Verification Continuum™

ZeBu® Debug Methodology Guide

Version S-2021.09-2, October 2023

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

About This Book

The ***ZeBu Debug Methodology Guide*** describes the methodologies that you can use for debugging and the available tools.

Intended Audience

This manual is written for engineers to assist them to debug designs targeted for emulation on the ZeBu system.

These engineers should have knowledge of the following Synopsys tools:

- ZeBu system (VCS, **zCui**, **zRci**, DPI, waveform reconstruction, and so on)
- Verdi

Contents of This Book

This document has the following sections:

Section	Describes
Introduction to Debug Technologies	Information on the debug technologies that you can use to debug your design
Debug Planning	Planning aides, such as checklists and debug methodology to requirements mapping.
Debug Methodologies	Five methodologies that leverage ZeBu debug technologies.

Related Documentation

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

Typographical Conventions

This document uses the following typographical conventions:

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

The following table describes the syntax used in this document:

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

Synopsys Statement on Inclusivity and Diversity

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

1 Introduction to Debug Technologies

ZeBu provides a host of technologies that you can use to debug your design.

Debugging in the ZeBu environment involves the use of the *ZeBu Runtime Control Interface (zRci)*. **`zRci`** captures raw data from ZeBu during runtime and saves the captured information in the ZTDB database file. The technology you use to debug the design depends on your requirements. For information on planning, see the Planning chapter. During the emulation runtime, with the testbench you can control key signals or get their values. For more information, see *Signal-Specific Information*.

Depending on the emulation compilation and runtime settings you have specified, you can use the ZTDB database to obtain the following debug-related information:

- **Waveforms**: Useful when you need to know how key signals are behaving, such as IP boundary, or when you need visibility on partial or the entire DUT. The information is generated in `zwd` format for Verdi. For more information, see *Waveform Capture and Reconstruction*.
- **DPI Calls**: Useful when you are observing DPI calls and SystemVerilog system tasks. The information is outputted as a log file. For more information, see *DPI Calls*.
- **Events to trigger**: Useful when you intend to use Dynamic Triggers and Runtime Triggers. The information is returned to the testbench and outputted as a log file. For more information, see *Event Detection Using Dynamic Trigger and Runtime Triggers*.
- **SVA assertions**: Useful when you are using SVA assertions. The information is outputted as a log file. For more information, see *SystemVerilog Assertions (SVA)*.
- **Stimuli-Replay**: Useful when you want to record and replay the stimuli sent to the design during emulation. For more information, see *Stimuli Replay*.

This section provides a brief overview of these technologies. See the following topics:

- *ZeBu Runtime Control Interface (zRci)*
- *Debug Technologies*

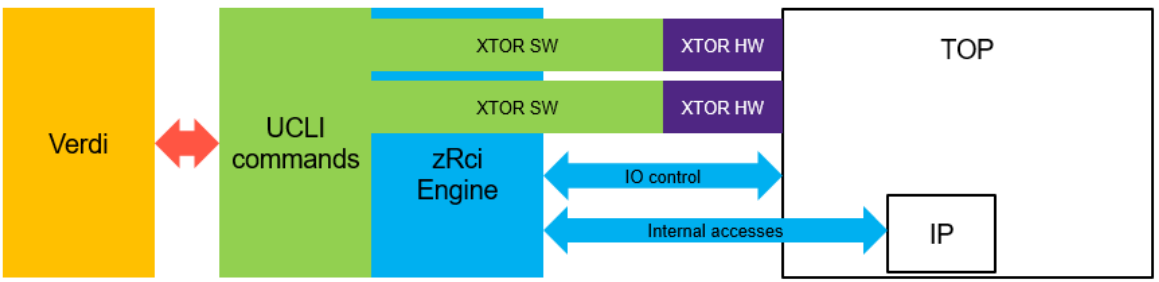
1.1 ZeBu Runtime Control Interface (zRci)

`zRci` provides a Tcl interface that you can use to interact with the emulation at runtime. Through the Tcl interface, you can specify control signals as per your requirement and consequently capture the information you need. The raw data from the emulator is stored in the ZTDB.

There are two methods of invoking `zRci`:

- **Batch/Interactive**: Unified command-line interface (UCLI) commands are used to interact with the emulation at runtime. The UCLI commands relevant for debugging are listed as part of the methodologies described in the *Debug Methodologies* section.
- **Verdi**: Verdi supports `zRci`. Apart from support through the Verdi GUI, a console is provided for you to enter UCLI commands.

The following diagram shows the interaction of Verdi, UCLI commands, and `zRci`.



For more information on `zRci`, see the ***ZeBu Unified Command-Line User Guide*** and the ***ZeBu-Verdi Integration Guide***.

1.2 Debug Technologies

This section describes the benefits of each debug technology and when to best use them. For more information, see the following subsections:

- *Signal-Specific Information*
- *Waveform Capture and Reconstruction*
- *DPI Calls*
- *Event Detection Using Dynamic Trigger and Runtime Triggers*
- *SystemVerilog Assertions (SVA)*
- *Stimuli Replay*

1.2.1 Signal-Specific Information

When you want to debug or control specific signals during emulation or when you want to observe specific signals from the testbench, use the following commands:

- **probe_signals** (Dynamic-probes/readback): Ensures visibility of the designated signal to read its value during runtime or output it into waveforms. It requires recompilation.
- **zForce**: Used to force a value on a variable and keep it until the force is released. The `zForce` command requires recompilation.

For more information on these commands, see the ***ZeBu Getting Started Guide*** and the ***ZeBu Unified Command-Line User Guide***.

Benefits and Considerations

The following table shows the benefits and considerations of using this approach:

TABLE 1 Capturing Signal-Specific information

Benefits	Considerations
Minor impact on hardware	Slow speed
Clocks are stopped whenever applied	

Associated Methodologies

- *Methodology 3: Applying Runtime Triggers to Capture and Expand Waveforms*

1.2.2 Waveform Capture and Reconstruction

You can use the waveform capture and reconstruction technologies to visualize waveforms of specific signals and for partial or full visibility of the DUT during emulation.

Use the following methods to capture waveforms:

- **Using FWC Technology**: This technology captures signal waveforms at runtime. It is used on key DUT signals only because of its impact on hardware resources and compile time. It is suitable for a large window of debug.

  To use FWC, you need to specify what to capture at compile time. This done using named "`begin…end`" blocks in the Verilog file.

  See the *Signal-Specific Information* section for more information on how to specify key signals.

- **Using Quick Waveform Capture (QiWC) technology**: Used on design instances. This is suitable for a large window of debug.

  To use QiWC, you need to specify what to capture at runtime. This done using named "`begin…end`" blocks to control the output at runtime.

- **Using Dynamic-probes/Readback technology**: Dynamic-probes are available by default on all the sequential cells. They can also be applied to any signal in the DUT. This method is suitable for a small window of debug. See the *Signal-Specific Information* section for more information on how to specify dynamic-probes to signals.

Benefits and Considerations

The following table shows the benefits and considerations of using each method:

TABLE 2 Capturing and Reconstructing Waveforms

Waveform capture method	Benefits	Considerations
Using FWC Technology on DUT key signals	The driver clock frequency can reach up to 2.9 MHz.	High impact on hardware resources and compile time
	Unlimited number of samples can be captured	
Using QiWC technology on design instances	The driver clock frequency can reach 45 Khz.	Medium impact on hardware resources and compile time
	Unlimited number of sample can be captured	
Using dynamic-probes/ readback technology on a part of the DUT or the whole DUT	No impact on HW resources and compile time	Slow waveform capture because the driver clock frequency is less than 100 Hz.

Associated Methodologies

- *Methodology 2: Capturing zDPI Information and the Associated Waveforms*
- *Methodology 3: Applying Runtime Triggers to Capture and Expand Waveforms*
- *Methodology 4: Stimuli Record With zDPI and Replay With Waveform*
- *Methodology 5: Prenotification Waveform*

1.2.3 DPI Calls

zDPI technology can be used either on-the-fly or offline. C functions are executed when using the on-the-fly mode and logs can be generated. ZTDB data is captured when using offline mode. After having run emulation, the C functions are applied while processing the ZTDB with `zdpiReport`.

