ZeBu® Verdi Integration Guide

Version V-2024.03-1, July 2024

SYNOPSYS®

Copyright and Proprietary Information Notice

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 https://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 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

Feedback

Contents

Preface

This chapter has the following sections:

About This Book

The *ZeBu-Verdi Integration Guide* describes the ZeBu-Verdi debug features, technologies, and methodologies to be used with ZeBu.

Contents of This Book

The *ZeBu-Verdi Integration Guide* has the following chapters:

Chapter	Describes...
Waveform Reconstruction Using Verdi	Methods used for waveform reconstruction.
Interactive Debug Using Verdi	How to use Verdi for interactive debug.
Viewing ZeBu Emulation Summary Result	How to view emulation summary in Verdi.
Viewing ZeBu Internal Gate-Level Netlists in Verdi	How to view to ZeBu internal gate-level netlist in Verdi.
Complex Event Language Support in nTrace	Usage of CEL for ZeBu.
Monitoring State Machine Variables in nWave	How to monitor state machine variables in Verdi.
Viewing ZeBu Memory Using Verdi	Using Verdi, both HDL and EDIF memories can be viewed during zRci or Simulation Acceleration runtime.

Chapter	Describes...
ZeBu Design Manipulation Using Verdi	Verdi supports the ZeBu design manipulation based on Knowledge Database (KDB)-Engineering Change Order (ECO) to view the original source code.
Simulation Acceleration Interactive Mode	Simulation Acceleration interactive debug that helps debug both software (testbench) and hardware (DUT) in the Verdi interactive debug environment.
Analyze FSDB Using the zwdutils Utility	FSDB can be analyzed more efficiently using the FSDB utilities `fsdbreport`, `fsdb2saif`, and `fsdbextract`.

Related Documentation

Document Name	Description
ZeBu Debug Guide	Provides information on tools you can use for debugging.
ZeBu User Guide	Provides information about the ZeBu emulation syste,
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.

Document Name	Description
ZeBu Synthesis Verification User Guide	Provides a description of zFmCheck.
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.

Customer Support

For any online access to the self-help resources, refer to the documentation and searchable knowledge base available in SolvNetPlus.

To obtain support for your Verdi product, choose one of the following:

- Open a case through SolvNetPlus.

Go to https://solvnetplus.synopsys.com/s/contactsupport and provide the requested information, including:

- Product - L1 as Verdi
- Case Type

Fill in the remaining fields according to your environment and issue.

- Send an e-mail message to *verdi_support@synopsys.com*.

Include product name (L1), sub-product name/technology (L2), and product version in your e-mail, so it can be routed correctly.

Your e-mail will be acknowledged by automatic reply and assigned a Case number along with Case reference ID in the subject (ref:_...:ref).

For any further communication on this Case via e-mail, send an e-mail to *verdi_support@synopsys.com* and ensure to have the same Case ref ID in the subject header or else it opens duplicate cases.

Note:

In general, we need to be able to reproduce the problem in order to fix it, so a simple model demonstrating the error is the most effective way for us to identify

the bug. If that is not possible, then please provide a detailed explanation of the problem along with complete error and corresponding code, if any/permissible.

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

Waveform Reconstruction Using Verdi

There are two methods to reconstruct the waveforms:

- **zCSA** reconstructs the waveforms of signals and instances you specify. It provides both a GUI and a batch mode, using a project file or the command line.

  For more information about reconstructing the waveforms using **zCSA**, see *ZeBu Debug Guide*.

- Interactive Waveform Reconstruction within Verdi allows you to view and reconstruct waveforms quickly through drag-and-drop operations and it can be run in two modes:

  - Command Mode For more information about reconstructing the waveforms in command mode, see *ZeBu Debug Guide*.
  - GUI Mode

2

Interactive Debug Using Verdi

The **Emulation** menu is introduced in the *nTrace* window in Verdi to support ZeBu-Verdi integration features. The **Emulation** menu consists of the following submenus:

- **Load Netlist**: Enables you to debug the ZeBu internal gate-level netlist (ZNL).
- **Close Netlist**: Enables you to close the debug flow for ZNL.
- **Emulation Setup**: Enables you to setup the `simz` environment for emulation.
- **Complex Event**: Enables you to identify a sequence or a sequence of events to find the internal state of an emulator.
- **Summary View**: Loads the emulation database directory (`*.db`) to view the emulation result.
- **Open ZTDB Waveform**: Enables you to reconstruct the signal waveform.

The generation of RTL waveforms is integrated with Verdi debugging platforms. The values for combinational signals in these waveforms are calculated through software simulation using the Combinational Signals Accessibility (CSA) feature. You can run Verdi directly to invoke the interactive Waveform Reconstruction functionality.

The interactive Waveform Reconstruction loads the RTL code and ZeBu waveform file (ZTDB) using the following steps:

- Loading the KDB
- Invoking zRci Flow
- Reconstructing Signals Waveform Using ZTDB Waveform

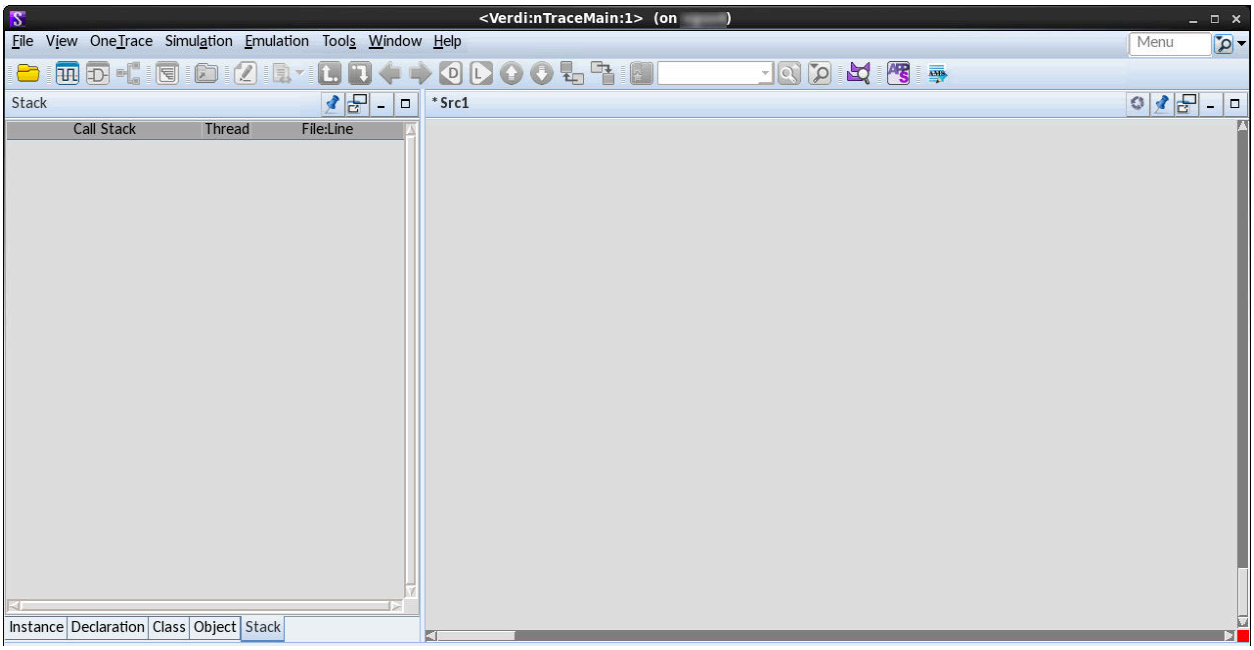
Loading the KDB

Signals added to the waveform are computed quickly from cycle *0* to the last cycle and are displayed on the **Waveform** window. Source code annotation are computed only for the displayed signals. If a signal has added to the **Waveform** window on a certain timing range, while zooming out, the CSA engine computes the waveform on the new range.

Feedback

To run interactive Waveform Reconstruction using the Verdi GUI, run the following command:

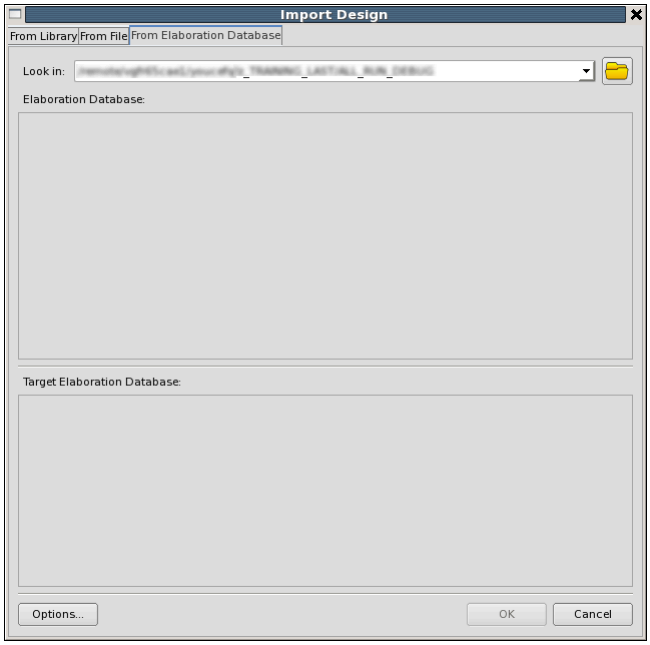
```
verdi -emulation
```



To import the design compiled and elaborated by Verdi, perform the following steps.

1. On the **nTrace** window, click the  icon.

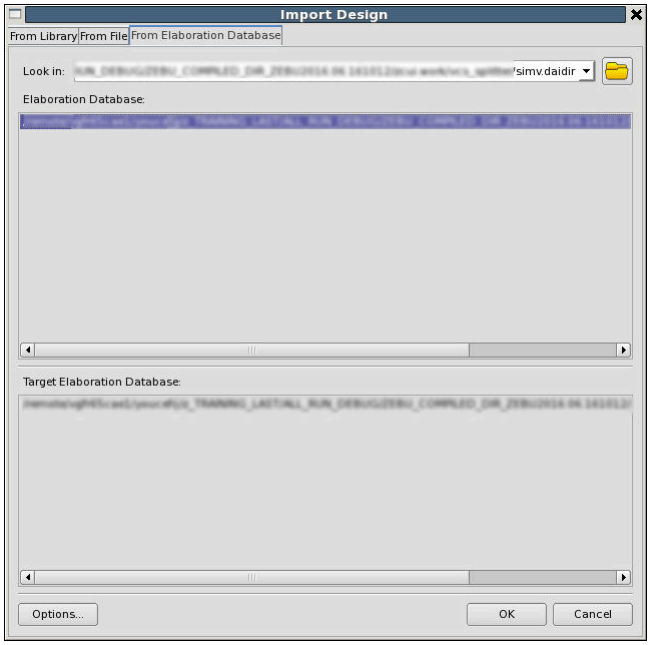
   The **Import Design** dialog box appears.



2. Click the **From Elaboration Database** tab, and the click the  icon.

   The **Working Directory** dialog box appears.

