Parameters** to handle the splits using the number of bundle or the size of each bundle.
 - ◆ Click **Number of bundles** and then enter the number of bundle.

NOTE: *It is limited by the number of hierarchies.*

 - ◆ Click **Max Size of a bundle (signals per bundle)** and then enter the size of a bundle.
 - ☐ Click **Manual Split (pre-existing)**, and then click **Browse** to select the bundle hierarchically to the CSA engine.

NOTE: *The hierarchies must exist.*
6. In the **Project Properties** panel, perform the steps listed in the **Job Scheduling** section to schedule the number of runs allowed in parallel.
For more information about the options, see [Job Scheduling](#).

Creating a Task

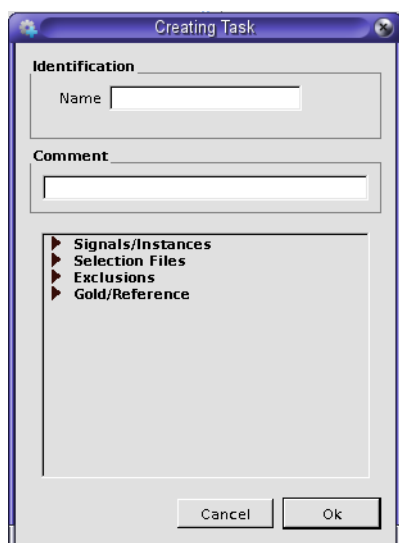
To create the task, perform the following steps:

1. Launch **zCSA** graphical interface with the following command:

```
zCSA
```

2. Click the **Tasks** panel.
3. Click the  icon.

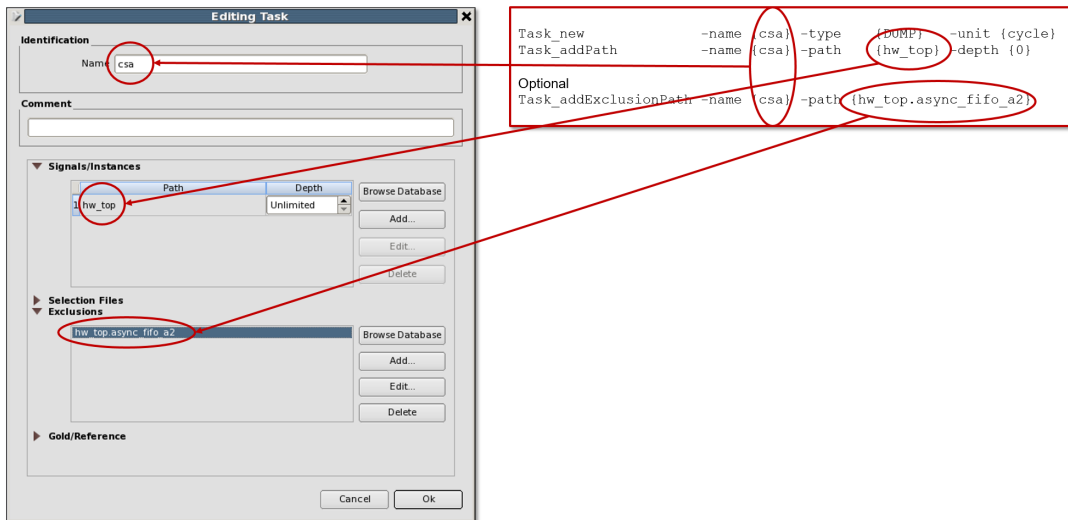
The **Creating Task** dialog box appears.



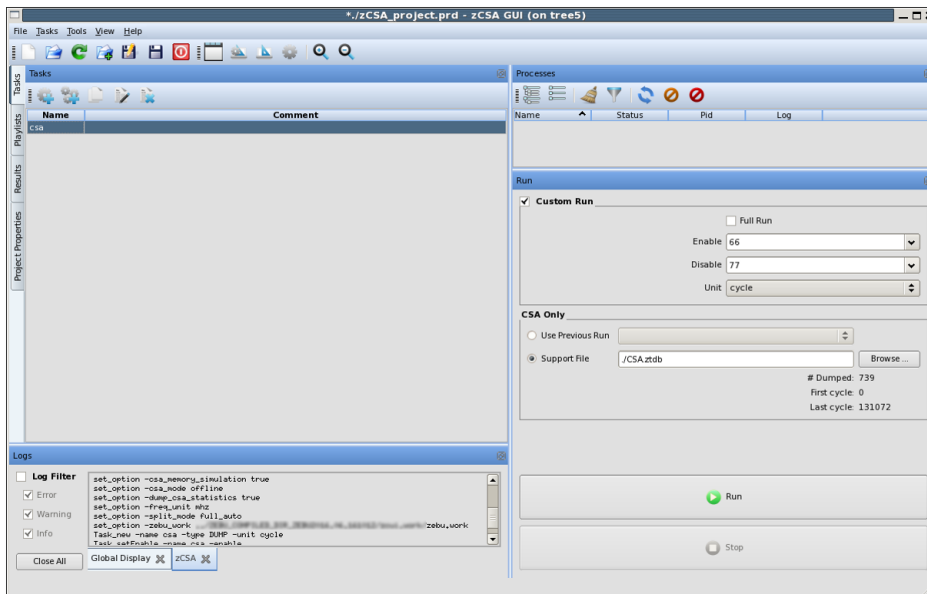
4. In the **Creating Task** dialog box, type a name for the task.
5. Expand **Signals/Instances**, and then click **Browse the Database** to select the instances whose waveforms you want to reconstruct.
6. Click **OK**.
7. If you want to edit the task, double-click the task or right-click the task and then click **Edit Task/Group**.

The **Editing Task** dialog box appears.