This technology is also useful for capturing information by using SystemVerilog system tasks. Typically, **zDPI** is applied on internal DUT interfaces.

Benefits and Considerations

The following table shows the benefits and considerations of this approach:

TABLE 3 Using zDPI

zDPI method	Benefits	Considerations
On-the-fly zDPI	Results are available immediately	Impacts runtime because time is taken during disk access and software processing.
Offline zDPI	Smallest impact on runtime	

Associated Methodologies

- *Methodology 1: Batch/In Regression for zDPI Calls*
- *Methodology 2: Capturing zDPI Information and the Associated Waveforms*

1.2.4 Event Detection Using Dynamic Trigger and Runtime Triggers

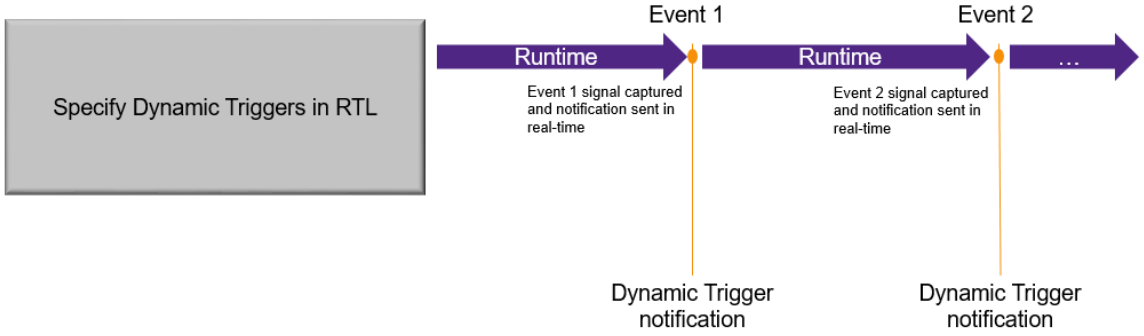
Events are represented as a list of signals compared with values. You can use the event detection technology to debug a single event (single cycle) or a sequence of events (state machine).

For more information, see the following subsections:

- *Dynamic Trigger*
- *Runtime Trigger*
- *Benefits and Considerations*

Dynamic Trigger

Dynamic triggers are useful for capturing a single event. Each Dynamic Trigger automatically stops the emulation runtime. Before using dynamic triggers, you need to identify the signals on which you want the trigger to be set. At runtime, when the event is triggered, the notification and trigger information is provided in real time and with no delay (See the following figure). If you add new signals, recompilation is required.



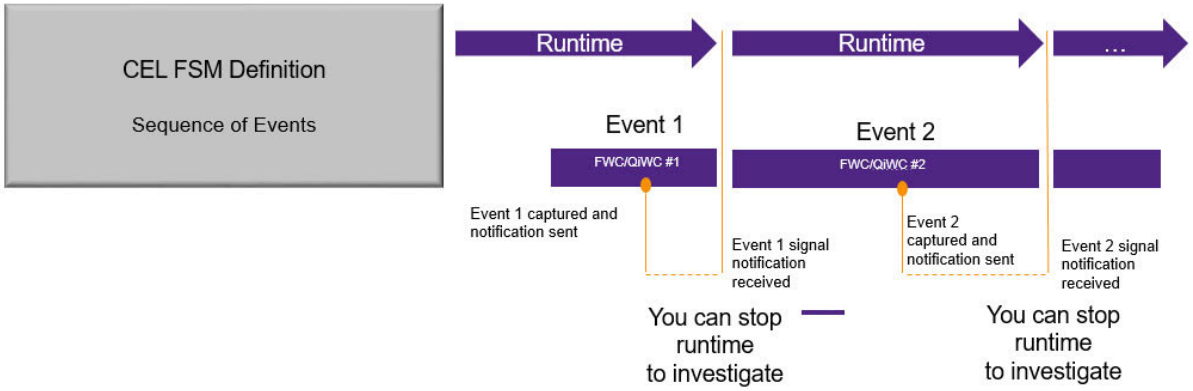
There is no theoretical limit to the number of signals used by a dynamic trigger (aka `zceiTrigger`). When using one trigger, the only allowed operation is AND. To achieve OR functionality, combine two different triggers.

At runtime, a limited set of logical operations on the different signals is supported.

For details on Dynamic Trigger, see the ***ZeBu Debug Guide***.

Runtime Trigger

Runtime triggers are useful for capturing a sequence of events. No special commands are required before compilation. At runtime, when the event is detected, the notification and trigger information is provided by a callback procedure to the testbench. The Runtime Trigger technology only captures FWC/QiWC bits. For QiWC, compilation is not required (See the following figure).



Benefits and Considerations

The following table shows the benefits and considerations of using this approach:

TABLE 4 Dynamic Trigger and Runtime Trigger

Event detection method	Benefits	Considerations
Dynamic Trigger	Cycle accurate	Selection of signals done at compile time
	Event or Trigger condition defined at runtime	
Runtime Trigger	Available online and offline	Runtime speed impacted by FWC/QiWC capture
	No recompilation required when signals are captured	

Associated Methodologies

- *Methodology 1: Batch/In Regression for zDPI Calls*
- *Methodology 2: Capturing zDPI Information and the Associated Waveforms*
- *Methodology 3: Applying Runtime Triggers to Capture and Expand Waveforms*
- *Methodology 5: Prenotification Waveform*

1.2.5 SystemVerilog Assertions (SVA)

Use this debug technology if you need to validate design behavior with SystemVerilog assertions (SVA). You can capture SVA on-the-fly or offline.

Benefits and Considerations

The following table shows the benefits and considerations of using this approach:

TABLE 5 SVA Debug

SVA method	Benefits	Considerations
On-the-fly SVA	Results are available immediately	Impacts runtime because time is taken during disk access and software processing.
Offline SVA	Smallest impact on runtime	

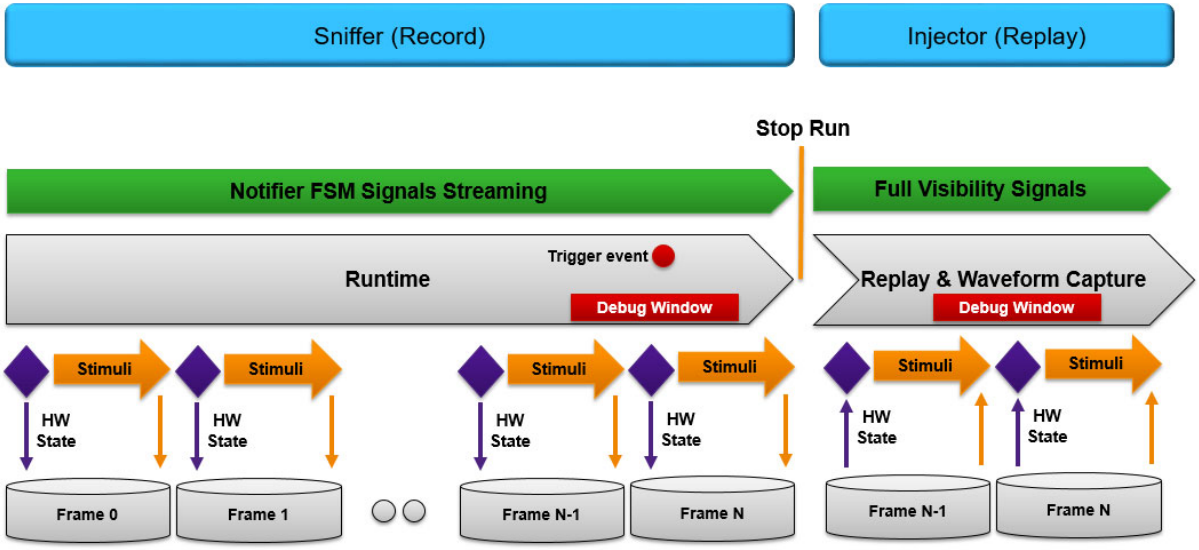
Associated Methodologies

Applicable to all methodologies listed in *Debug Methodologies*.