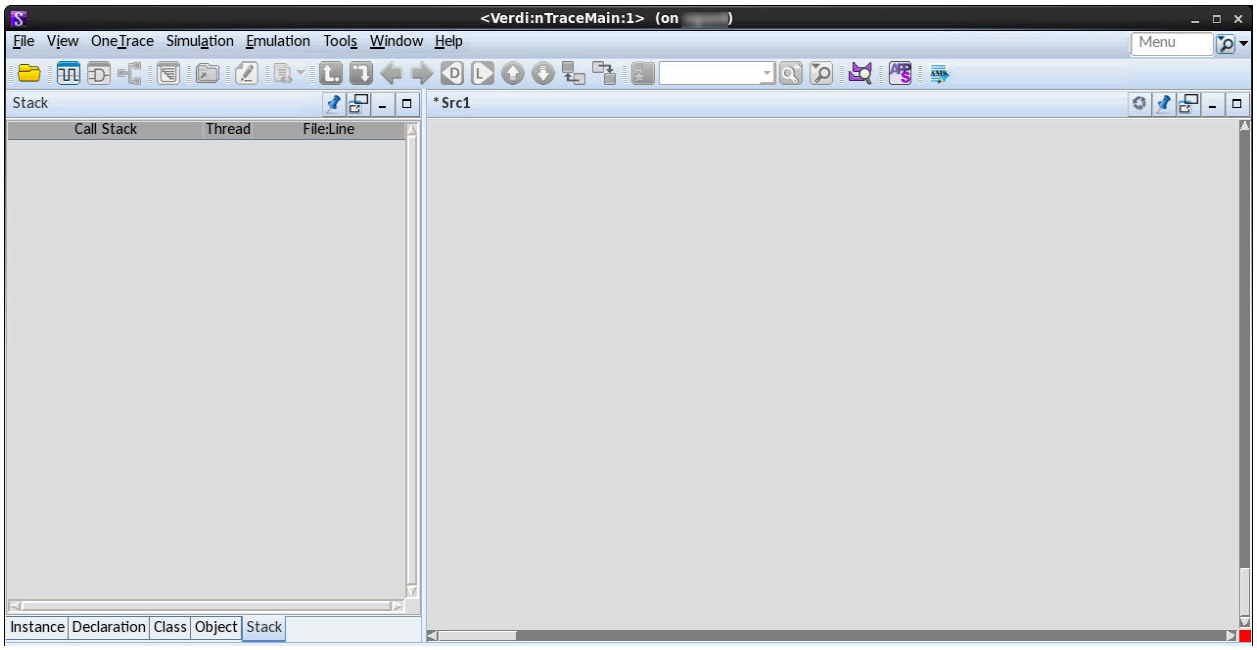
3. Browse to the `zcui.work/vcs_splitter/simv.daidir` directory that contains the Verdi elaboration database (`kdb.elab++`) directory, and click **OK**.



Feedback

4. Click **OK**.

   The **nTrace** window displays the loaded design.



Invoking zRci Flow

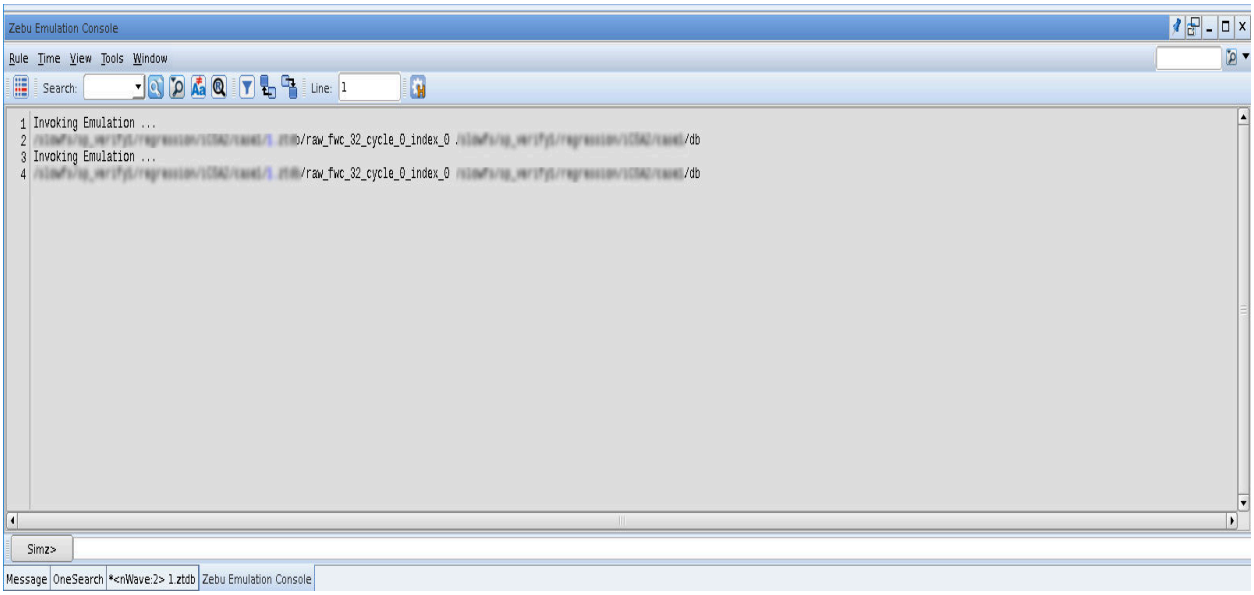
To invoke the **zRci** flow, perform the following steps in the **nTrace** window.

Note:

This is an LCA feature. Therefore, use the `-lca` option.

1. On the **nTrace** window, choose **Emulation** > **Emulation Setup**.
2. Click the  icon to select the following:
   - **Emulation executable**
   - **Emulation database**
   - **Emulation argument**
3. Click the **Emulation argument** list and select an argument.
4. Click **Invoke**.

The **ZeBu Emulation Console** window opens.



Reconstructing Signals Waveform Using ZTDB Waveform

You can run Verdi directly to invoke interactive Waveform Reconstruction functionality, compute the CSA value change, open a ZTDB waveform, view the signal waveforms, and save the simulation results into an FSDB file.

To interactively run the CSA, open the ZTDB waveform. You can reconstruct the waveform and save the results into an FSDB file.

You can invoke the ZTDB waveform using one of the following ways.

- Invoking the ZTDB Waveform Using the Command line
- Invoking the ZTDB Waveform Using Tcl
- Invoking the ZTDB Waveform From nTrace
- Invoking the ZTDB Waveform From nWave

The **Open ZTDB Waveform** dialog box includes the options, as listed in the following table:

Options	Description
ZTDB	Allows you to specify the ZTDB directory.
ZeBu.work	Allows you to specify the ZeBu work directory (default of `$ZEBU_WORK`).

Options	Description
Simulate Memories	Allows you to view the memory signals' waveform value, when the **Simulate Memories** check box is selected. By default, this check box is not selected and the memory signals are not visible in the `getSignal` form.
Open File by Time Range	Allows you to specify a time range for the waveform file. For limitations, seeLimitations.
Show Reason Code from ZeBu Work When no Ztdb is specified	Allows `ZeBu.work` to open and display the reason code for all signal when the **Show Reason Code** from **ZeBu Work When no Ztdb is Specified** check box is selected. By default, this check box is not selected. For more information about the reason code, see Waveform Reason Code.

Invoking the ZTDB Waveform Using the Command line

You can invoke ZTDB waveform using the following command:

```
verdi --zebu-work <zebuWorkName> --input <ZTDBName> --root rootScope
 -emulation -lca --from_time <StartTime> --to_time <EndTime>
```

where,

- `<zebuWorkName>`: Specifies the ZeBu work directory
- `<ZTDBName>`: Specifies the ZeBu Waveform file name
- `<rootScope>`: Specifies the directory of the waveform file from the root scope to access the top of the DUT
- Options to specify a time range for a ZTDB waveform file
  - `<StartTime>`: (Optional) Specifies the start time of the ZTDB waveform file that you want to open. The ZTDB waveform displayed starts from the option you set with this option.
  - `<EndTime>`: (Optional) Specifies the end time of the ZTDB waveform file that you want to open. The ZTDB waveform displayed ends from the option you set with this option.

If you set an invalid value for the `<StartTime>` and `<EndTime>`, Verdi reports the following error and opens the ZTDB waveform in the full time range:

```
Fail to set time range in the opened waveform file. Please specify valid
 time range: 0 ~ 481.
```

Feedback

Note:

Verdi opens the ZTDB with waveform captured time unit. If the time unit for capturing the waveform is not specified, Verdi considers 10ns as the default time unit.

To import a design from `simv.daidir`, specify the `-dbdir` option in Verdi. The following command imports a design from `simv.daidir` specified by `-dbdir`:

```
verdi -dbdir simv.daidir -emulation -lca
```

Invoking the ZTDB Waveform Using Tcl

You can open a ZTDB waveform within a time range by specifying the debLoadSimResult and wvOpenFile commands. The -time option is used to specify the time range and the ZTDB file name.

Example

In the following example, the start time of the ZTDB waveform file (qiwc.ztdb) is set to 100. The end time is set to 200.

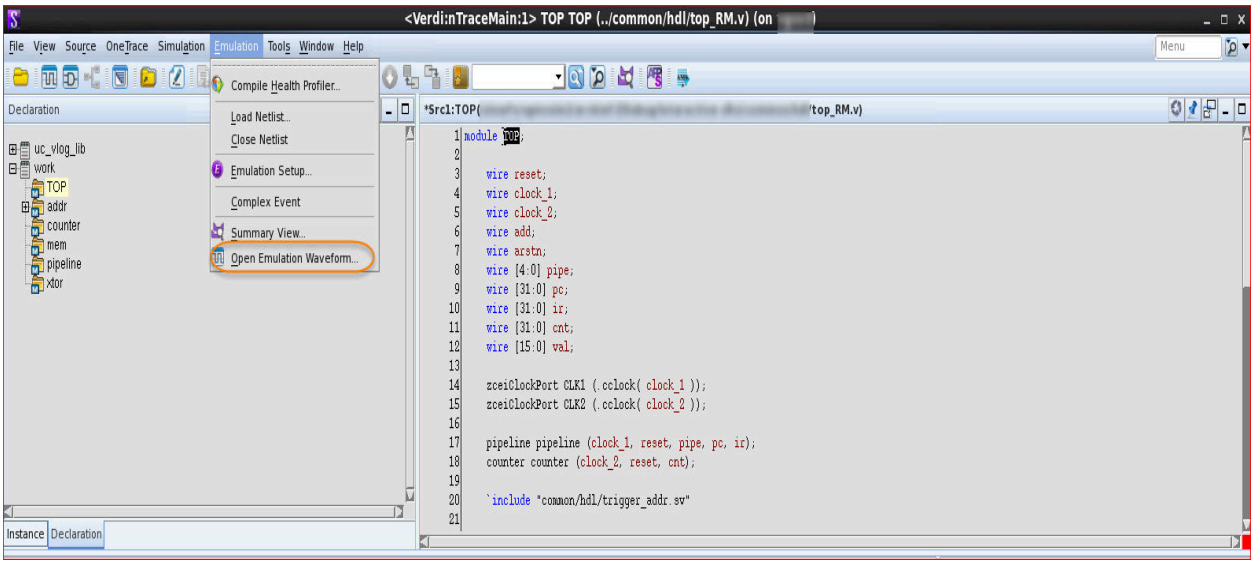
```
debLoadSimResult -time 100 200 {qiwc.ztdb}
wvOpenFile -time 100 200 {qiwc.ztdb}
```

Invoking the ZTDB Waveform From nTrace

You can invoke ZTDB waveform from **nTrace** using **Emulation > Open ZTDB Waveform**, as displayed in the following figure

Feedback

Figure 1 Open ZTDB Waveform From nTrace



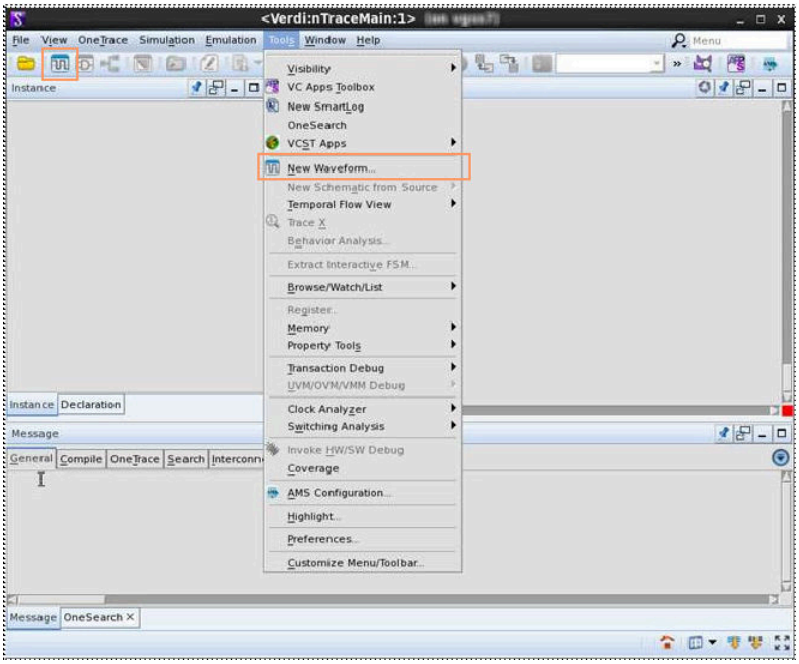
Note:

The Open ZTDB Waveform menu option is always enabled in nTrace.