Waveform Viewing and Analysis



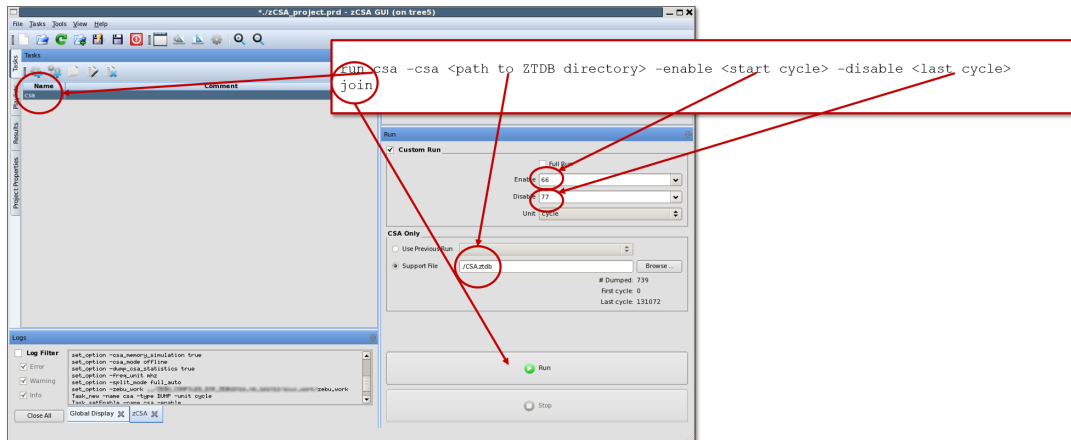
8. Select the task in the **Tasks** panel, then press **Run** in the **Run** panel.



NOTE: By default, the CSA is executed within the begin and end cycles of the ZTDB waveform. Alternatively, you can change the begin and end cycles (in the ZTDB waveform window) using the following command: `run csa -csa <path`

to ZTDB directory> -enable <start cycle> -disable <last cycle> join.

9. Select the **Custom Run** check box to update the cycles in the **Enable** and **Disable** list.

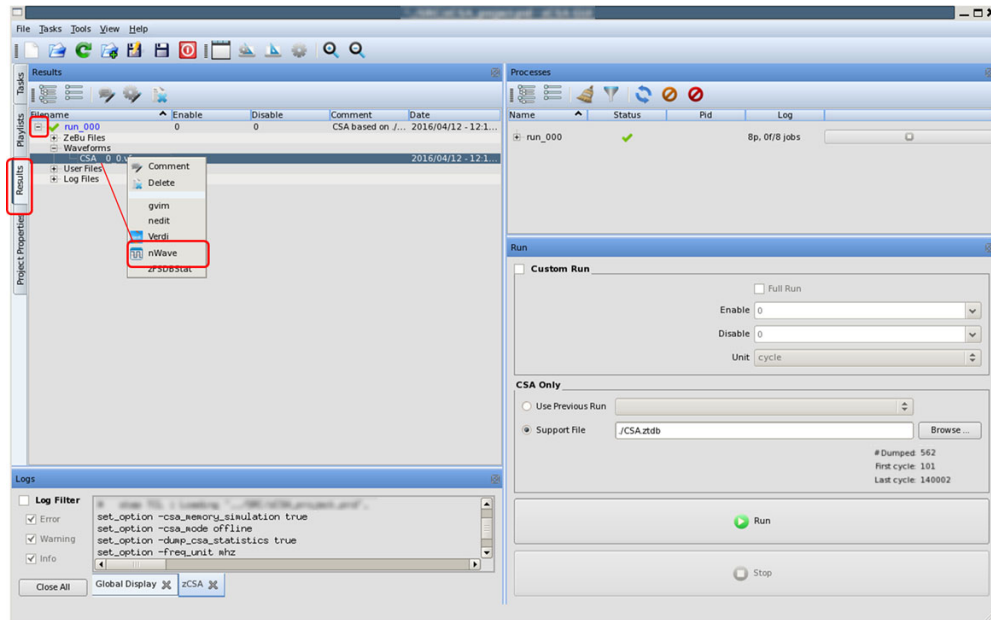


10. In the **Process** pane, you can track the CSA progress.

11. To view the results:

- a. In the **Results** panel, expand the `run_xxx/Waveform` results, then right-click the **Virtual File** to see the Waveforms.