1.2.6 Stimuli Replay

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.

The following figure shows the Sniffer Replay technology.



Limitations exist when:

- Using Direct ICE - Smart ZICE
- Transactors need to be debugged
- Runtime uses several clock groups

Benefits and Considerations

The following table shows the benefits and considerations of using this approach:

TABLE 6 Stimuli Replay

Benefits	Considerations
Get rid of non-determinism from Transactors	Clocks are stopped when the hardware state is saved
Avoid rerunning billions of cycles to capture data	

Associated Methodologies

- *Methodology 4: Stimuli Record With zDPI and Replay With Waveform*
- *Methodology 5: Prenotification Waveform*

2 Debug Planning

The goal of debugging is to determine the root cause for an identified issue. Since emulators are a shared resource, the time required to determine the root cause is a key criteria for success. In addition, emulation test scenarios are long and extensive. As a result, the turnaround time for debug iteration is increased. Long test scenarios can also lead to data overload.

To address these challenges, careful debug planning is a necessity. This section enables you to choose a debug technology or a combination of the debug technologies based on your requirements. Debug requirements are mapped to each methodology.

For more information, see the following subsections:

- *Planning Checklist for Using Debug Methodologies*
- *Requirements Addressed by Each Debug Methodology*

2.1 Planning Checklist for Using Debug Methodologies

Use the following debug planning checklist to identify the debug methodology most suitable for your requirements. The **Keywords** column provides pointers to debug technology commands used to address a specific requirement. Additional information resources are also provided.

Requirement	Keywords	Support debug methodology
How to access signals, read and drive them during emulation?	**UTF**: `probe_signals, zForce, zInject` **zRci**: `get, force` For details, see the ***ZeBu User Guide***.	All methodologies
How to enable on-the fly and offline DPI calls?	**UTF**: `dpi_synthesis` **zRci**: `ccall` **Post-run**: `zdpiReport` For details, see the ***ZeBu User Guide***.	- *Methodology 1: Batch/In Regression for zDPI Calls* - *Methodology 2: Capturing zDPI Information and the Associated Waveforms*

Requirement	Keywords	Support debug methodology
How to capture and view key signals waveforms?	**RTL**: `(* fwc *)` `$dumpvars(...)` **zRci**: `dump -fwc` **Post-run**: `zSimzilla` For details, see ***ZeBu Debug Guide***.	- *Methodology 2: Capturing zDPI Information and the Associated Waveforms* - *Methodology 3: Applying Runtime Triggers to Capture and Expand Waveforms*
How to capture hierarchies and view the waveform of their signals?	**RTL for QiWC**: `(* qiwc *)` `$dumpvars(...)` **zRci**: `dump -qiwc` or `dump -dynamic_probe` **Post-run**: `/zSimzilla` For details, see the ***ZeBu Debug Guide***.	- *Methodology 4: Stimuli Record With zDPI and Replay With Waveform* - *Methodology 5: Prenotification Waveform*
How to capture a specific cycle-based event?	**RTL**: `zceitrigger` **zRci**: `stop -expression` For details, see the ***ZeBu Debug Guide***.	- *Methodology 1: Batch/In Regression for zDPI Calls* - *Methodology 2: Capturing zDPI Information and the Associated Waveforms*

Requirement	Keywords	Support debug methodology
How to capture Sequence of Events (SoE)?	**RTL**: Signals used must be capturable with FWC or QiWC **zRci**: `stop -cel <FSM.cel>` For details, see the ***ZeBu Debug Guide***.	- *Methodology 3: Applying Runtime Triggers to Capture and Expand Waveforms* - *Methodology 4: Stimuli Record With zDPI and Replay With Waveform* - *Methodology 5: Prenotification Waveform*
How to enable SystemVerilog Assertions?	**RTL**: `sva` must have been applied. **UTF**: `assertion_synthesis` **zRci**: `sva` For details, see the ***ZeBu User Guide***.	
How to capture and replay Stimuli?	**UTF**: debug-offline_debug true **zRci**: sniffer, replay	
What are the technologies for which I can choose the moment to process them(Runtime or Post-run)	DPI calls SystemVerilog Assertions Runtime trigger/ZTDB scanner	*Methodology 1: Batch/In Regression for zDPI Calls*

2.2 Requirements Addressed by Each Debug Methodology

The following table lists the requirements addressed by each debug methodology.

Debug methodology	Main uses
Methodology 1: Batch/In Regression for zDPI Calls	• Capture the DPI ZTDB using zDPI • Post-process the ZTDB to generate logs
Methodology 2: Capturing zDPI Information and the Associated Waveforms	• Capture the DPI ZTDB using zDPI. Post-process the ZTDB to generate logs • Capture the waveforms at runtime using QiWC on design instances at runtime • Capture the waveforms using dynamic-probes/readback technology on any signal in the DUT • Waveform reconstruction
Methodology 3: Applying Runtime Triggers to Capture and Expand Waveforms	• Capture the waveforms using FWC on DUT key signals at runtime and the Sequence of Events (SoE) of state machine using Runtime Trigger • Capture the waveforms using dynamic-probes/QiWC readback technology on any signal in the DUT • Waveform reconstruction
Methodology 4: Stimuli Record With zDPI and Replay With Waveform	• Record the stimuli at runtime • Capture the DPI ZTDB using zDPI. Post-process the ZTDB to generate logs • Replay the stimuli at runtime • Capture the waveforms using dynamic-probes/readback technology on any signal in the DUT • Waveform reconstruction
Methodology 5: Prenotification Waveform	• Record the stimuli at runtime • Apply Runtime Trigger (while capturing of FSM signals) • Replay the stimuli at runtime • Capture the waveforms using QiWC or readback technology on any signal in the DUT • Waveform reconstruction

3 Debug Methodologies

This section describes the common methodologies that you can follow when using the ZeBu debug technologies. The methodology you choose depend on the debug requirements and the resource constraints. See the Planning for Debug chapter for information on identifying the most suitable methodology based on your debug requirements.

For more information on debug methodologies, see the following subsections:

- *Methodology 1: Batch/In Regression for zDPI Calls*
- *Methodology 2: Capturing zDPI Information and the Associated Waveforms*
- *Methodology 3: Applying Runtime Triggers to Capture and Expand Waveforms*
- *Methodology 4: Stimuli Record With zDPI and Replay With Waveform*
- *Methodology 5: Prenotification Waveform*

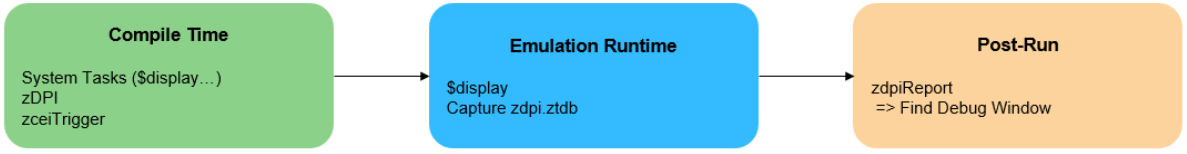
Note *To view and run examples of each methodology, go to the example directory in the installation.*

3.1 Methodology 1: Batch/In Regression for zDPI Calls

The Batch/In regression methodology is best suited when you have C DPI imported functions. The advantages of using this methodology are as follows:

- DPI-based testbench
- Fastest emulation runtime

The high-level flow of this methodology is as follows:



This section describes changes you need to make in the RTL, UTF file for compile time, procedure at runtime, and the output generated. For more information, see the following subsections:

- *RTL Preparation and Compilation Changes for Batch/In Regression*
- *Runtime for Batch/In Regression Methodology*
- *DPI ZTDB Post Processing for Batch/In Regression Methodology*