Invoking the ZTDB Waveform From nWave

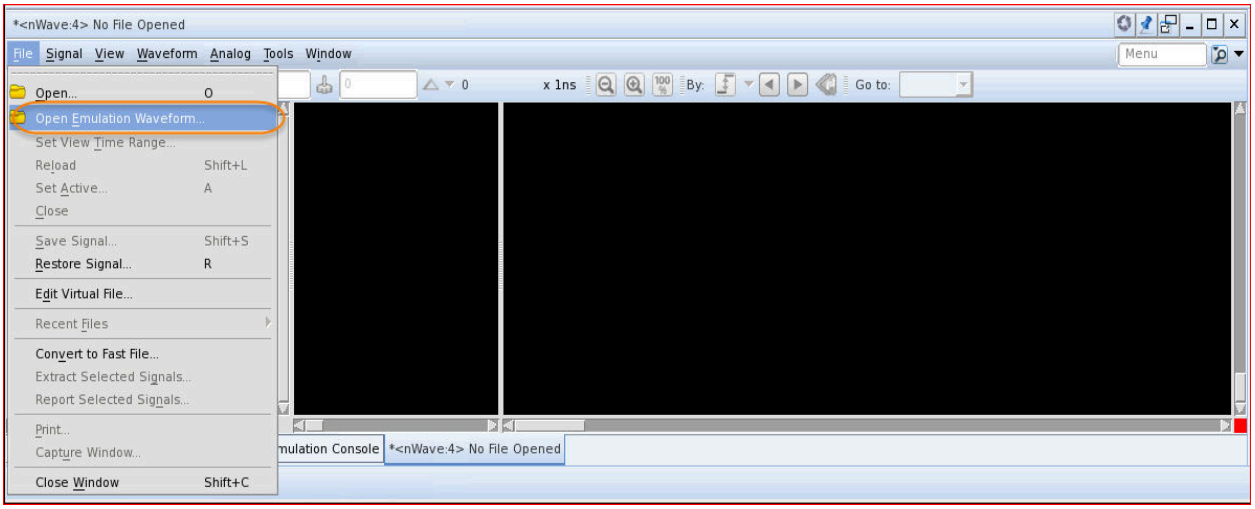
You can invoke the Verdi **nWave** window using **Tools > New Waveform** or the **New Waveform** icon, as highlighted in the following figure.

Figure 2 Open nWave Frame



You can invoke ZTDB waveform from **nWave** using **File > Open ZTDB Waveform**, as displayed in the following figure:

Figure 3 Open ZTDB Waveform From nWave



Note:

The Open ZTDB Waveform menu option is enabled only for a primary nWave window. If the primary nWave window has any opened files and you are trying

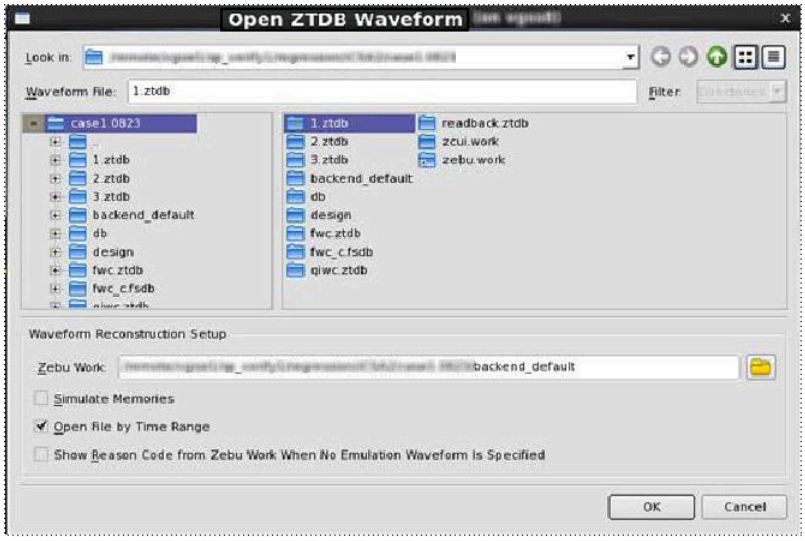
to open another instance of Open ZTDB Waveform, a message is displayed. You must click Yes to continue and close all files running in the primary nWave window are closed. The signals are restored from the new ZTDB file.

Viewing the ZTDB Waveform

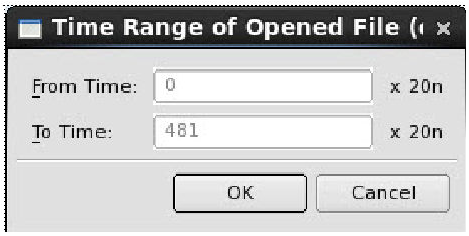
To view the waveform, perform the following steps.

1. Choose **Emulation > Open ZTDB Waveform** to open the ZTDB from the **nTrace** window.

   The **Open ZTDB Waveform** dialog box appears, as displayed in the following figure.



2. Browse to the `zebu.work` directory to specify the database and to load the waveform.
3. (Optional) To open waveform within a time range, select the **Open File by Time Range** check box. The **Time Range of Opened File** dialog box appears.



   Enter the start and end time of the ZTDB waveform that you want to open

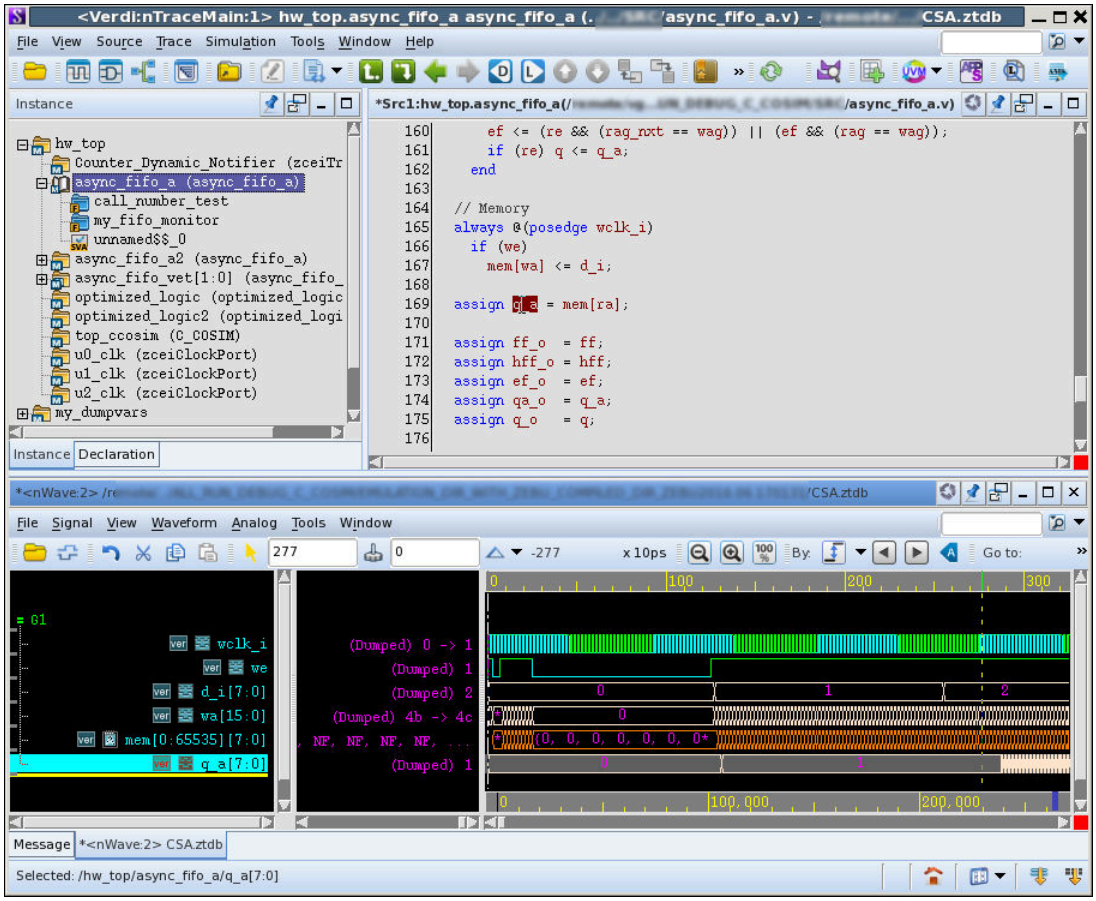
Feedback

- **From Time**: Specifies the start time of the ZTDB waveform file that you want to open. The ZTDB waveform displayed starts from the option you set with this option.
- **End Time**: Specifies the end time of the ZTDB waveform file that you want to open. The ZTDB waveform displayed ends from the option you set with this option.

After the waveform is displayed in **nWave**, you cannot change the time range settings because the ZTDB cannot be changed at runtime. Therefore, in nTrace the **Set View Time Range** option is disabled. To set the time range, repeat Step 3, that is select the **Open File by Time Range** check box from the **Open ZTDB Waveform** dialog box.

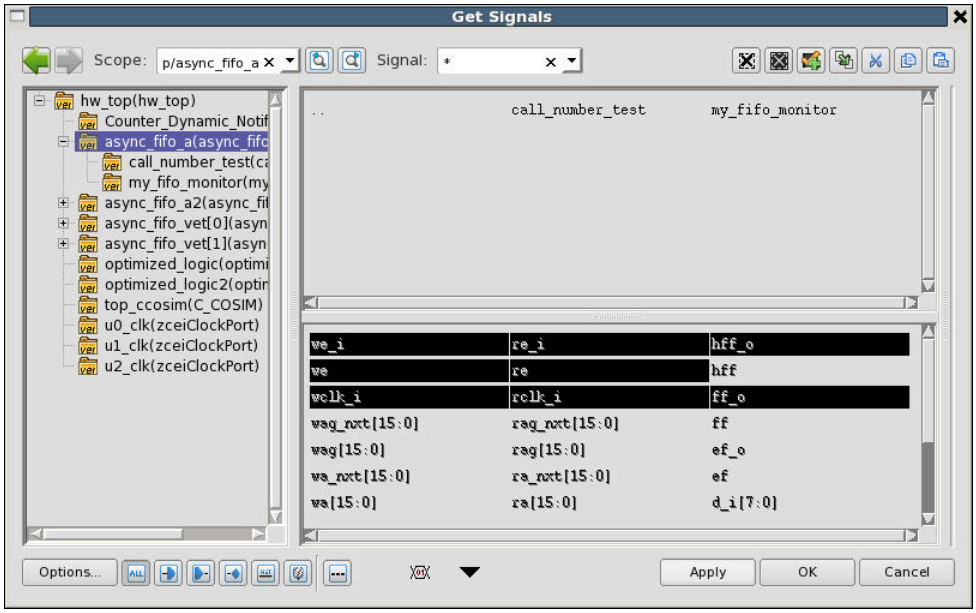
4. Click **OK**.

   The waveform appears in the **nWave** window.



5. On the **nWave** window, choose **Signal > Get Signals**.

   The **Get Signals** dialog box appears.



6. Select the signals to be added to the waveform, and then click **Apply** and **OK**.

   The **nWave** window displays the waveforms for the selected signals with the calculated signal value changes.