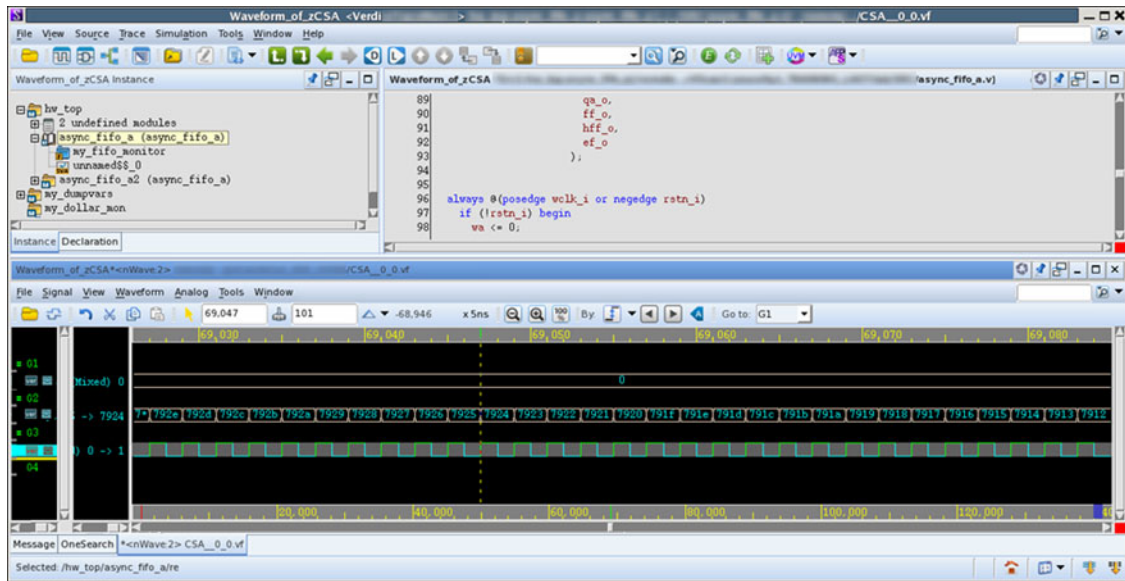
Waveform Viewing and Analysis



or

b. Launch Verdi:

```
verdi prd.work/run_<xxx>/CSA__0_0.vf \
-simflow \
-dbdir <path_to>/zcui.work/vcs_splitter/simv.daidir
```



2.5.2.5 Rerunning zCSA in Batch Mode

After you have used **zCSA** with the GUI and saved the project, you can rerun the **zCSA** project in batch mode. To do so:

1. Uncomment the following line from the generated project in the directory `<prd.work>/run_<run number>_<id>/zcsa.prd`:

```
# run_full_QiWC
# join
```

2. Reuse the file to run **zCSA** in batch mode with the exact same settings.

NOTE: `zCSA --help TCL` can be used for viewing the Tcl commands applicable for **zCSA**.

2.5.2.6 Running Interactive CSA in Command Mode

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.

```
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
```

Where, you

- specify the path to the KDB directory - if it has been moved or created from the compilation flow using `-dbdir`

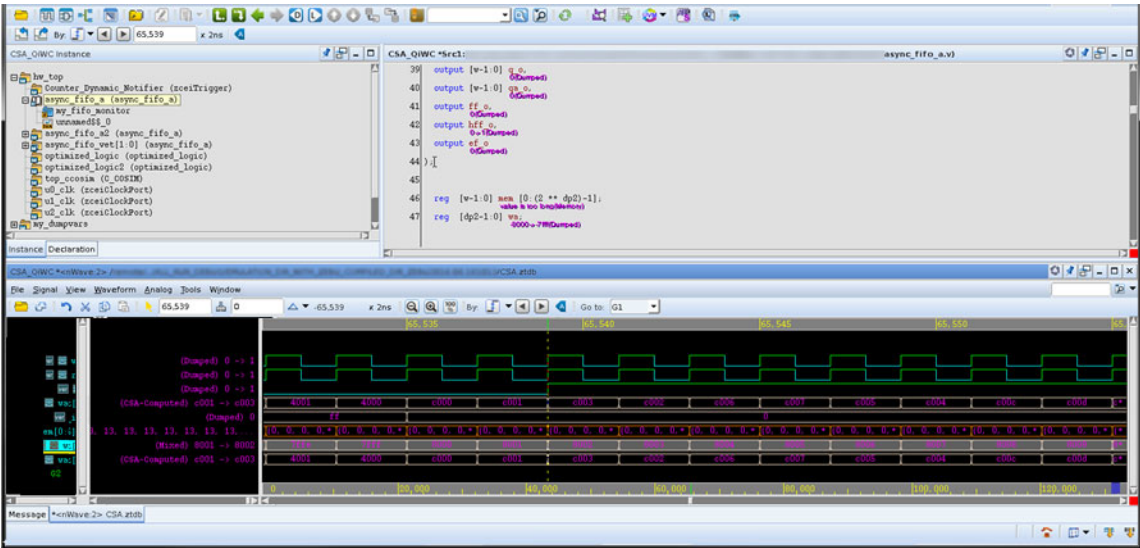


FIGURE 15. Interactive CSA With Verdi

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. It uses **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 also uses the **Sniffer** technology.
- Output waveforms using any Waveform Capture technology during rerun.
- Run CSA on the generated ZTDB waveforms.
- Bypass the testbench (including transactors) when rerunning.

Before using the stimuli replay technology, you need to compilation setup is required. For more information, see the [Compilation Setup for Stimuli Replay](#) section.

During the first emulation runtime, you use the **Sniffer API**.

The following figure illustrates the data generated by the **Sniffer API**. For more information, see the [Initial Emulation Runtime for Sniffer](#) section.

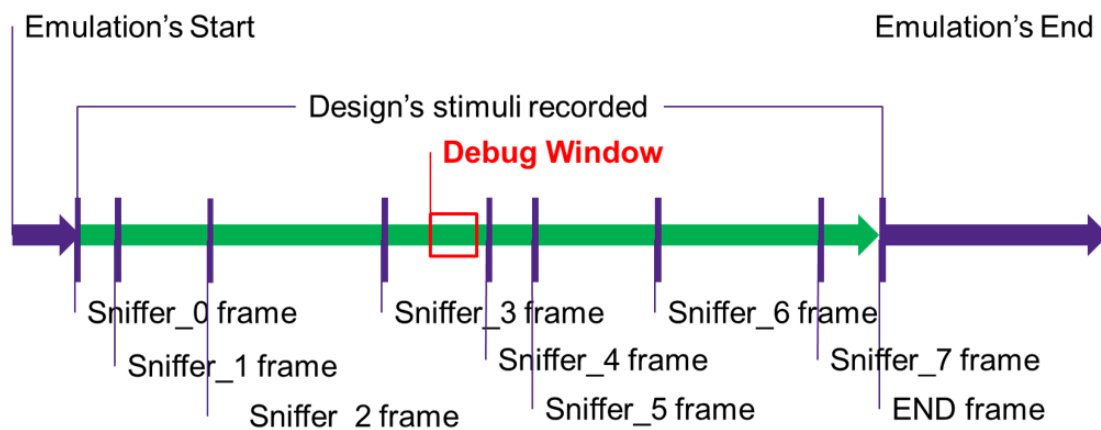


FIGURE 16. Global View During the Emulation Runtime

After using the **Sniffer API**, you can use **zPostRunDebug** to restore and rerun a snapshot.