3.1.1 RTL Preparation and Compilation Changes for Batch/In Regression

RTL Updates

Methodology 1 requires you to make sure the RTL is updated with dynamic triggers using the `zceiTrigger` command. For example:

```
zceiTrigger My_Dynamic_Trigger_for_Counters (
  .trigger_input({hw_top.dut.signal1[31:0],
                  hw_top.dut.signal2[31:0]})
);
```

Compilation Changes

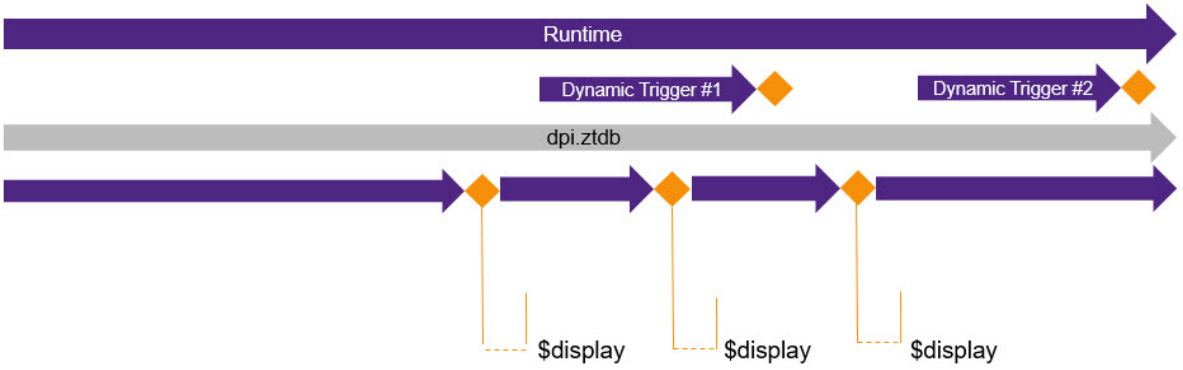
For compilation updates for Methodology 1, specify the following `dpi_synthesis` UTF command and relevant system tasks. For example:

```
dpi_synthesis -enable  all
system_tasks  -enable -task {$display}
system_tasks  -enable -task {$finish}
```

3.1.2 Runtime for Batch/In Regression Methodology

During the emulation run, you can choose when to start the capture of DPI calls in ZTDB and choose how to use dynamic triggers. Capturing offline DPI calls enables the fastest emulation run.

The following figure shows the runtime flow:



Start Capture of DPI Calls in ZTDB

To start and capture the ZTDB file, use the `ccall` UCLI command to generate a ZTDB file called `dpi.ztdb`:

```
ccall -dump_offline dpi.ztdb -dump_all
```

Stopping Emulation Based On Single Events With Dynamic Trigger

Use the `stop` UCLI command to stop the emulation on specific events. For example:

```
stop -enable hw_top.My_Dynamic_Trigger_for_Counters -action Trigger_callback
```

The following code snippet shows how to use dynamic triggers at runtime:

```
# Callback procedure to specify the action on the dynamic trigger
proc Trigger_callback {} {
    # Action example:
    set trigger_cycle [run 0]
    puts "Trigger Callback: at cycle $trigger_cycle "
}

# Configuration of the dynamic trigger
stop -expression {
        (hw_top.dut.signal1[31:0] == 32'd17) &&
        (hw_top.dut.signal2[31:0] == 32'd4294967278)
        } hw_top.My_Dynamic_Trigger_for_Counters

# Callback procedure and enable the dynamic trigger
stop -enable hw_top.My_Dynamic_Trigger_for_Counters -action Trigger_callback
...
# Actions…
```

For more information on the `ccall` and `stop` UCLI commands, see the ***ZeBu Unified Command-Line User Guide***.

3.1.3 DPI ZTDB Post Processing for Batch/In Regression Methodology

After the emulation runtime is complete, use the g++ compiler create a shared object (`shared_object.so`) file, which is then passed to the **`zdpiReport`** tool. You can view the report using the following **`zdpiReport`** tool. Use the output of this tool to analyze the results of the C DPI calls.

The key options that you can specify are as follows:

```
zdpiReport -f <file_containing_list_of_functions_to_Call> \
           -i <dpi.ztdb> \
           -l <shared_object.so> \
           -z <zebu.work>
```

Where:

- `-f`: Specifies the file containing the list of DPI functions to call. See the *Runtime for Batch/In Regression Methodology* section for a sample code snippet containing calls.
- `-i`: Specifies the ZTDB file name where the offline calls were outputted during runtime.
- `-l`: Specifies the shared library object that contains the DPI function implementation. The shared object was created by the g++ compiler.
- `-z`: Specifies the compilation directory where the runtime database is available (default is `./zebu.work`).

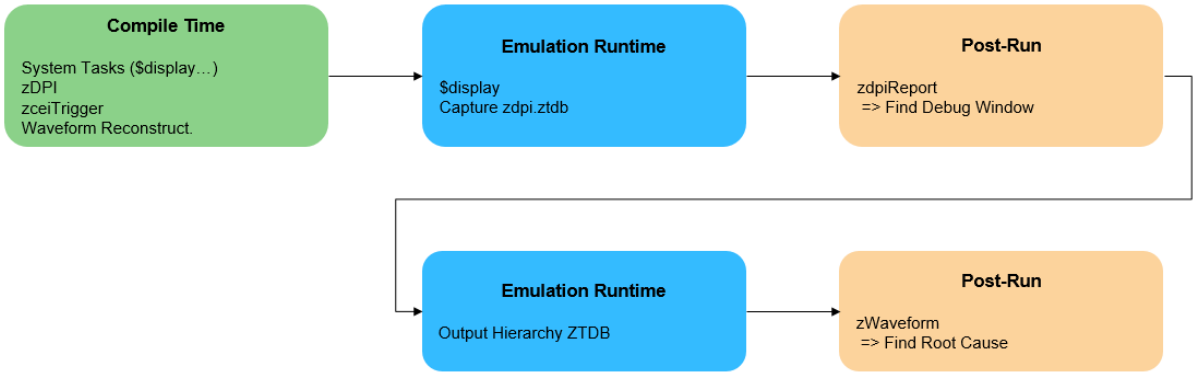
For more information on the **`zdpiReport`** tool, see the ***ZeBu Debug Guide***.

3.2 Methodology 2: Capturing zDPI Information and the Associated Waveforms

This methodology is best suited when you have C DPI imported functions and you want to view waveforms. The advantages of using this methodology are as follows:

- DPI-based testbench
- Ability to view waveforms after having identified the window of debug

The high-level flow of this methodology is as follows:



This section describes changes you need to make in the RTL, UTF file for compile time, procedure at runtime, and the output generated. For more information, see the following subsections:

- *RTL Preparation for Capturing zDPI Information*
- *Compilation Updates for Capturing zDPI Information*
- *Runtime for Capturing Waveforms in Methodology 2*

3.2.1 RTL Preparation for Capturing zDPI Information

For this methodology, you need to modify the RTL for dynamic triggers and QiWC waveform capture.

Update RTL for Dynamic Triggers

Update the RTL with dynamic triggers using the `zceiTrigger` command. For example:

```
zceiTrigger My_Dynamic_Trigger_for_Counters (
  .trigger_input({hw_top.dut.signal1[31:0],
                  hw_top.dut.signal2[31:0]})
);
```

Update RTL for QiWC Waveform

Create a new Verilog file with the following module. This module defines hierarchies on which to apply QiWC.

For example:

```
module my_dumpvars();
    initial begin: Full_Chip_VS
        (* qiwc *) $dumpvars (0, hw_top);
    end
endmodule
```

For VCS elaboration, this module is called by the VCS script.

3.2.2 Compilation Updates for Capturing zDPI Information

For compilation, you need to *Update the UTF File* and the *VCS Command*.