   Note:

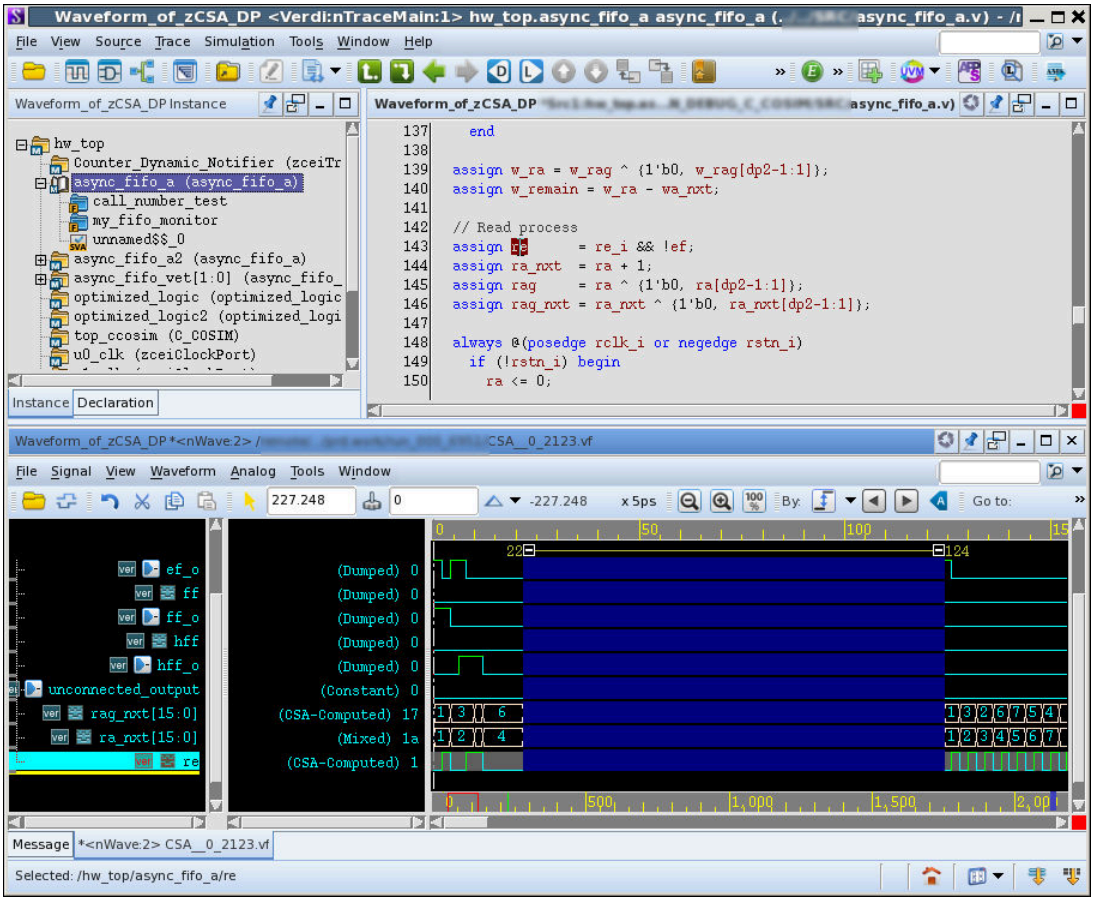
   If the time range for a signal is more than 60000 cycles, the signal value change is displayed as "NV" in the nWave window and nWave highlights the waveform in blue color, as displayed in the following figure.

7. To see the calculation waveform, perform one of the following steps:

   - Zoom in to smaller time range
   - Hold **Ctrl** key and drag left mouse button horizontally in ruler or signal waveform
   - Search next/previous value to force the server to perform the calculation.

     Note:

     If a signal has no calculated value change (NV for entire time range), its reason code is not displayed in the value pane and annotation.

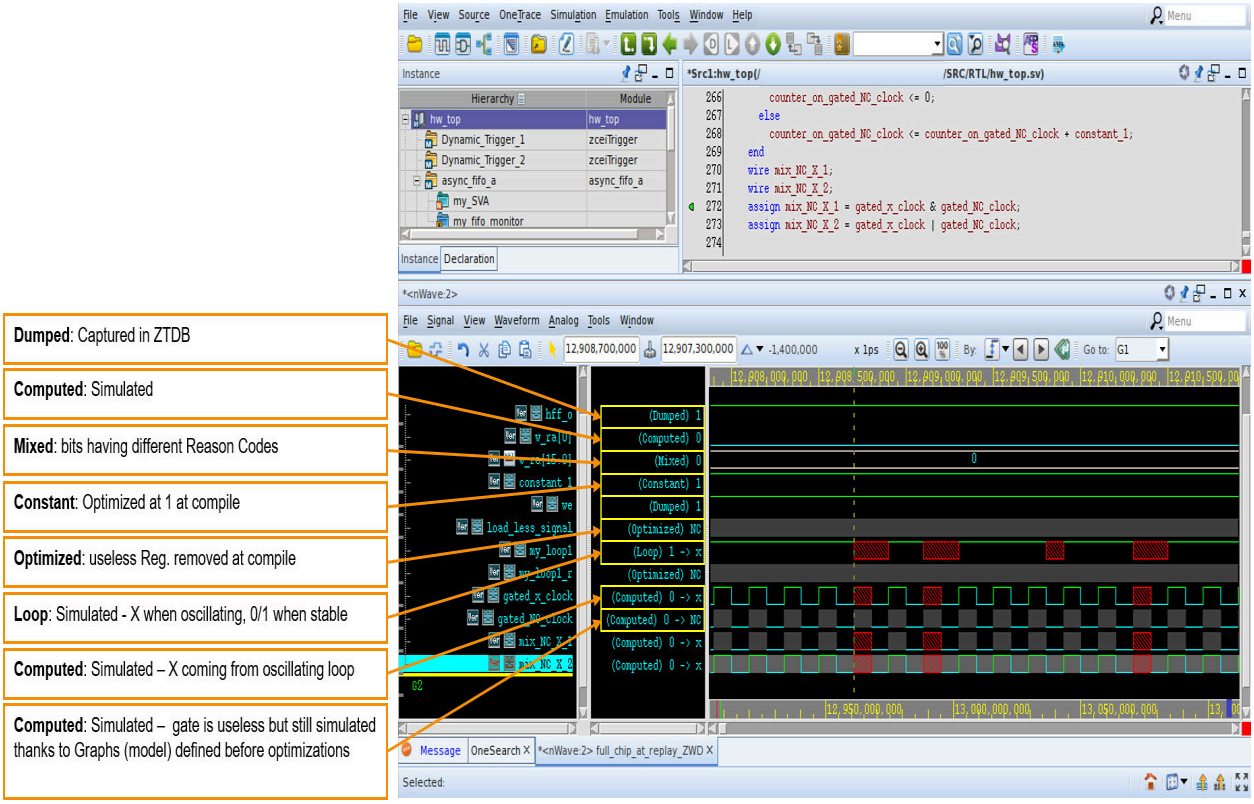


Waveform Reason Code

Each signal has a corresponding reason code and Table 1 lists the reason code for each signal. The following figure shows the waveform value for each signal.

Feedback

Figure 4 Example for Waveform Reason Code



Setting Preferences for Triggering nWave

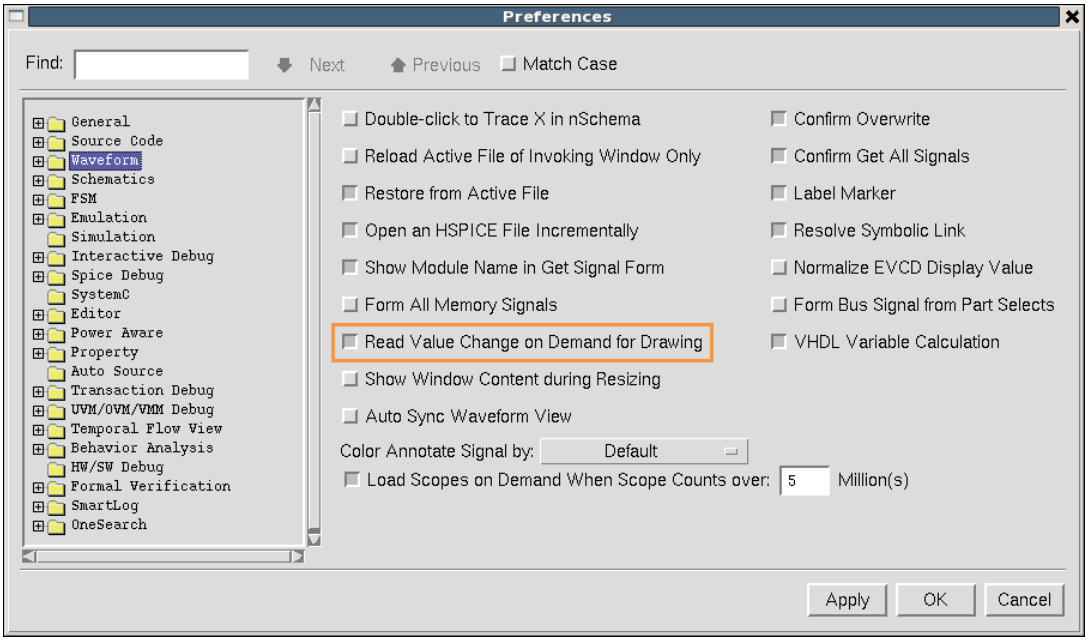
Table 1 Waveform Reason Codes

Waveform Value	Reason Code	Tooltip Description
Value	Constant	Signal is tied to a constant value at Compile Time. It can tied by VCS or zTopBuild
Value	Dumped	Signal was captured during emulation. Value is coming from ZTDB.
Value, X or NC	Computed	Value was computed by Waveform Expansion engine. X or NC values are results from upstream signals.
Value, X or NC	Loop	"Signal is part of an EDIF combinational loop.X value shows probable instable loops or is the result from upstream signals".

Table 1 Waveform Reason Codes (Continued)

Waveform Value	Reason Code	Tooltip Description
NC	Optimized	Signal has been optimized away during synthesis/compilation. Value cannot be computed by Waveform Expansion engine.
NC	Undriven	Signal has no driver. Value cannot be computed by Waveform Expansion Engine. Contact support.
NC or NF	Memory	Signal is mapped into a memory. It is not simulated.
X	Missing-Input	Signal is not in ZTDB file. Signal selection should be checked. Values cannot be computed by Waveform Expansion engine.
X	Missing-Accessibility	Failed to add accessibility during design compilation. Values cannot be computed by Waveform Expansion engine. Contact support.
X	RTL-Unresolved	Failed to perform RTL to EDIF resolution. Values cannot be computed by Waveform Expansion engine. Contact support.
X	RTL-Size-Mismatch	RTL size mismatch during RTL to EDIF resolution. Values cannot be computed by Waveform Expansion engine. Contact Synopsys support.

By default, **nWave** triggers the interactive Waveform Reconstruction engine to calculate the value change for all the added signals with viewable time range. You can trigger the interactive Waveform Reconstruction engine to calculate the value change only for the viewable signals (with viewable time range) when you scroll up or down in the signal or waveform frame. This can be achieved using the **Read Value Change on Demand for Drawing** check box available in **Tools** > **Preferences** in the **nWave** window. The following figure displays the **Preferences** dialog box and the available options.



Note:

Scrolling in the signal pane redraws the waveform only when you finish the scrolling. However, scrolling in the waveform pane redraws the waveform while scrolling. Therefore, you can scroll in the signal pane instead of the waveform pane to prevent performance issues.

Note:

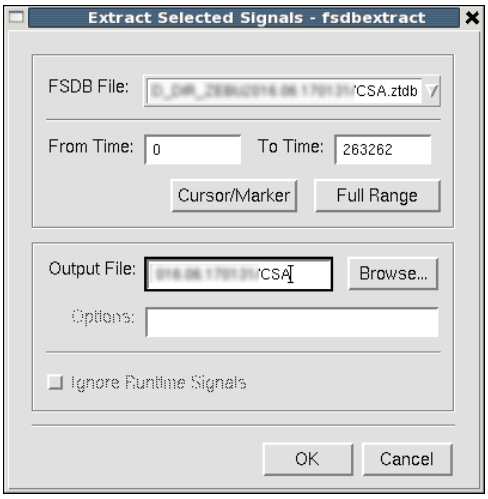
For reading ZTDB Waveform file, this flow is always enabled irrespective of the **Read Value Change on Demand for Drawing** check box.

Saving the Waveform as FSDB File

You can the waveform of selected signals with the specified time range into an FSDB file using **File** > **Extract Selected Signals** present in the **nWave** window.

You can also specify the time range in the **From Time** and **To Time** fields in the **Extract Selected Signals** form, as shown in the following figure.

Figure 5 Extract Selected Signals Dialog Box



The **From Time** and **To Time** fields have the default of file full time range.

Limitations

The following limitation exists when using a time range with ZTDB waveforms:

- Save/restore session does not support the specified time range

3

Viewing ZeBu Emulation Summary Result

An xterm console is supported in Verdi to invoke ZeBu simulation process (simz) and enter ZeBu simulation Unified Command Line (UCLI) commands to perform interactive debug.

This section describes the following subtopics:

Viewing the ZeBu Emulation Summary Result

Summary View support is integrated with the main **nTrace** window. When the **Emulation** > **Summary View** command of **nTrace** is selected, the **Summary View** frame and icons associated with Zebu emulation result summary are available in nTrace.