The following figure illustrates the **zPostRunDebug** usage.

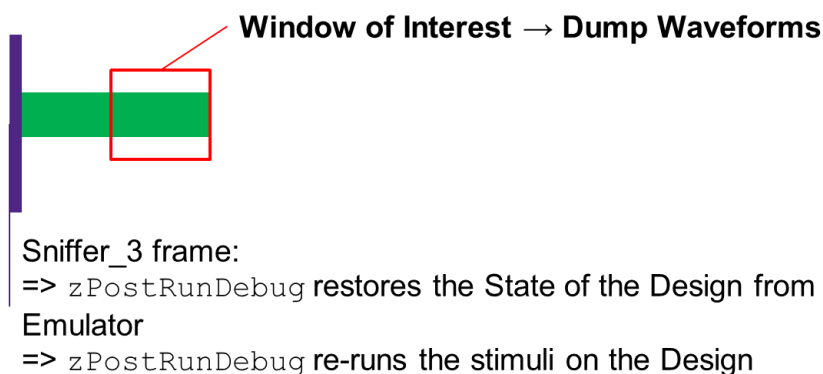


FIGURE 17. Overview of the Debug With zPostRunDebug

To use **zPostRunDebug**, the compilation must activate the rerun technology. For more information, see the [Using zPostRunDebug \(zPRD\) to Replay Stimuli](#) section.

This section describes the following subtopics:

- [Compilation Setup for Stimuli Replay](#)
- [Initial Emulation Runtime for Sniffer](#)
- [Using zPostRunDebug \(zPRD\) to Replay Stimuli](#)

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

3.2 Initial Emulation Runtime for Sniffer

The C and C++ **Sniffer APIs** are used to control the **Sniffer** technology at runtime. For more details, see the `ZEBU_Sniffer.h` and `Sniffer.hh` header files.

The following shows an example.

```
// some code in Main function
Board *zebu    = NULL;
Sniffer::Configuration  zPRD_config;
try {
    zebu                                = Board::open(ZEBUWORK);
    zPRD_config.folderName              = "zPostRunDebug_data_dir";
    zPRD_config.clockName               = "hw_top.clk";
    zPRD_config.sniffOutput             = true;
    //some code
    //start recording the emulation stimuli
    Sniffer::Start(zPRD_config); // starts & create a snapshot
    //some code
    Sniffer::CreateFrame(); // create a snapshot
    //some code
    Sniffer::CreateFrame(); // create a snapshot
    //some code
    Sniffer::Stop(zebu);
}
//board closed
```

3.3 Using zPostRunDebug (zPRD) to Replay Stimuli

zPostRunDebug and its graphical user interface is similar to the **zCSA** user interface. But, it is not intended for CSA only; its primary goal is to rerun stimuli.

zPostRunDebug is useful especially when:

- You need to investigate non-deterministic issues in the design.
- You do not want or cannot share the runtime environment.
- Generating waveforms takes longer than rerunning the window of interest.
- Generating waveforms takes more than one hour of Emulation.

zPostRunDebug can also be used to apply ZTDB slicing. The slicing is applied using the number of cycles per slice.

zPostRunDebug needs access to the emulator and the Sniffer directory `zPostRunDebug_data_dir`. ZeBu Server is used to rerun and typically generate waveforms. However, **zPostRunDebug** does not use ZeBu Server to run CSA.

Restoring frames to rerun, display waveforms, and run CSA are performed through tasks in the same way it is done with **zCSA**.

zPostRunDebug is NOT useful when you need to debug transactors or when the the emulation uses Direct ICE and Smart Z-ICE.

For more information, see the following subsections:

- [Using the zPostRunDebug GUI](#)
- [Creating a Task to Output Dynamic-Probes Waveforms and Run CSA Offline](#)
- [Creating zPostRunDebug Tasks for QiWC Output and CSA](#)
- [Creating zPostRunDebug Tasks for FWC Output](#)
- [Using zRci UCLI Commands for Stimuli Replay](#)

Note

`zPostRunDebug --helptcl` can be used for viewing the Tcl commands applicable for `zPostRunDebug`.

3.3.1 Using the zPostRunDebug GUI

The following figure displays the **zPostRunDebug** GUI.