Update the UTF File

For compilation, specify the following `dpi_synthesis` UTF command, relevant system tasks, and the `-waveform_reconstruction` option. For example:

```
dpi_synthesis -enable  all
system_tasks  -enable -task {$display}
system_tasks  -enable -task {$finish}
# Enable waveform reconstruction and Verdi KDB generation
```

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

VCS Command

The VCS command calls the module (`my_dumpvars`) that you added in the *Update RTL for QiWC Waveform* stage. For example:

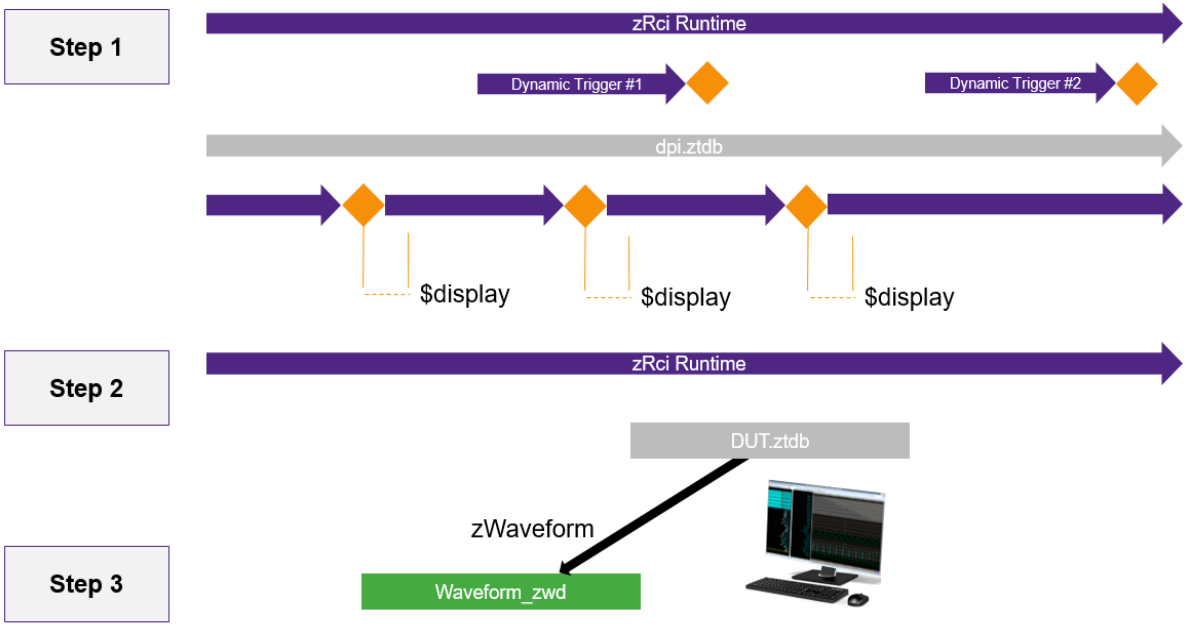
```
vlogan SRC/RTL/my_dumpvars.v
…
vcs hw_top my_dumpvars
```

3.2.3 Runtime for Capturing Waveforms in Methodology 2

There are three stages to the process of capturing zDPI information and associated waveforms in Methodology 3. The stages are as follows:

1. *Step 1: Identify Debug Window*
2. *Step 2: ZTDB Waveform Capture*
3. *Step 3: Expand and View the ZWD Waveform*

The following figure shows the runtime flow:



Step 1: Identify Debug Window

During the emulation run, you can choose when to start the capture of DPI calls in ZTDB and choose how to use the dynamic triggers. Capturing offline DPI calls enables the fastest emulation run. For more information, see the *Runtime for Batch/In Regression Methodology* and *DPI ZTDB Post Processing for Batch/In Regression Methodology* sections of *Methodology 1: Batch/In Regression for zDPI Calls*.

Step 2: ZTDB Waveform Capture

In this step, you need to update the UCLI file and run a new emulation with **`zRci`**.

To capture a slice of the ZTDB (`full_chip.ztdb`), create or update the UCLI file with the following commands.

```
set dut_fid [dump -file full_chip.ztdb -qiwc]
dump -add_value_set {Full_Chip_VS} -fid $dut_fid
dump -interval {20000000total_samples,250slices} -fid $dut_fid
dump -enable -fid $dut_fid
…
dump -disable -fid $dut_fid
dump -flush -fid $dut_fid
dump -close -fid $dut_fid
```

Step 3: Expand and View the ZWD Waveform

Before you can view the ZWD waveform, the ZTDB waveform needs to be expanded using the **zSimzilla** tool. This tool generates the ZWD directory, which you can then pass to Verdi to view the waveform.

In the following code snippet, the ZTDB file is `full_chip.ztdb` and the ZWD directory is `full_chip_zwd`.

Waveform Expansion

```
zSimzilla --zebu-work zcui.work/zebu.work \
          --ztdb full_chip.ztdb \
          --timescale 1ps \
          --jobs 250 \
          --command <qrsh | lsf> \
         --zwd full_chip_zwd
```

Viewing the Waveform

```
verdi -emulation --zebu-work zcui.work/zebu.work –ssf full_chip_zwd
```

For more information on waveform capture using Verdi, see the ***ZeBu-Verdi Integration Guide***.

3.3 Methodology 3: Applying Runtime Triggers to Capture and Expand Waveforms

This methodology 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.

With this methodology, you can use offline ZTDB scanner or Runtime Trigger to process a sequence of events (SoE). The SoE represents the Finite State Machine (FSM) that is defined using the CEL language. The events are based on the captured signals. You can use either of the following tools to identify the window of debug and determine the root cause:

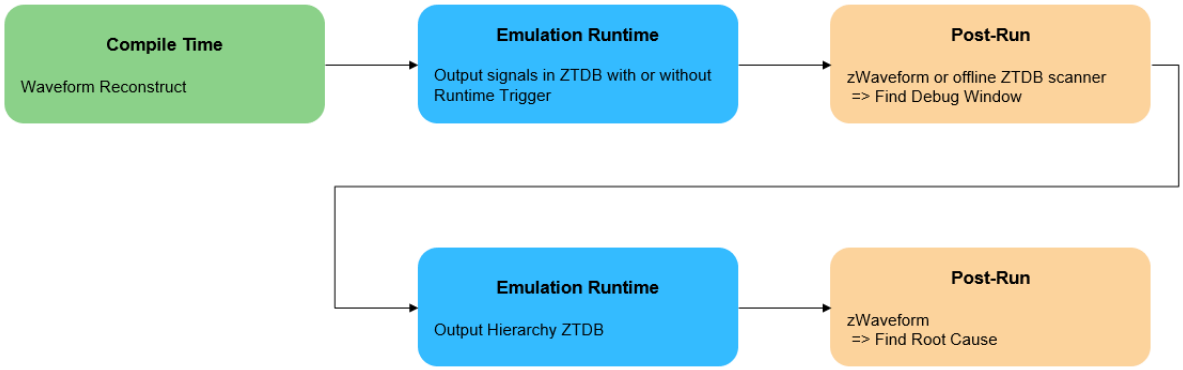
- **Runtime Trigger**: This is used during emulation. After a sequence of events is triggered, a notification is sent to the host PC. There is a delay. You can stop the emulation at the time of receiving the notification. Therefore, the window of debug is identified during emulation.
- **ZTDB Scanner**: The ZTDB scanner is used after the emulation run is complete. The ZTDB scanner outputs the cycle of notification.

To view the waveform for the window of debug, you can capture ZTDB waveform and expand with either QiWC or dynamic-probe (Readback) technology.

The advantages of using this methodology are as follows:

- FWC has the benefit of a high driver clock frequency (up to 2.9 MHz)
- With Runtime Trigger, you can change the FSM without having to recompile, if the signals of the FSM are captured.

The high-level flow of this methodology is as follows:



This section describes changes you need to make in the RTL, UTF file for compile time, procedure at runtime, and the output generated. For more information, see the following subsections:

- *RTL Preparation for QiWC and FWC Waveform Capture*
- *Compilation Changes for Capturing Waveforms*
- *Runtime for Debug Methodology 3*

3.3.1 RTL Preparation for QiWC and FWC Waveform Capture

For Debug Methodology 3, you need to update the RTL for QiWC or FWC waveform capture.