Note:

The **Emulation** > **Summary View** and the corresponding toolbar icon of nTrace is available if the `verdi -emulation` option has been specified when invoking Verdi.

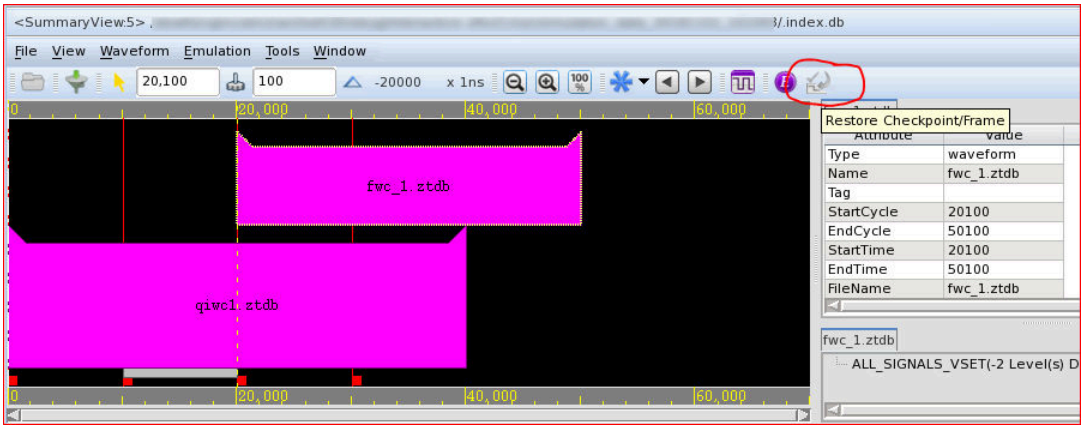
The emulation generates huge amount of results in different directories. To view and manage data, the emulation process dumps the summary of emulation result.

The **Summary View** menu option is introduced in **nTrace** window in Verdi to load the emulation database directory (`*.db`). This frame displays the summary of the emulation result with the corresponding attribute and the hierarchy of the selected objects. The waveforms in the **Summary View** frame can be selected and the corresponding details can be viewed in the **nWave**. In addition, you can rerun the emulation, dump the debug waveform, or debug the generated waveform in the **Summary View**. If new waveforms are captured or the database directory has new dumped results, the **Summary View** frame is updated accordingly only after reloading the database directory.

Feedback

The following figure displays an example of a loaded emulation database directory in the **Summary View**

Figure 6 Summary View



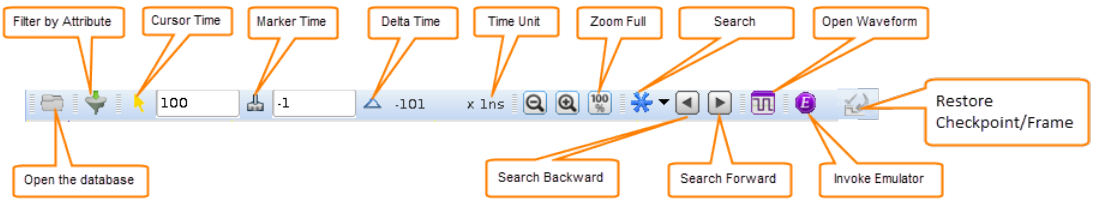
The following sections are available in the **Summary View** frame:

- Toolbar
- Summary pane
- Detail pane
- Hierarchy pane

Toolbar

The **Summary View** frame includes a toolbar that allows you to perform the various tasks as displayed in the following figure.

Figure 7 Summary View Toolbar



The **Search** toolbar option allows you to search various objects as listed in Summary Pane.

Feedback

Summary Pane

Table 2 Toolbar Search Icons

Toolbar Search Icon	Description
	Opens the **Open iDB Directory** form where the emulation result database directory (`*.db`) can be specified and the **Summary** pane can be invoked to load and display the summary of the database directory. Alternatively, use the **Emulation** > **Summary View** command in the **nTrace** window to open the **Open iDB Directory** form. **Note:** This command is disabled when a database directory is already opened.
	Opens the **Filter by Attributes** form where filter criteria can be set. Only the attributes that match the filter criteria are displayed in the **Summary** pane.
	Searches forward in time and locates the object by moving the cursor to the searched object in the **Summary** pane.
	Searches backward in time and locates the object by moving the cursor to the searched object in the **Summary** pane.
	Provides a close-up view of the contents in the **Summary** pane. The magnification of the viewing area is changed to half the magnification of the previous view. **Note:** A specific area can be zoomed by dragging-left to form a rectangle around the area to be zoomed.
	Enables more of the contents to be seen in the **Summary** pane at a reduced size. The magnification of the viewing area is changed to two times of the magnification of the previous view from the center point .
	Displays all of the contents of the **Summary** pane.
	Invokes the **nWave** window for the specified waveform object in the **Summary** pane. After the waveform is opened in **nWave**, the specified waveform object in the **Summary** pane is marked in blue. **Note:** This command is enabled only when a waveform object is specified in the **Summary** pane.

Feedback

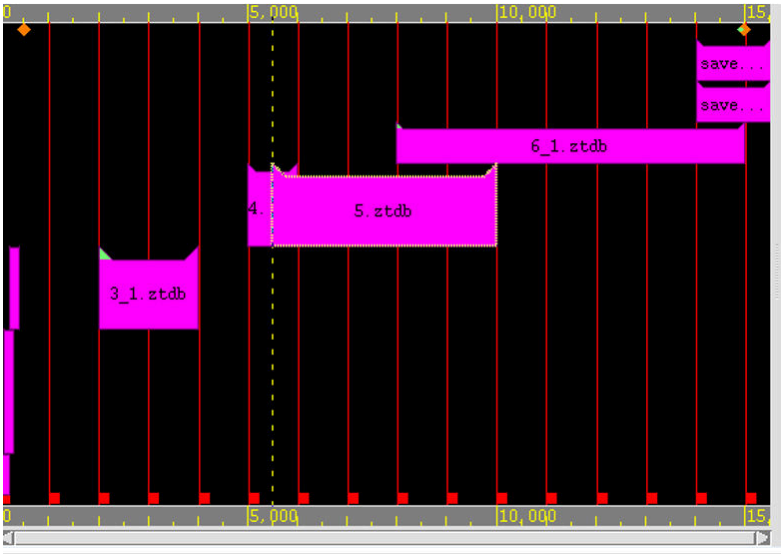
Table 2 Toolbar Search Icons (Continued)

Toolbar Search Icon	Description
	Enables you to set the cursor position in the current **Summary** pane.
	Enables you to set the marker position in the current **Summary** pane.
	Displays objects during the delta time in the **Summary** pane. The delta time between the cursor and marker is displayed on the right of this icon. That is, **Delta = Marker - Cursor**.
	Displays the time unit of the database.
	Opens the **Run Emulation** form where the UCLI command and environment settings can be configured to run the emulation in the **Summary** pane.
	Searches any objects.
	Searches checkpoint objects.
	Searches event objects.
	Searches waveform objects.
	Opens the **Search by Attribute** form where the search criteria can be set.
	Restores the Checkpoint or frame.

When the ZeBu emulation result database directory (`*.db`) is selected from the **Open iDB Directory** dialog box, the **Summary** pane loads and displays all the objects with full time range, by default.

Feedback

Figure 8 Summary Pane in the Summary View Frame



You can zoom in/out using the toolbar options or dragging the mouse. The following types of objects can be viewed in the **Summary** pane.

- Checkpoint objects are displayed as red lines.
- Event objects are displayed as orange diamonds.
- Waveform objects are displayed as purple boxes.

Note:

Only one Summary pane can be invoked and only one Zebu emulation result database directory can be viewed in the pane at a time.

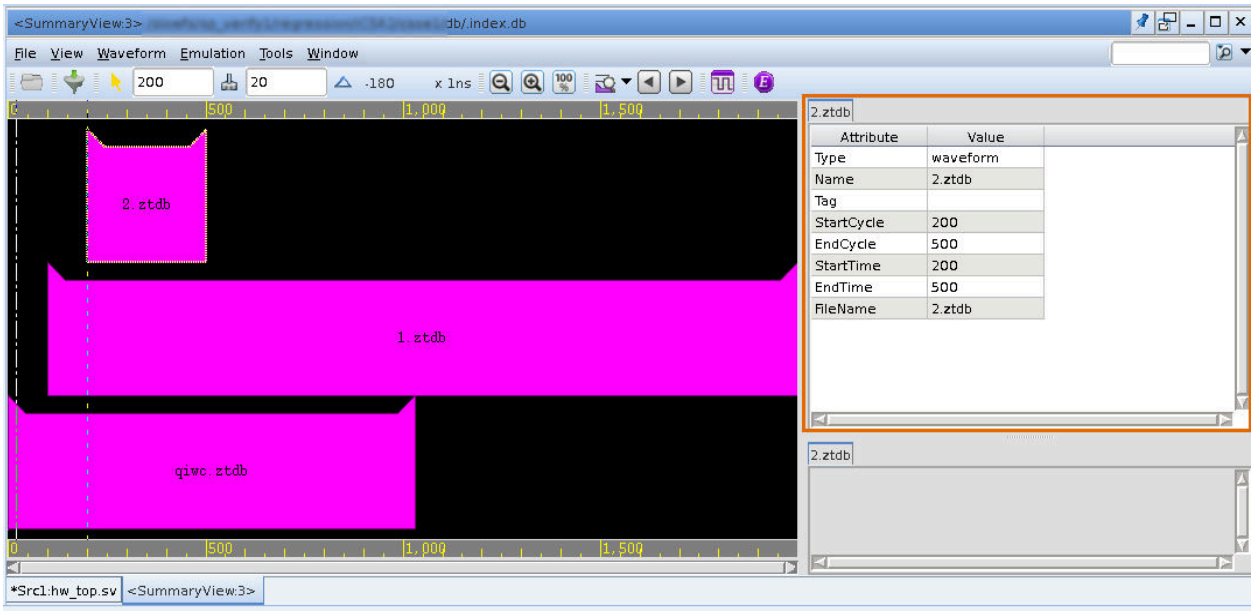
Detail Pane

The **Detail** pane displays the attribute values of a selected Checkpoint/Event/Waveform in the **Attribute** and **Value** columns of the **Summary** pane. The tab name follows the object name. If multiple objects are overlapped in the Summary pane, click the overlapped area to open the **Detail** pane.

The following figure displays the **Detail** pane in the **Summary View** frame.

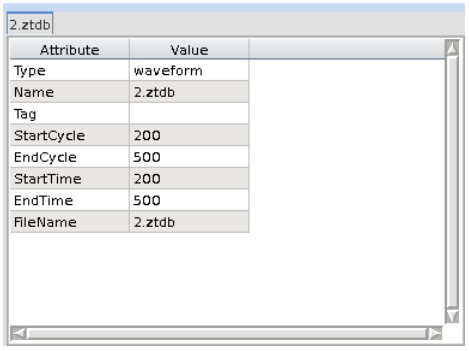
Feedback

Figure 9 Detail Pane in Summary View Frame



The **Detail** pane displays multiple tabs and each tab displays details for each overlapped object. You can display or hide the **Detail** pane using the **View** > **Show/Hide Detail Pane** menu option. The **Detail** pane has **Attribute** and **Value** columns.