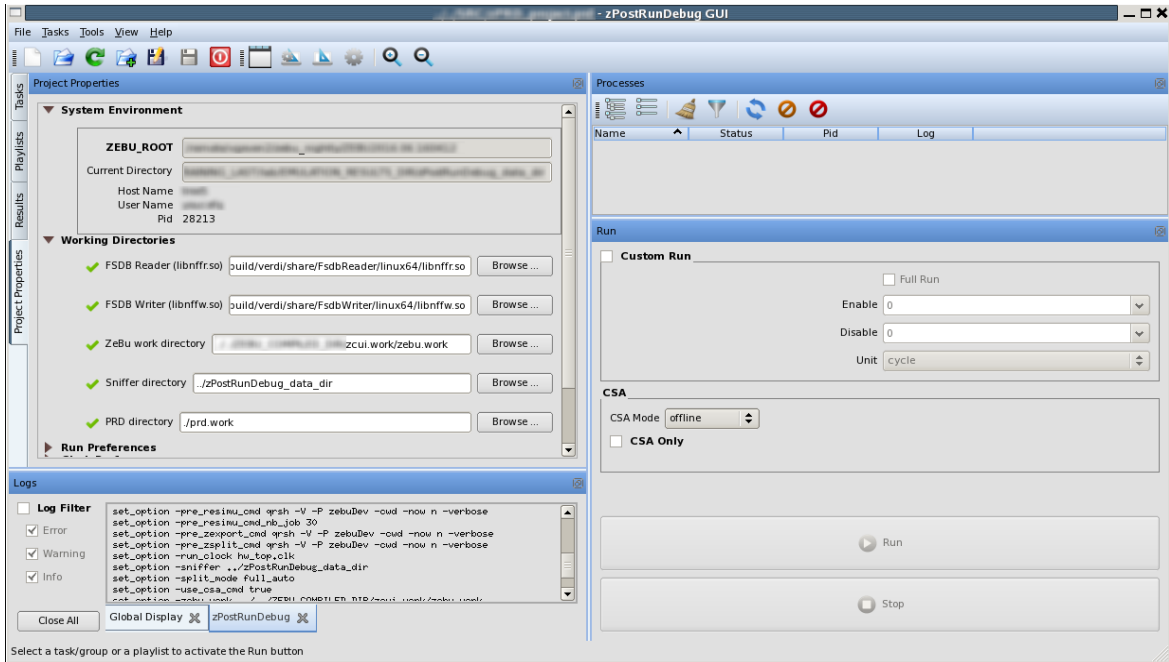


FIGURE 18. zPostRunDebug Main Window

1. Launch the **zPostRunDebug** graphical interface with the following command:

```
zPostRunDebug
```

For more details on options, run `zPostRunDebug -h`.

2. In the **Project Properties** panel, update if needed:
 - ☐ **FSDB Reader** and **FSDB Writer** fields (normally part of the LD_LIBRARY_PATH environment variable)
 - ☐ ZeBu work directory field
 - ☐ Sniffer directory

- ☐ **Virtual Period/Frequency** fields
- ☐ **Clocks/Sniffer**
 - ◆ Set sampling clock
 - ◆ See **Frames** details
- ☐ Precommands
 - ◆ If **zPostRunDebug** is launched directly from the Host PC, avoid any command in the **Default Pre-command** field
 - ◆ Select **CSA** and set a precommand to take advantage of the compute farm
- ☐ Increase the numbers of:
 - ◆ **zResimu** jobs
 - ◆ Number of runs allowed in parallel

3.3.2 Creating a Task to Output Dynamic-Probes Waveforms and Run CSA Offline

When using Dynamic-Probes to generate waveforms and run Combinational Signal Accessibility (CSA) offline, **zPostRunDebug** requires the creation of only one task.

To create a task, perform the following steps:

1. Click **Create Task**  in the **Task** panel.

The **Editing Task** window appears.

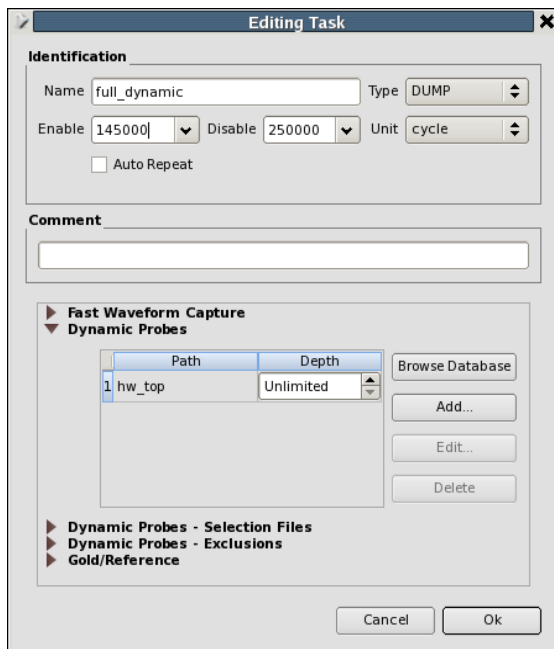
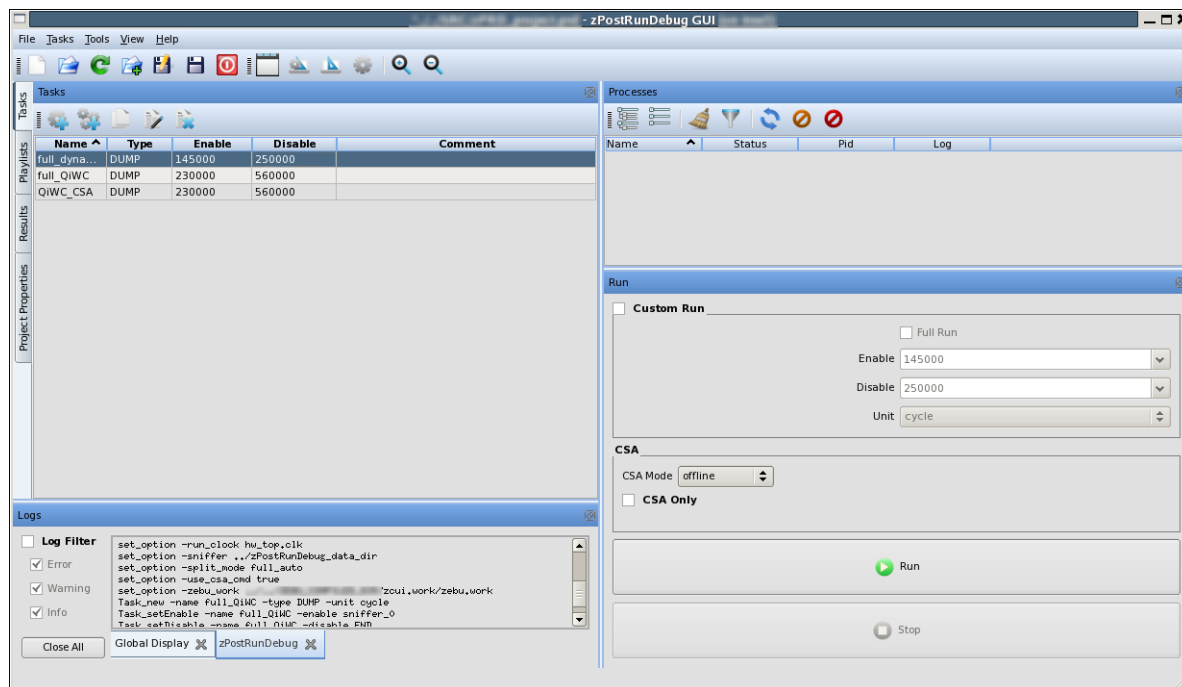