Update RTL for QiWC or FWC Waveform

Create a new Verilog file with the following module. This module defines the key signals that need to monitor during emulation or post emulation. In the following code snippet, the key signals are defined in the `Key_Signals_VS` value-set.

The `Full_Chip_VS` value-set is used to define the hierarchies on which to apply QiWC for debug.

For example:

```
module my_dumpvars();
    initial begin: Key_Signals_VS
        (* fwc *) $dumpports (hw_top.main_cpu);
    end
    initial begin: Full_Chip_VS
        (* qiwc *) $dumpvars (0, hw_top);
    end
endmodule
```

For VCS elaboration, this module is called by the VCS script.

3.3.2 Compilation Changes for Capturing Waveforms

For compilation in Debug Methodology 3, you need to *Update the UTF File* and the *VCS Command*.

Update the UTF File

For compilation, specify the `-waveform_reconstruction` option. For example:

```
# Enable waveform reconstruction and Verdi KDB generation
debug –waveform_reconstruction true
debug –verdi_db true
```

VCS Command

The VCS command calls the module (`my_dumpvars`) that you added in the *Update RTL for QiWC or FWC Waveform* stage. For example:

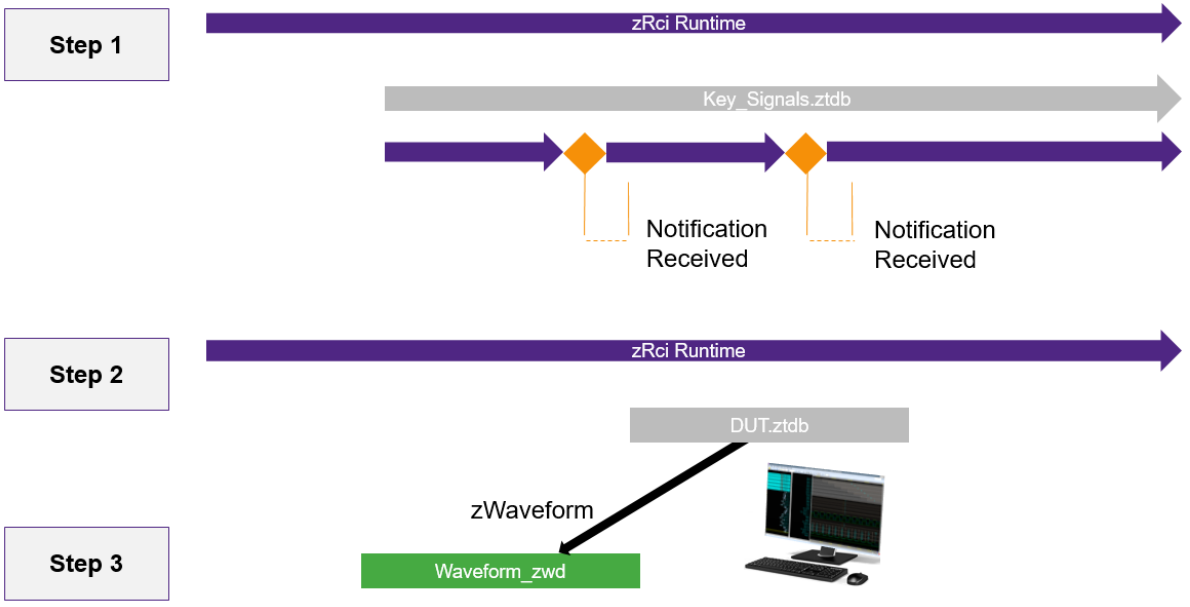
```
vlogan SRC/RTL/my_dumpvars.v
…
vcs hw_top my_dumpvars
```

3.3.3 Runtime for Debug Methodology 3

There are three stages to the process of capturing and expanding waveforms in debug Methodology 3. The stages are as follows:

1. *Step 1: Apply the Runtime Trigger and Identify Window of Debug*
2. *Step 2: ZTDB Waveform Capture*
3. *Step 3: Expand and View the ZWD Waveform*

The following figure shows the runtime flow:



Step 1: Apply the Runtime Trigger and Identify Window of Debug

During the emulation run, you can choose when to start the capture of key DUT signals in the ZTDB. The key DUT signals are captured using the Value-Set. After the sequence of events (SoE) is triggered, the Runtime Trigger sends the signal information to the testbench. There is a delay in receiving the notification.

```
set __notifier_done 0
set __notifier_cycle 0
# Callback procedure to collect notification signal
proc RT_callback {module sampleNumber ClockCycle isLastNotify} {
    global __notifier_done
    global __notifier_cycle
    set __notifier_done $isLastNotify
    set __notifier_cycle $ClockCycle
}
# Start FWC output using FWC value-set #
set keys_id [dump -file key_signals_for_rt.ztdb -fwc]
dump -add_value_set {Key_Signals_VS} -fid $keys_id
stop -cel FSM_Notification.cel -action RT_callback -fid $keys_id
dump -enable -fid $keys_id

…
run
while {!$__notifier_done} {
   after 10
}
# The next line provides the exact cycle of notification
puts "SoE Triggered at cycle $__notifier_cycle"
…
dump -disable -fid $keys_id
dump -flush -fid $keys_id
dump -close -fid $keys_id
```

Step 2: ZTDB Waveform Capture

In this step, you need to update the UCLI file and run a new emulation with **zRci**.

To capture a sliced ZTDB (`full_chip.ztdb`), create or update the UCLI file with the following commands.

```
set dut_fid [dump -file full_chip.ztdb -qiwc]
dump -add_value_set {Full_Chip_VS} -fid $dut_fid
dump -interval {20000total_samples,10slices} -fid $dut_fid
dump -enable -fid $dut_fid
…
dump -disable -fid $dut_fid
dump -flush -fid $dut_fid
dump -close -fid $dut_fid
```

To generate and view the readback waveform, add `set dut_fid [dump -file full_chip.ztdb -dynamic_probe]` in place of the following lines:

```
set dut_fid [dump -file full_chip.ztdb -qiwc]
dump -add_value_set {Full_Chip_VS} -fid $dut_fid
```

Note *The value-set is only applicable to QiWC.*

Step 3: Expand and View the ZWD Waveform

Before you can view the ZWD waveform, expand the ZTDB waveform using the **zSimzilla** tool. This tool generates the ZWD directory, which you can then pass to Verdi to view the waveform.

In the following code snippet, the ZTDB file is `full_chip.ztdb` and the ZWD directory is `full_chip_zwd`.

Waveform Expansion

```
zSimzilla --zebu-work zcui.work/zebu.work \
          --ztdb full_chip.ztdb \
          --timescale 1ps \
          --jobs 250 \
          --command <qrsh | lsf> \
          --zwd full_chip_zwd
```

Waveform Viewing

```
verdi -emulation --zebuwork zcui.work/zebu.work -ssf full_chip_zwd
```

For more information on waveform capture using Verdi, see the ***ZeBu-Verdi Integration Guide***.