Figure 10 Detail Pane



The following attributes are displayed in the **Attribute** column:

- Type
- Name
- Tag
- StartCycle
- EndCycle

Feedback

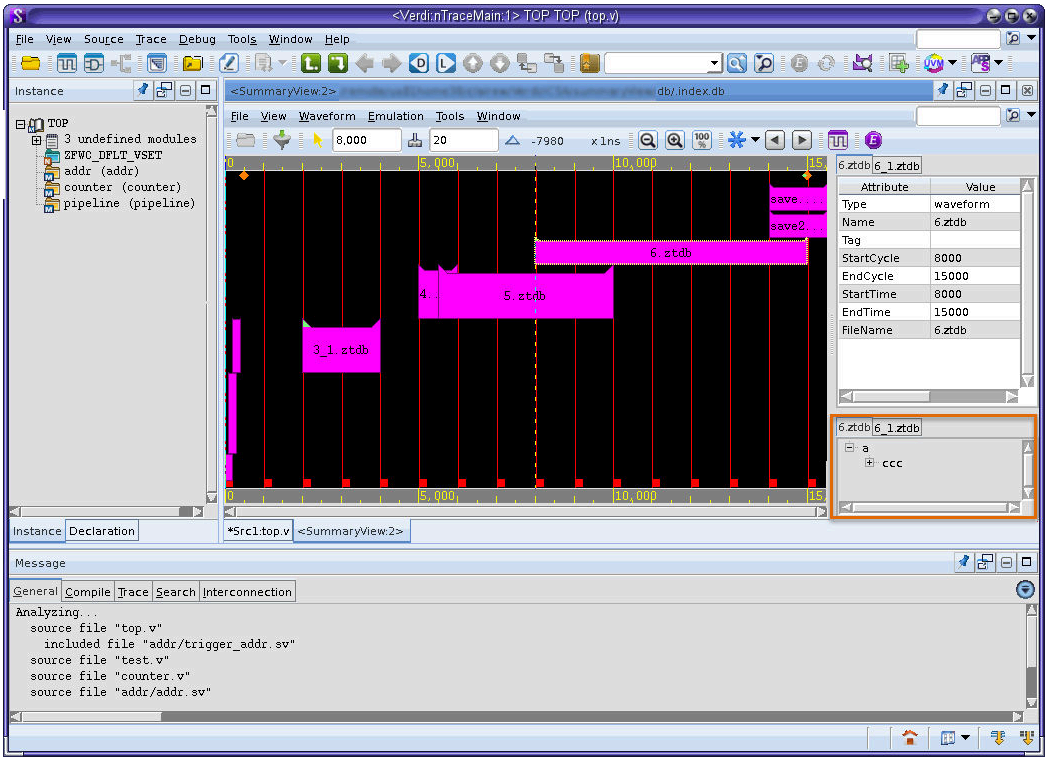
- StartTime
- EndTime
- FileName

Hierarchy Pane

The **Hierarchy** pane displays the hierarchy tree of a selected waveform, and it is located just below the **Detail** pane. If a checkpoint or an event is selected, there is no hierarchy tree to view and the tab is left as empty.

The following figure displays the **Hierarchy** pane in the **Summary View** frame.

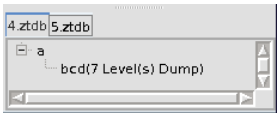
Figure 11 Hierarchy Pane in the Summary View Frame



If the waveforms overlap in the **Summary** pane, click the overlapped area to view the **Hierarchy** pane. The **Hierarchy** pane displays multiple tabs with each tab displaying the hierarchy tree for each object. You can use the **View** > **Show/Hide Hierarchy Pane** menu option to display or hide the **Detail** pane.

Feedback

Figure 12 Hierarchy Pane



Note:

The **Open** command is available before the database directory is loaded.

4

Viewing Emulation Capacity in Verdi

Verdi enables you to analyze emulation capacity by providing capacity utilization statistics. The statistics pertain to hierarchies and modules in the design that have the highest emulation cost. As a result, you can efficiently identify the critical parts impacting of a design.

For more information, see the following subsections:

Prerequisites

Before launching Verdi, ensure the following:

- Import *zebu.work* with the capacity report generated by ZeBu
- Add the following option in the *zTopbuild* command in the UTF file:

```
ztopbuild -advanced_command {enable generate_zvdb_emucap}
```

Viewing Emulation Capacity

To view the emulation capacity, perform the following steps:

1. Launch Verdi. The Verdi home screen appears. For example, the following command launches Verdi:

   ```
   verdi -emulation --zebu-work zcui.work/backend_default
   ```

2. Choose **Emulation** > **Capacity Visualization**. The Emulation Capacity window appears. From this window, you can either use the features provided to view capacity related information or you can save the information to a CSV file for visualization in a spreadsheet tool.

Feedback

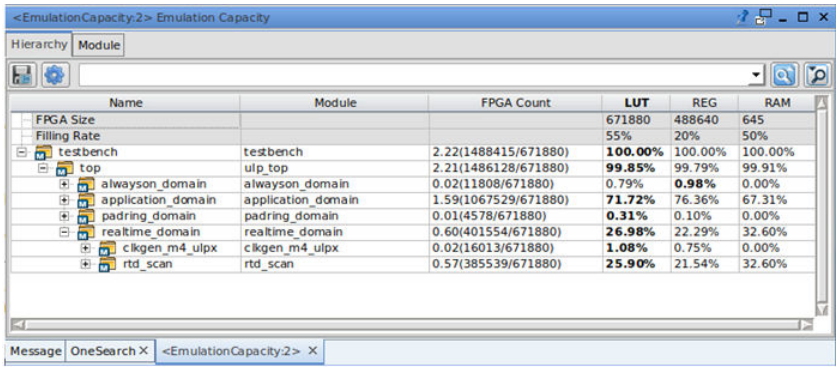
By using the Emulation Capacity window, you can perform the following:

- Viewing Capacity for Hierarchies
- Viewing Capacity for Modules

Viewing Capacity for Hierarchies

By default, the **Hierarchy** tab displays the following columns: Name, Module, FPGA Count, LUT, REG, and RAM columns. Click the **Configure** icon to display other columns, such as LUT6, DSP, RAMLUT, FWC_IP_NUM, and FWC_IP_BITS. The following screen shot shows the **Hierarchy** tab.

Figure 13 Hierarchy Tab of the Emulation Capacity Window



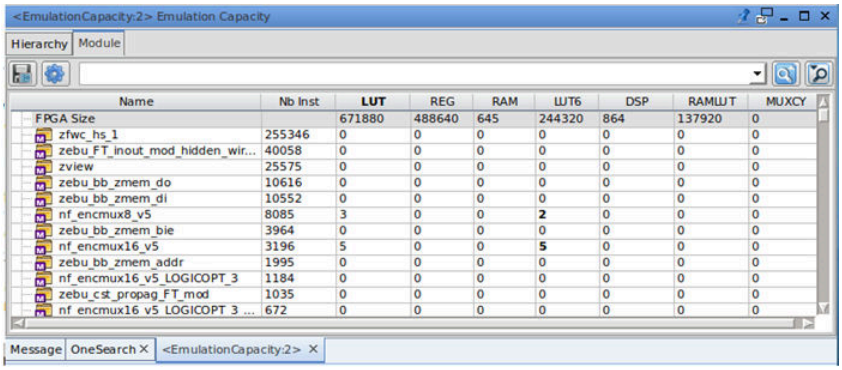
The FPGA Size and FPGA Filling Rate for the hierarchy are displayed in the top two rows. In addition, the most used resource is highlighted through bold text. In the figure above, the most used resource is LUT. You can organize the data by sorting and searching for a specific hierarchy.

Viewing Capacity for Modules

By default, the **Module** tab displays the following columns: Name, Nb Inst, LUT, REG, RAM, LUT6, DSP, RAMLUT, and MUXCY columns. Click the *Configure* icon to display other columns, such as ALL REG, ALL LUT, ALL RAM, ALL DSP, ALL RAMLUT, ALL LUT6, and ALL MUXCY. The following screenshot shows the **Module** tab.

Feedback

Figure 14 Module Tab of the Emulation Capacity Window



The first row displays the total FPGA Size. The other rows show the statistics for each module. The most used resource is highlighted through bold text. You can organize the data by sorting and searching for a specific module.

Using Tcl Commands for Viewing Capacity Utilization

The following table shows the Tcl commands supported while viewing capacity utilization.

Command	Description
`zcvAppNewWin`	Open capacity window and load hierarchy and module report.**Value Returned**: windowID**Example** `zcvAppNewWin`
`zcvAppCloseWin`	Close capacity window.**Value Returned**: 1 for success; 0 for failure**Example** `zcvAppCloseWin`

Search Hierarchy Tab
`zcvSearchHierTree -pattern user_pattern -next|-previous`
Search Module Tab
`zcvSearchModTree -pattern user_pattern -next|-previous`
	Search string in Hierarchy tab or Module tab.*Options*:-pattern: Specify the search pattern-next: Search forward-previous: Search backward**Value Returned**: 1 for success; 0 for failure**Example**
	`zcvSearchHierTree -pattern "a*" -next`
	`zcvSearchModTree -pattern "b" -previous`

Save Hierarchy Report
`zcvSaveHierReport filename`
Save Module Report
`zcvSaveModReport filename`
	Save Hierarchy report or Module report.**Value Returned**: 1 for success; 0 for failure**Example**
	`zcvSaveHierReport "file/capacity_hier.csv"`
	`zcvSaveModReport "file/capacity_mod.csv"`

Feedback

Command	Description
`zcvHierTreeConfigColumns [-lut on|off] [-reg on|off] [-ram on|off] [-lut6 on|off] [-dsp on|off] [-ramlut on|off] [-fwc_ip_num on|off] [-fwc_ip_bits on|off] [-all on|off]`	Configure Hierarchy tab columns.**Options**-lut: Show or hide LUT column-reg: Show or hide REG column-ram: Show or hide RAM column-lut6: Show or hide LUT6 column-dsp: Show or hide DSP column-ramlut: Show or hide RAMLUT column-fwc_ip_num: Show or hide FWC_IP_NUM column-fwc_ip_bits: Show or hide FWC_IP_BITS column-all: Show all columns or hide all columns except Name, Module, and FPGA Count**Value Returned**: 1 for success; 0 for failure**Example** `zcvHierTreeConfigColumns -dsp on -reg off`
`zcvModTreeConfigColumns [-lut on|off] [-reg on|off] [-ram on|off] [-lut6 on|off] [-dsp on|off] [-ramlut on|off] [-all_reg on|off] [-all_lut on|off] [-all_ram on|off] [-all_dsp on|off] [-all_ramlut on|off] [-all_lut6 on|off] [-all_muxcy on|off] [-all on|off]`	Configure Module tab columns**Options**-lut: Show or hide LUT column-reg: Show or hide REG column-ram:Show or hide RAM column-lut6: Show or hide LUT6 column-dsp: Show or hide DSP column-ramlut: Show or hide RAMLUT column-all_reg: Show or hide ALL REG column-all_lut: Show or hide ALL LUT column-all_ram: Show or hide ALL RAM column-all_dsp: Show or hide ALL DSP column-all_ramlut: Show or hide ALL RAMLUT column-all_lut6: Show or hide ALL LUT6 column-all_muxcy: Show or hide ALL MUXCY column-all: Show all columns or hide all columns except Name and Nb Inst**Value Returned**: 1 for success; 0 for failure**Example** `zcvModTreeConfigColumns -all_dsp off -dsp on`

Limitations

The following limitations exist with this feature:

- Save/restore is not supported.
- Data in Emulation Capacity View is not refreshed on Verdi Reload.
- Emulation Capacity View does not display all hierarchies and modules in the design. Only hierarchies and modules exceeding a defined threshold is displayed.

5

Viewing Design Loops in Verdi

Design loops complicate both functionality and debugging. To identify design loops, Verdi provides the loop visualization. For more information, see the following subsections:

Prerequisites

Before launching Verdi, ensure the following:

- Import `zebu.work` with the loop report generated by ZeBu.

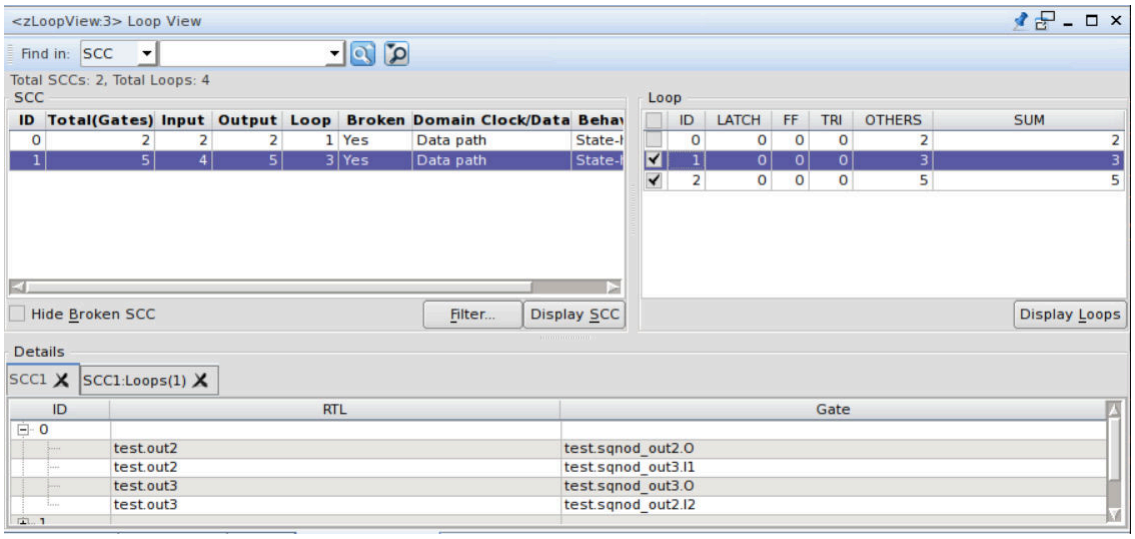
Feedback

Viewing Design Loops

To view design loops, perform the following steps:

1. Launch Verdi. The Verdi home screen appears.
2. Choose **Emulation** > **Loop Visualization**. The Loop View window appears.