FIGURE 19. Dynamic-Probe Dump Task

2. Specify the name of the task.
3. Click **Browse Database** to select the instances for which you want to reconstruct the waveforms.
4. Click **OK**. The Editing Task window closes. The task appears in the **Tasks** panel.
5. Select the task in the **Tasks** panel, then click **Run** in the **Run** panel.
6. Track the CSA progress in the **Process** panel.

Using zPostRunDebug (zPRD) to Replay Stimuli

**FIGURE 20.** Dynamic-Probes Output with CSA

7. Set the CSA mode to **offline** in the **CSA** panel, .
8. Select the task and click **Run**.
9. See the results in the **Results** panel.

To apply ZTDB slicing, use the following Tcl command:


```
set_option -wave_slicing{size=<DUT clock cycles>}
```

3.3.3 Creating zPostRunDebug Tasks for QiWC Output and CSA

When using QiWC to generate waveforms and run CSA offline, **zPostRunDebug** requires two tasks:

- **1st Task:** Generate the waveforms
- **2nd Task:** Run CSA offline

Perform the following steps:

1. In the **Task** panel, click **Create Task** .

The **Editing Task** window appears.

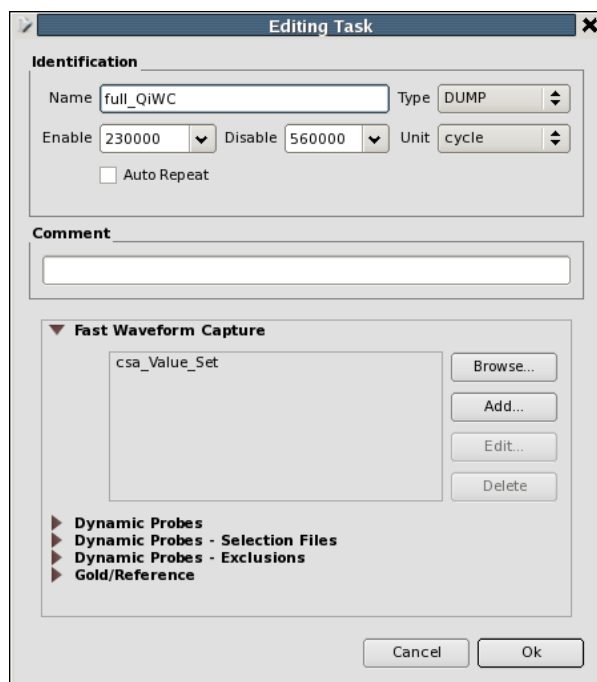
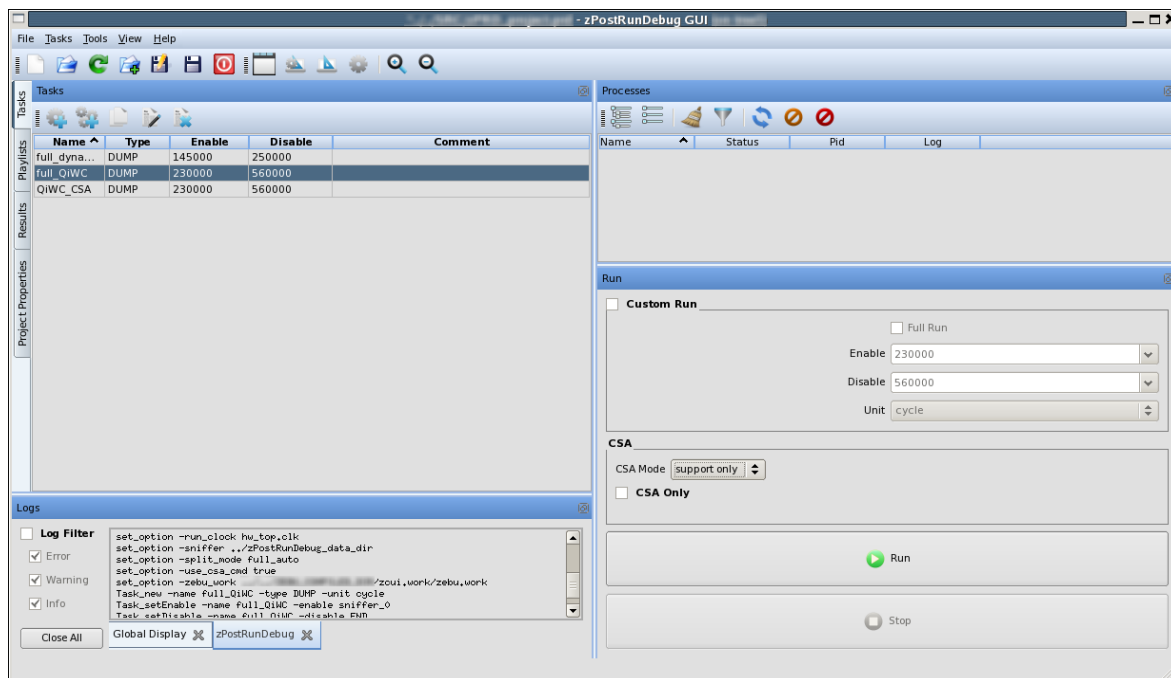


FIGURE 21. QiWC Dump Task

2. Specify the name of the task in the **Editing Task** window.

Using zPostRunDebug (zPRD) to Replay Stimuli

3. Browse the **Fast Waveform Capture** list, select the required QiWC Value-Set, and activate its CSA.
4. Click **OK**. The task appears in the **Tasks** pane.