3.4 Methodology 4: Stimuli Record With zDPI and Replay With Waveform

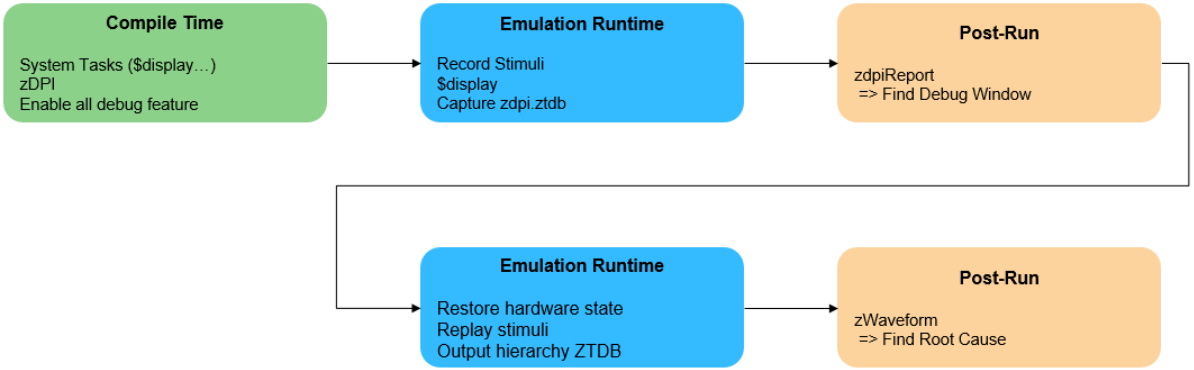
This is an advanced methodology suited for complex debug scenarios and scenarios involving billions of cycles. This methodology avoids rerunning the billions of cycles to capture waveforms.

Using Debug Methodology 4, you can perform the following:

1. Record stimuli at runtime and capture zDPI calls in the ZTDB.
2. Post process the ZTDB to identify the window of debug.
3. With **`zRci`**,capture waveforms in the window of debug.
4. Expand waveforms with **zSimzilla** and view them in Verdi.

To learn more about the stimuli-replay technology, see the *Stimuli Replay* section.

The high-level flow of this methodology is as follows:



This section describes changes you need to make in the RTL, UTF file for compile time, and procedure at runtime. For more information, see the following subsections:

- *RTL Preparation and Compilation Updates for Debug Methodology 4*
- *Main Run for Stimuli Record With zDPI and Replay With Waveform*

3.4.1 RTL Preparation and Compilation Updates for Debug Methodology 4

The updates required in the RTL are described in the **RTL Preparation** section of *Methodology 3: Applying Runtime Triggers to Capture and Expand Waveforms*.

Compilation

Similar to *Methodology 2: Capturing zDPI Information and the Associated Waveforms*, specify the `dpi_synthesis` UTF command and the relevant system tasks.

To enable offline debug, waveform expansion, and Verdi KDB generation, you need to specify only one UTF command:

```
debug -all true
```

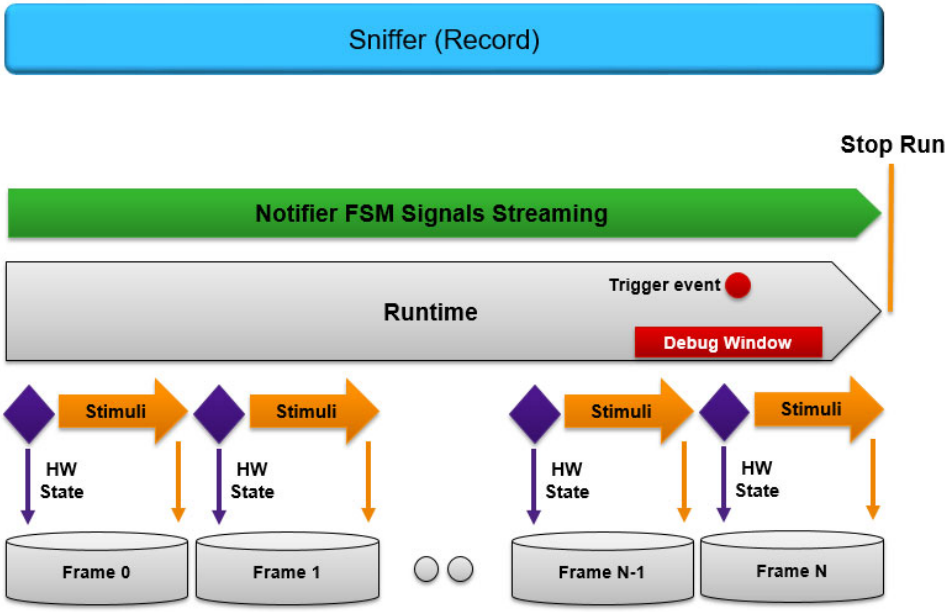
The `debug -all true` command is equivalent to specifying the following commands:

```
debug –offline_debug true
debug –waveform_reconstruction true
debug –verdi_db true
```

3.4.2 Main Run for Stimuli Record With zDPI and Replay With Waveform

During the emulation run record the stimuli and capture DPI calls simultaneously in ZTDB using `sniffer` and `ccall` UCLI commands. For more information on zDPI calls, see the *Methodology 1: Batch/In Regression for zDPI Calls* section.

Use the `sniffer` UCLI command to record stimuli. The stimuli recording is done in frames. While debugging, this enables you to isolate the issue and view only the specific frame. The following figure shows the record stimuli stage during the main emulation run.



The following code snippet shows the key UCLI commands required in this methodology:

```
…
#start Emulation
start_zebu emulation_output_dir
…
#offline DPI
ccall -dump_offline -dump_all dpi.ztdb
ccall -enable_offline
…
sniffer -auto_create 200s
…
sniffer -stop
ccall -flush -disable
```

After the `sniffer -stop` command is executed, you can extract the available replay windows with the following commands:

```
foreach frame [sniffer -list] {
   set res [sniffer -info $frame -cycles];
   set start [lindex $res 0]; set end [lindex $res 1]
   puts " Frame: from $start to $end"
}
```

Then, you can restore the emulation to the required cycle to capture waveforms as shown in the following example:

```
sniffer -restore -at 1000000000
dump -file wave.ztdb -dyn
dump -file wave.ztdb -enable
replay 50000
dump -file wave.ztdb -close
```

Waveform Expansion

```
zSimzilla -zebu-work path/to/zebu.work \
--ztdb wave.ztdb \
--timescale 1ns \
--jobs 250 \
--command <qrsh | bsub>
--zwd wave_zwd
```

Waveform Viewing

```
verdi -emulation --zebu-work zcui.work/zebu.work -ssf wave_zwd
```

For more information on waveform capture using Verdi, see the ***ZeBu-Verdi Integration Guide***.

3.5 Methodology 5: Prenotification Waveform

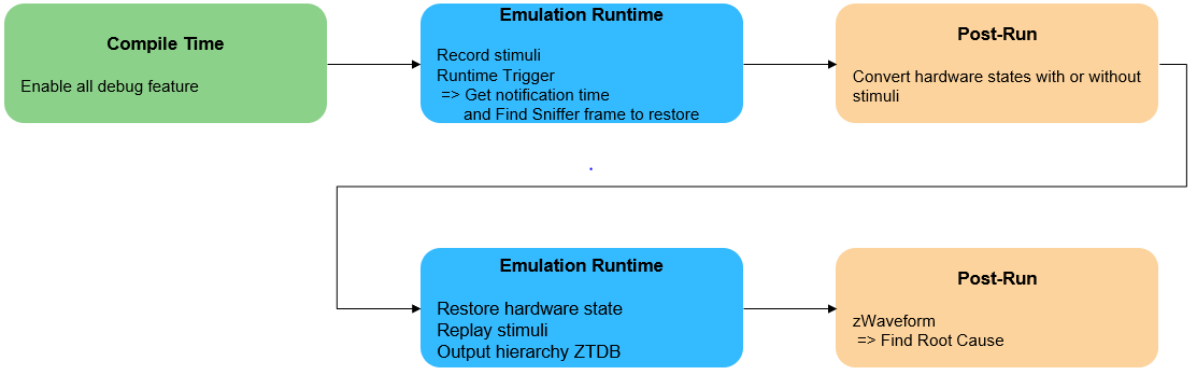
This methodology is best suited when you want to capture the waveform in a window, which includes a sequence of events (SoE) and a notification event.

This methodology leverages techniques described in *Methodology 3: Applying Runtime Triggers to Capture and Expand Waveforms* and *Methodology 4: Stimuli Record With zDPI and Replay With Waveform*.