Figure 15 Loop View Window



The Loop View window provides information in the following groups:

- **SCC**: Displays the Strongly Connected Components (SCC) in a table with the following columns: ID, Total (Gates), Input, Output, Loop, Broken, Domain Clock/Data, and Behavior. Select **Hide Broken SCC to view SCC** that are not broken.

Note:

If there are no SCCs in the design, the table is empty or Hide Broken SCC is selected

- **Loops**: Displays loop information for the selected SCC. The information is displayed in a table with the following columns: ID, LATCH, FF, TRI, OTHERS, and SUM.
- **Details**: Displays the selected SCC or loops in individual tabs. The ID, RTL, and Gate columns are shown.

1. Select an SCC from the **SCC** group. The **Loops** group autopopulates loops information pertaining to the selected group.
2. To view more information on the selected SCC, click **Display SCC**. The information is displayed in the **Details** group. To save the results, right-click the tab and choose **Save**.
3. To view more information on the loops present in the SCC, select the loops in the **Loops** group and click **Display Loops**. The information is displayed in the **Details** group. To save the results, right-click the tab and choose **Save**.

Applying Criteria to Identify Loops in an SCC

Identifying specific loops in an SCC, which has several loops, can be challenging. To make it easier, from the **Loop View** window you can access the **Filter Loop to Display in Details** dialog to see loops matching specified criteria and then display them in the **Details** group. For more information, see the following subsections:

- Defining the Criteria
- Finding Loops

Defining the Criteria

To define the criteria, each criterion has a condition that is chained with an `AND` or an `OR` connector. The conditions use operators to match the loops that have a specified type (SCC_ID, LATCH_NUM, FF_NUM, TRI_NUM, OTHER_NUM, PATTERN).

For example, suppose in `SCC 0` you want to identify all loops that have more than 10 latches. You can specify the following filtering criteria:

```
SCC_ID == 0 AND LATCH_NUM >= 10
```

Similarly, you can use other operators `==` and `<=` with the SCC_ID, LATCH_NUM, FF_NUM, TRI_NUM, and OTHER_NUM types.

The PATTERN type enables you to filter for loops based on a pattern. The *is* operator is the only operator supported by the PATTERN type. For example, suppose you need to search within the a*.b*.c hierarchy. In this case, the a1.b1.c hierarchy is matched, but a1.b1.c.d and top.a1.b1.c hierarchies are not matched.

Similarly, if no hierarchy is specified, such as ab*, then top.abc and abc.d are matched.

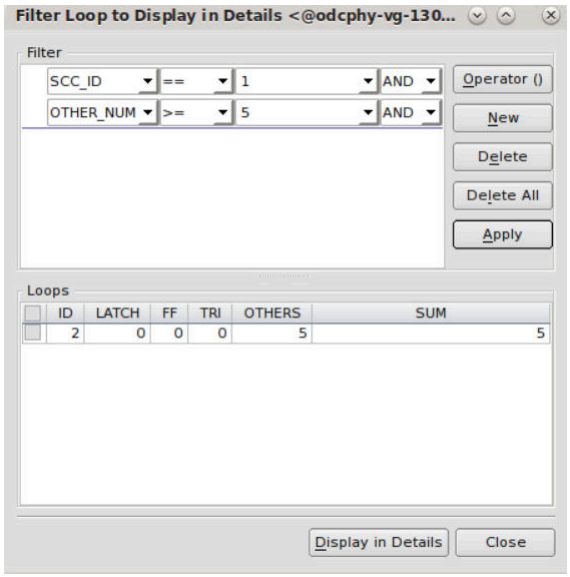
Feedback

Finding Loops

To identify loops based on specified criteria, perform the following:

1. Click **Filter** in the **Loop View** window. The **Filter Loop to Display in Details** dialog appears. The first row is prepopulated with the type as SCC_ID, the == operator, and the `AND` connector.

Figure 16 Filter Loop to Display in Details Dialog



2. Select the SCC ID type to specify the SCC for which you want to apply the criteria. You have specified the first criterion.
3. Click **New** to specify another criterion.
4. Specify the criterion by choose the type, operator, and connector. Similarly, other criterion are chained together.
5. (Optional) You can add parenthesis to the condition by clicking the **Operator ()** button. For example, if you want to create complex filtering condition such as:
6. Click **Apply** to view the loops that match the criteria.

```
SCC_ID == 1 AND (TRI_NUM >= 7 OR FF_NUM <= 5)
```

7. Click **Display in Details** to view the loops in the **Details** group.

Feedback

Using Tcl Commands for Viewing Design Loops

The following table shows the Tcl commands supported while viewing design loops.

Command	Description
`zlvNewWin`	Open the Loop View window. Only one window can be opened.**Value Returned**: windowID**Example** `zlvNewWin`
`zlvCloseWin`	Close the Loop View window.**Value Returned**: 1 for success; 0 for failure**Example** `zlvCloseWin`

Search Next
`zlvSearchNext -type SCC|Loop|Details -value pattern`
Search Previous
`zlvSearchPrev -type SCC|Loop|Details -value pattern`
	Search next or previous value in the Loop View window.*Options*:-type: Specify the table to search-value: Specify the search pattern**Value Returned**: 1 for success; 0 for failure**Example**
	`zlvSearchNext -type SCC -value "data"`
	`zlvSearchPrev -type SCC -value "data"`

`zlvSelect`
`[-scc row_id] | [-loop row_id] | [-detail -row row_id`
`[-port port_id]`
`[-column column_id]]`
	Select item in the Loop View window.**Options**-scc: Select the row ID in SCC table.-loop: Select the row ID in Loop table.-detail: Select the item in Details table.-row: Specify the row ID of parent loop node in Details table.-port: Specify the port ID under parent loop node in Details table.-column: Specify the column ID in Details table.**Value Returned**: 1 for success; 0 for failure**Example**
	`zlvSelect -scc "1"`

`zlvSetOptions -hideBrokenSCC on|off`	Set option in the Loop View window.**Options**-hideBrokenSCC: Enable or disable Hide Broken SCC.**Value Returned**: 1 for success; 0 for failure**Example** `zlvSetOptions -hideBrokenSCC off`

`zlvDisplayLoops`
`-scc scc_id`
`[-loop loop_id_list]`
	Display SCC or loops.**Options**-scc: Specify SCC ID.-loop: Specify the loop ID list; Display all loops in this SCC if loop ID list is not specified.**Value Returned**: 1 for success; 0 for failure**Example**
	`zlvDisplayLoops -scc 1 -loop {0 1}`

`zlvFilter -show on|off`	Display Filter Loops to Display in Details dialog in Loop View window.**Options**-show: Display the Filter Loops to Display in Details dialog.**Value Returned**: 1 for success; 0 for failure**Example** `zlvFilter -show on`

Feedback

Command	Description

Switch Active Details Tab
`zlvChangeTab tab_name`
Close Details Tab
`zlvCloseTab tab_name`
Save Loops on Active Details Tab
`zlvSaveLoop file_name`
	For changing the active Details tab, closing the Details tab, or saving loops displayed on the active Details tab.**Value Returned**: 1 for success; 0 for failure**Example**
	`zlvChangeTab "SCC1"`
	`zlvCloseTab "SCC1"`
	`zlvSaveLoop "loop.txt"`

`zlvDumpAU`
`-source SCC|Loop|Details`
`[-All | -selected]`
	Save information of the Loop View window.**Options**-source: Specify the table to save.-all: Save all items.-selected: Save the selected item.**Value Returned**: 1 for success; 0 for failure**Example**
	`zlvDumpAU -source SCC -all`

Limitations

The following limitations exist with this feature:

- Save and restore is not supported.
- Data in loop tables is not refreshed on Verdi Reload.

6

Viewing ZeBu Internal Gate-Level Netlists in Verdi

Verdi GUI provides a schematic netlist debug flow for the ZeBu internal gate-level netlist (ZNL) generated by ZeBu synthesis. To find, search, and trace the ZNL netlist, Verdi reads the ZNL netlist and provides an instance-tree view, a signal pane view, and a *nSchema* schematic window.

The **Instance (Hierarchy View)** window in *nTrace* displays the hierarchy of the ZNL netlist. When you select the required scope, the related signals under this scope display the HDL design on the **Signal** pane. The string (signal/instance) search function is also supported on the ZNL netlist. Therefore, you can create or add an instance/net into *nSchema* to view the schematic of the ZNL netlist.

This section describes the following subtopics.

Viewing ZNL Netlists in nTrace

The **nTrace** window supports ZNL netlist display, that is, you can open ZNL netlist in the **nTrace** window.

To import a ZNL file, perform the following steps:

1. Specify the `-emulation` command-line option in Verdi.
2. Click **Emulation** > **Load Netlist** in **nTrace**.

   The **Load Netlist Files** form opens.

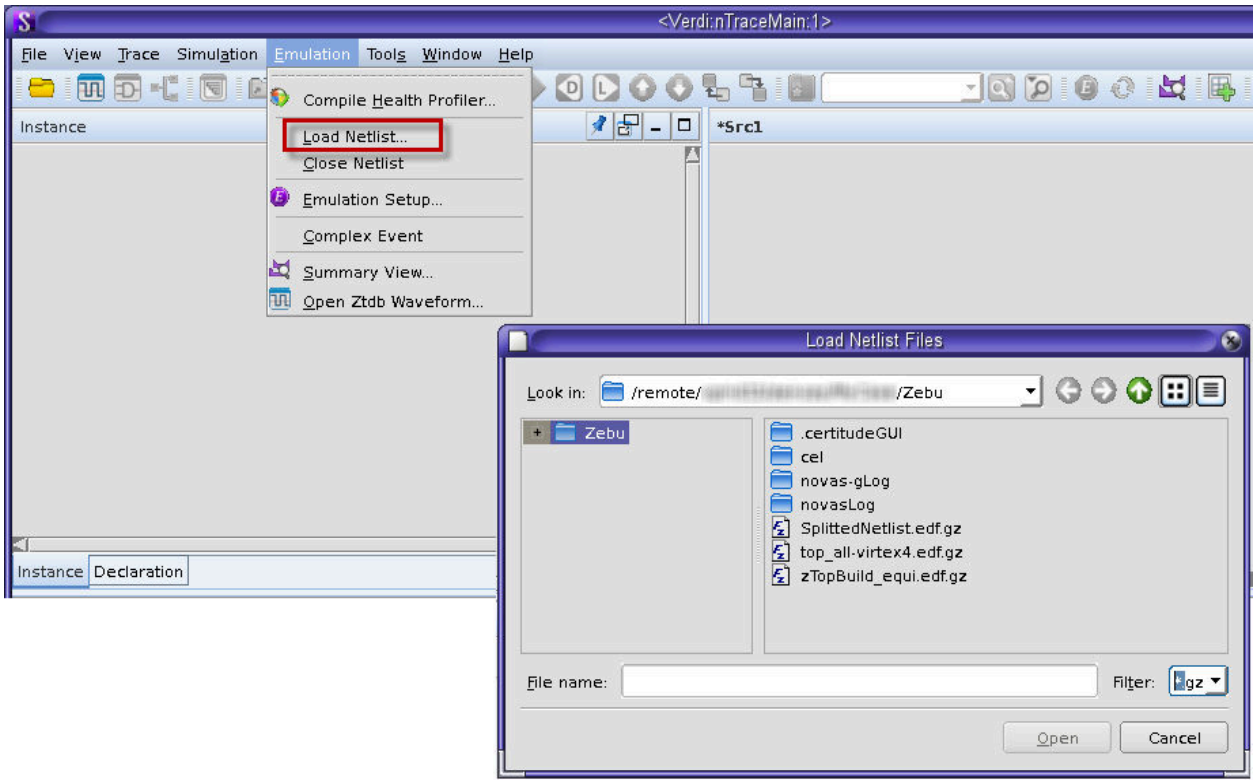
   You can also load the ZNL netlists using the command line option, `-znl_file`, as follows:

   ```
   novas -emulation -ultra -lca -znl_file topBuild.edf.gz
   ```

Feedback

3. Specify the file to be opened in the **File name** field and then click **Open**.

Figure 17 Load Netlist Files Form



Note:

To close the ZNL design, you can use **Emulation > Close Netlist** in nTrace.