**FIGURE 22.** QiWC Dump

5. Set the **CSA** mode to support only in the **CSA** panel.
6. Select the task and click **Run**.
7. See the results in the **Results** panel.

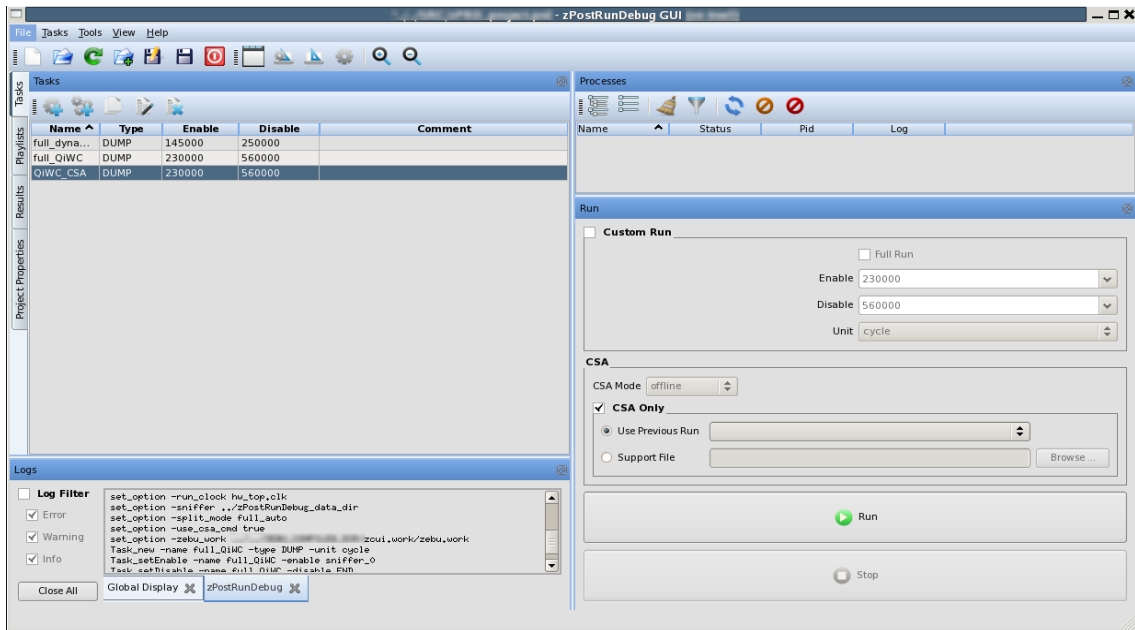


FIGURE 23. Activate CSA for QiWC Dump

- In the **CSA** panel, check **CSA Only**.
- Check **Use Previous Run** and select the one that corresponds to the QiWC waveforms.
- Select an existing dynamic-probe task or create a new one that specifies the hierarchy for which you want to run CSA (Ensure that the Value-Set(s) used for the capture and the CSA hierarchies are aligned).
- Select the task and click **Run**.
- Observe the results in the **Results** panel.

To apply ZTDB slicing, use the following Tcl command:

```
set_option -fwc_slicing{size=<DUT clock cycles>}
```

3.3.4 Creating zPostRunDebug Tasks for FWC Output

When using FWC to generate ZTDB waveforms, **zPostRunDebug** requires only one task.

You must create and generate the task in the same way when using QiWC (described in [Creating zPostRunDebug Tasks for QiWC Output and CSA](#)) by ensuring that you select only the Value-Set, which uses FWC.

In the **Results** panel, expand **Waveforms**, and see the file that has been automatically created. In the same way as with **zCSA**, a right-click allows you to open it with **nWave**.

3.3.5 Using zRci UCLI Commands for Stimuli Replay

You can now use **zRci** UCLI commands for debug technologies, such as Stimuli and Replay. The main UCLI commands are as follows:

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

With **zRci**, the Sniffer create frames for every <number> of cycles when only Blocking Runs are used.

The number of cycles are aligned with the default clocks used with **zRci**.

Example

```
sniffer -auto_create -cycles 1000000
...
run 1234
...
run 1002020290
...
```

For more details about the commands, see *Debug Commands* section in the *ZeBu Server Unified Command-Line User Guide*.

4 Capturing Using System Tasks

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 5 Top-Level Usage

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

TABLE 6 Special Cases

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

TABLE 6 Special Cases

\$dumpvars	\$dumpports
<code>\$dumpvars(1, dut.core.\escape[0] [31:0]);</code>	Range is not part of the vector name
<code>\$dumpvars(1, "dut.core.\escape[0] [31:0]");</code>	Range is not part of the vector name
<code>\$dumpvars(1, `MACRO_TOP.core.signal);</code>	Macro
<code>\$dumpvars(1, `"MACRO_TOP.core.signal`");</code>	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 triggered.

Example:

```
debug -dumpvars_maxbits 100
```


Specifying Maximum Bits for \$dumpvars/\$dumpports

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 into a warning using the VCS switch `-error=noFS_MAXBITS_EXCEEDED`.

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

5 Using Dynamic Trigger and Runtime Trigger

Dynamic triggers and runtime triggers are used in the debug flow to notify the impact 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 Trigger are used in the debug methodologies described in the *ZeBu Server Debug Methodology Guide*.

5.1 Dynamic-Trigger Technology

Dynamic-trigger technology enables you to choose the signals to connect to a dynamic-trigger at compile time.