Using debug Methodology 5, you can perform the following:

1. Using **`zRci`**, record stimuli at runtime and use the Runtime Trigger during emulation. After a sequence of events is triggered, a notification is sent to the testbench. You can stop the emulation at the time of receiving the notification. Therefore, the notification cycle allows you to determine the window of debug.
2. Using **`zRci`**, when replaying the stimuli you can capture ZTDB waveforms in the window of debug.
3. Expand waveforms with **zSimzilla** and view them in Verdi.

The high-level flow of this methodology is as follows:



This section describes changes you need to make in the RTL, UTF file for compile time, and procedure at runtime. For more information, see the following subsections:

- *RTL Preparation and Compilation Changes for Prenotification Waveform*
- *Main Run for Prenotification Waveform Debug Methodology*
- *Replay for Debug Methodology 5*

3.5.1 RTL Preparation and Compilation Changes for Prenotification Waveform

The updates required in the RTL are described in the **RTL Preparation** sections of the *Methodology 3: Applying Runtime Triggers to Capture and Expand Waveforms* section.

Compilation

To enable offline debug, waveform expansion and Verdi KDB generation, you need to specify only one UTF command:

```
debug -all true
```

The `debug -all true` command is equivalent to specifying the following commands:

```
debug –offline_debug true
debug –waveform_reconstruction true
debug –verdi_db true
```

3.5.2 Main Run for Prenotification Waveform Debug Methodology

During the emulation run, the stimuli is recorded and after the sequence of events (SoE), the Runtime Trigger sends a notification from the hardware to the testbench.

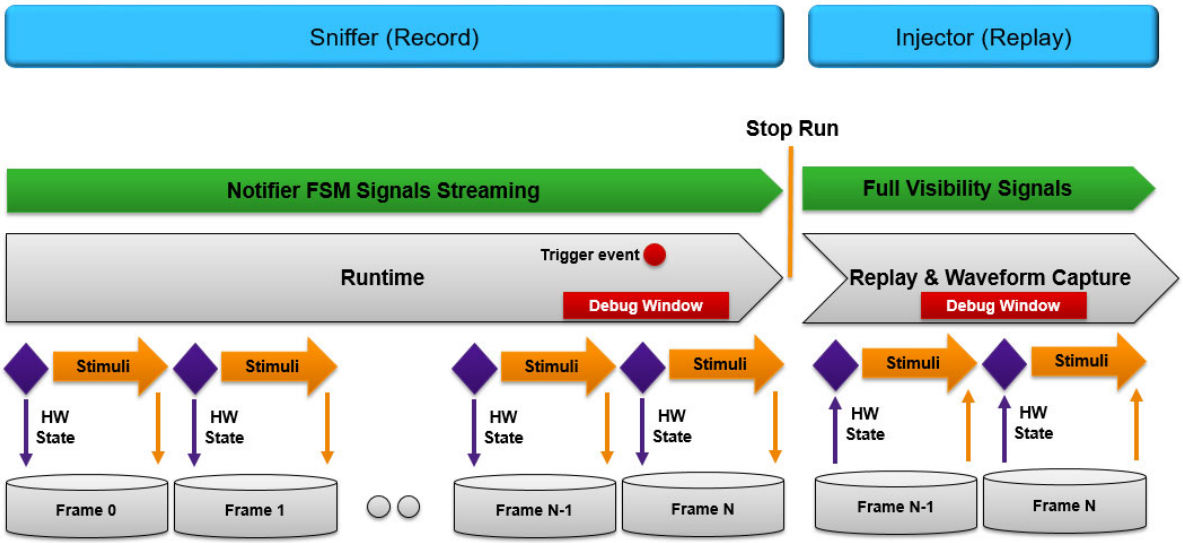
The window of debug is identified by recording the stimuli during the main run. After the SoE is triggered, the Runtime Trigger sends signal information to the testbench.

Use the `sniffer` UCLI command to record stimuli. The recorded stimuli and the saved hardware state are then stored in frames.

Use the `stop` UCLI command to enable the Runtime Trigger and get the notification cycle.

You can compare the notification cycle with the start-and-end cycles of frames to determine the hardware state to be restored. Consequently, you can capture the ZTDB waveform for the window of debug.

The following figure shows the record stimuli and replay stages during the main emulation run.



The following code snippet shows the key UCLI commands required in debug methodology 5:

```
…
#start Emulation
start_zebu emulation_output_dir
…
# To allow ZTDB replay (avoiding ZTDB to SMD stimuli conversion)
replay -config -reader ztdb_threaded_scanner
set __notifier_done 0
# Callback procedure to collect SW Notification signal
proc RT_callback {module sampleNumber ClockCycle isLastNotify} {
    global __notification_cycle
    global __notifier_done
```

```
    # Runtime Trigger Callback to record the Notification cycle
    set __notification_cycle $ClockCycle
    set __notifier_done $isLastNotify
}
sniffer -auto_create 3600s
# Start FWC dump using FWC value-set #
set keys_id [dump -file key_signals_for_rt.ztdb -fwc]
# RT_callback stops the notification cycle
stop -cel FSM_Notification.cel -action RT_callback -fid $keys_id
dump -add_value_set {Key_Signals_VS} -fid $keys_id
# Make emulation stop on SW Notification
stop -config stop_on_notify on
dump -enable -fid $keys_id
…
#testbench run
while {!$__notifier_done} {
   after 10
}
#testbench stop
…
sniffer -stop
dump -close -fid $keys_id
#size of the window of Debug
set cycles_to_dump 10000
#comparing Notification cycle with the Star-and-end cycles of the
Frames
foreach frame [sniffer -list] {
    set res [show $frame -cycles]
```

```
    set start [lindex $res 0]
    set end   [lindex $res 1]
    set cycle_to_start_capture [ expr {$__notification_cycle -
$cycles_to_dump} ]
    if { ($start <= $cycle_to_start_capture) && ($end >=
$cycle_to_start_capture) } {
        set frame_to_restore $frame
        set cycles_before_capture  [ expr {$cycle_to_start_capture -
$start} ]
    }
}
#The frame to be restored is $frame_to_restore
```

3.5.3 Replay for Debug Methodology 5

After you have identified the window of debug, you can replay the specific frame that led to the issue. The stages in this process are as follows:

- *Step 1: Capture a ZTDB Waveform for the Window of Debug*
- *Step 2: Expand and View the ZWD Waveform*

Step 1: Capture a ZTDB Waveform for the Window of Debug

Using **`zRci`**, when replaying the stimuli, you can capture ZTDB waveforms in the window of debug.

The following code snippet shows the commands required to replay.

```
# Restore Frame previously identified
sniffer -restore $frame_to_restore
# Replay the Simutili until the start of Window of Debug
replay $cycles_before_capture
# Start ZTDB waveform capture for the window of debug
```

```
set full_chip_pre_rt [dump -file full_chip_pre_rt.ztdb -qiwc]
dump -add_value_set {Full_Chip_VS} -fid $full_chip_pre_rt
dump -interval 4 -fid $full_chip_pre_rt
dump -enable -fid $full_chip_pre_rt
replay $cycles_to_dump
dump -close -fid $full_chip_pre_rt
```

Step 2: Expand and View the ZWD Waveform

Before you can view the ZWD waveform, the ZTDB waveform needs to be expanded using the **zSimzilla** tool. This tool generates the ZWD directory, which you can then pass to Verdi to view the waveform.

In the following code snippet, the ZTDB file is `full_chip_pre_rt.ztdb` and the ZWD directory is `full_chip_pre_rt_zwd`.

Waveform Expansion

```
cd emulation_output_dir/


zSimzilla --zebu-work zcui.work/zebu.work \
          --ztdb full_chip_pre_rt.ztdb \
          --timescale 10ns \
          --jobs 250 \
          --command <qrsh | lsf> \
          --zwd full_chip_pre_rt_zwd
```

Waveform Viewing

```
verdi -emulation --zebuwork zcui.work/zebu.work –ssf
full_chip_pre_rt_zwd
```

For more information on waveform capture using Verdi, see the ***ZeBu-Verdi Integration Guide***.