At runtime, you can define an 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.

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

For more information, see the following:

- [Defining the Signals at Compile Time](#)
- [Defining a Value to Stop Runtime](#)

The *Batch/In Regression for zDPI Calls* debug methodology is described in the *ZeBu Server Debug Methodology Guide*.

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 define a value to stop the runtime by choosing the value that is a combination of the signal's bit. The value to stop the runtime can be defined using **RunManager** or **zRci**. See the following:

- [Stopping Runtime Using RunManager](#)
- [Stopping Runtime Using zRci](#)

Stopping Runtime Using RunManager

The following example explains on stopping the runtime using the **RunManager**.

```
Trigger *p_Counter_Dynamic_Trigger = zebu-
>getTrigger("hw_top.Counter_Dynamic_Trigger");
*p_Counter_Dynamic_Trigger="hw_top.async_fifo_a.counters_st.cycle_counter
_inc[31:0] == 32'd8";
run_manager->advanceClock("hw_top.clk", 2000, true,
*p_Counter_Dynamic_Trigger);
// Clocks stopped here, actions can be taken
```

For more details, see the `Trigger.hh` API.

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

Stopping Runtime Using zRci

The following example explains how to stop the runtime using the `stop zRci` 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
```

For more details on Debug UCLI Commands, see *ZeBu Server Unified Command Line User Guide*.

5.2 Runtime Trigger

At runtime, the testbench can be notified using a callback function or 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 C++ API `SwNotifier.hh` and the *ZeBu Server Unified Command-Line User Guide*.

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

This section describes the following subtopics:

- [Using Runtime Trigger With a CEL Module](#)
- [Using Runtime Trigger in C++ Testbench](#)
- [Using Runtime Triggers With zRci](#)

The debug methodology associated with Runtime Trigger is best suited when you need 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 Methodologies Guide*.

5.2.1 Using Runtime Trigger With a CEL Module

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
```

5.2.2 Using Runtime Trigger in C++ Testbench

To use the Runtime Trigger in the C++ testbench, a shared object must be created and given to the C++ object at runtime. To create the shared object, use the following command:

```
ce --cel_fsdb --software --compile --top hw_top --output SW_NOTIFIER ../
SRC/CEL/complex.cel
```

This command generates `SW_NOTIFIER/complex_cel_intf.so` to be used with the C++ object at runtime.

C++ Example

Here is an example to show the usage of Runtime Trigger with the C++ testbench.

```
#include <SwNotifier.hh>

//Call Back definition
void SWN_callback (const SWN::NotifyCb& NotifyCb_param) {
    if (NotifyCb_param.lastNotify)
    {
        cout << "  module      = " << NotifyCb_param.moduleName
<< endl;
        cout << "  stat_name   = " << NotifyCb_param.stateName
<< endl;
        cout << "  cycle      = " << NotifyCb_param.cycle
<< endl;
        cout << "  isNotify   = " << NotifyCb_param.isNotify
<< endl;
        cout << "  notifyId   = " << NotifyCb_param.notifyId
<< endl;
        cout << "  lastNotify  = " << NotifyCb_param.lastNotify
<< endl;
        cout << "  designClock = " << NotifyCb_param.designClockCycle
<< endl;
        cout << "-----"
<< endl;
        //take action
    }
}
```

```

    }
    else if (NotifyCb_param.isNotify)
    {
        cout << " Notification at cycle = " << dec << NotifyCb_param.cycle
<< endl;
        //take action
    }
    else
    {
        cout << " At stat_name      = " << NotifyCb_param.stateName
<< endl;
        cout << " At FSM at cycle = " << dec << NotifyCb_param.cycle
<< endl;
        //take action
    }
}

// in Main function
Captured_Data_For_SWN = new FastWaveformCapture ;
//if you do not need to capture QiWC waveform while using Runtime
Trigger, you can comment the following line to increase runtime
performance
Captured_Data_For_SWN->add ("my_qiwc_value_set");

SWN::SwNotifier *swn_object = new SWN::SwNotifier(zebu, "./
SW_NOTIFIER/complex_cel_intf.so", SWN_callback);
swn_object->attachFwc(Captured_Data_For_SWN, true);

waveform_CSA->dumpFile ("captured_data_for_swn.ztdb");
swn_object->start();

// run emulation

```


5.2.3 Using Runtime Triggers With 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 ""
}

#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
```

6 Debug-Related Limitations

This section describes debug-related limitations pertaining to the following:

- [Value-Sets](#)
- [Dynamic-Probes](#)
- [Emulation Runtime Speed During Capture](#)
- [Dynamic Trigger](#)

Value-Sets

Value-Set labels must be different from the enclosing SystemVerilog module name (ex: `hw_top`).

Dynamic-Probes

When using Dynamic-Probes, the following options of **zConvertToFsd** 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 During Capture

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

With **zRun**, when using several dynamic-triggers, only the OR between triggers is possible to get the clocks stopped when any dynamic-trigger is fired.

Example

```
ZEBU_Trigger_setExpr u_trig0 "top.trig0==1"
ZEBU_Trigger_setExpr u_trig1 "top.trig1==1"
ZEBU_Trigger_setExpr u_trig2 "top.trig2==1"
ZEBU_Trigger_setExpr u_trig3 "top.trig3==1"
ZEBU_Trigger_setExpr u_trig4 "top.trig4==1"
ZEBU_Trigger_setExpr u_trig5 "top.q5==32'h1000000"
ZEBU_LA_sot u_trig0 or u_trig1 or u_trig2 or u_trig3 or u_trig4 or
u_trig5
```

With **zRci**, when using several dynamic-triggers, you can enable them simultaneously or at any time. **zRci** stops the clocks when the first dynamic-trigger is fired.