Netlist Hierarchy Pane

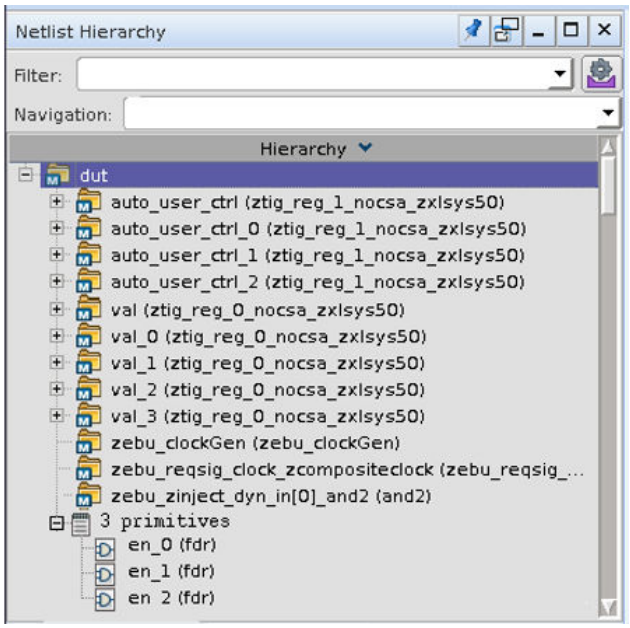
The **Netlist Hierarchy** pane displays the instances and primitives present in the netlist. The instances are represented by the icon and the primitives are represented by the icon.

The primitives are grouped under another node named as "# primitives", where the # sign displays the total number of primitives. By default, the primitives are displayed. To display or hide all primitives, use **Display/Hide Xilinx Primitive**. You can also use the context menu command to display primitive for each node.

Feedback

The following figure illustrates the netlist hierarchy pane:

Figure 18 Netlist Hierarchy Pane



The **Filter** field supports wildcards and case insensitive matching by default and supports the following:

- To filter the hierarchy display by string, enter the string in the **Filter** field in the **Netlist Hierarchy** pane.
- To filter by type, you can toggle on/off the primitives to display primitives in the hierarchy tree.

The full hierarchy name of the selected node is displayed in the **Navigation** field. To find a specific tree node, enter the full hierarchy name in the **Navigation** field in the **Netlist Hierarchy** pane and press **Enter**. You must specify "." as a delimiter in the **Navigation**field.

To sort the hierarchy display by name (ascending/descending), left-click/left-mouse- button (LMB) on the **Hierarchy** field in the **Netlist Hierarchy** pane.

The following right-click/right-mouse-button options are supported in the **Netlist Hierarchy** pane:

- **Expand Node by Level**: Expands the selected node by a specified number level (choose the number level in the submenu).
- **Collapse All Nodes**:Collapses the complete hierarchy tree to two-level.

Feedback

- **New Full Schematic**: Opens the schematic window based on the active node.
- **Show Primitive**: This option is only active when selected on "# primitives" node. It displays all primitives under the instance.

Netlist Signal List Pane

The **Netlist Signal List** pane displays all ports or wires under the selected hierarchy or scope. To switch displaying ports or wires in the **Netlist Signal List** pane, click the  icon.

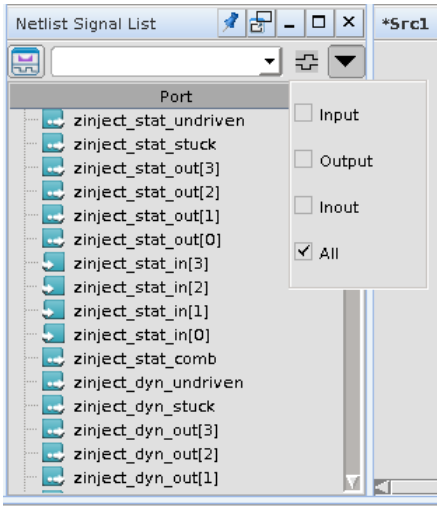
The following icons represent the different types of ports:

- Input port - 
- Output port - 
- Inout port - 
- Wire - 

Verdi displays the signal name with the range information as the label of a signal in the **Netlist Signal List** pane.

The following figure illustrates the **Netlist Signal List** pane:

Figure 19 Netlist Signal List Pane



The **Filter** field supports wildcards and case insensitive matching by default.

If you drag-and-drop a signal into the **Netlist Signal List** pane, it highlights the signal (may change scope first if dropped signal is in a different hierarchy).

The following right-click/right-mouse-button options are supported in the **Netlist Signal List** pane:

- **New Schematic** > **Driver**: Opens the schematic window and displays the driver result of the selected signal.
- **New Schematic** > **Load**: Opens the schematic window and displays the load result of the selected signal.
- **New Schematic** > **Connectivity**: Opens the schematic window and displays both driver and load result of the selected signal.

Filtering Ports and Netlists

To filter the netlist signal display by string, enter the string in the **Filter** field in the **Netlist Signal List** pane and press `Enter`.

To filter the ports by type, click the  icon and toggle on/off the Input/Output/Inout/All options to display/hide the ports.

Sorting the Signal List

To sort the signal list display by name (ascending/descending), left-click/left-mouse-button click the **Port** field in the **Netlist Signal List** pane.

Limitations

As the hierarchy-tree is based on the ZNL netlist, it might not be the same as the original design, which is based on HDL (SV, Verilog or VHDL). Until the emulation design is exposed to gate-level mapping database, Verdi has no way to associate the design to the ZNL netlist.

Viewing ZNL Netlists in nSchema

The **nSchema** window supports the ZNL netlist display, that is, you can open full (modular)/flatten (trace result) window for ZNL netlist.

The **Full Schematic** window displays one hierarchy netlist at one time. You can double-click hierarchy instance to view the complete netlist.

The **Full Schematic** window displays instances of different scopes at one time. You can double-click the instance pin to expand its driver/load. This window has the hierarchy mode, which displays the hierarchy boundary.

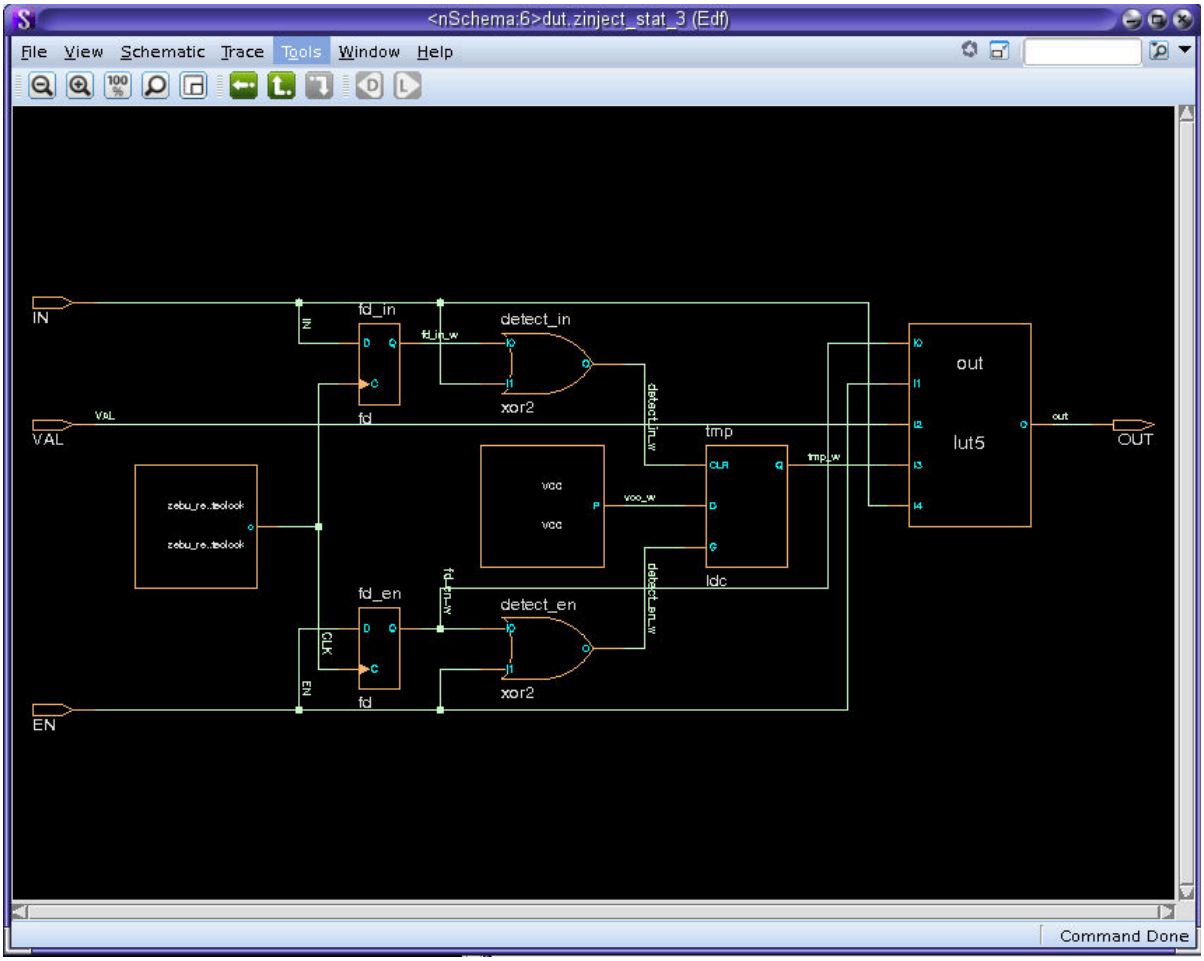
Feedback

Full Schematic Window

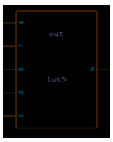
To open the **Full Schematic** window for ZNL, click the **New Full Schematic** right-click menu option in the **Netlist Hierarchy** pane in **nTrace**.

The following figure illustrates the **Full Schematic** window for ZNL:

Figure 20 Full Schematic View



In the **Full Schematic** window, the following are the schematic representations:

Image	Description
	**Module Instance** Double-click the module instance to push the view in the lower scope

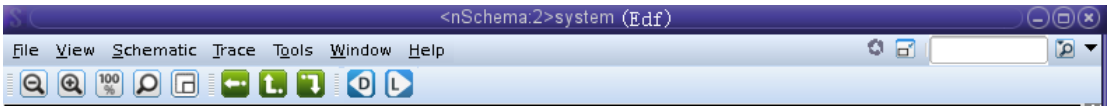
Feedback

Image	Description
	**Primitive Instance** This is the leaf instance that cannot be pushed to view the lower scope
	**Module Port** Double-click the module port to jump to the upper scope and highlight the upper signal

Toolbar

The following figure illustrates the toolbar for **nSchema** ZNL window:

Figure 21 Toolbar



The toolbar options (from left to right) are as follows:

- **Zoom Out**: Performs zoom out. You can also perform zoom out using "lower-to-upper right" mouse gesture by clicking the mouse's left button.
- **Zoom In**: Performs zoom in. You can also perform zoom in using "upper-to-right-lower" mouse gesture by clicking the mouse's left button.
- **Zoom All**: Performs 100% zoom. You can also perform zoom all using "upper-to-left-lower" mouse gesture by clicking the mouse's left button.
- **Magnifier**: Displays the magnifier glass.
- **Mini Map**: Displays the min-map of the current window.
- **Last View**: Displays the last view.
- **Go Up**: Navigates to the upper scope.
- **Go Down**: Navigates to the lower scope for the selected hierarchy instance.

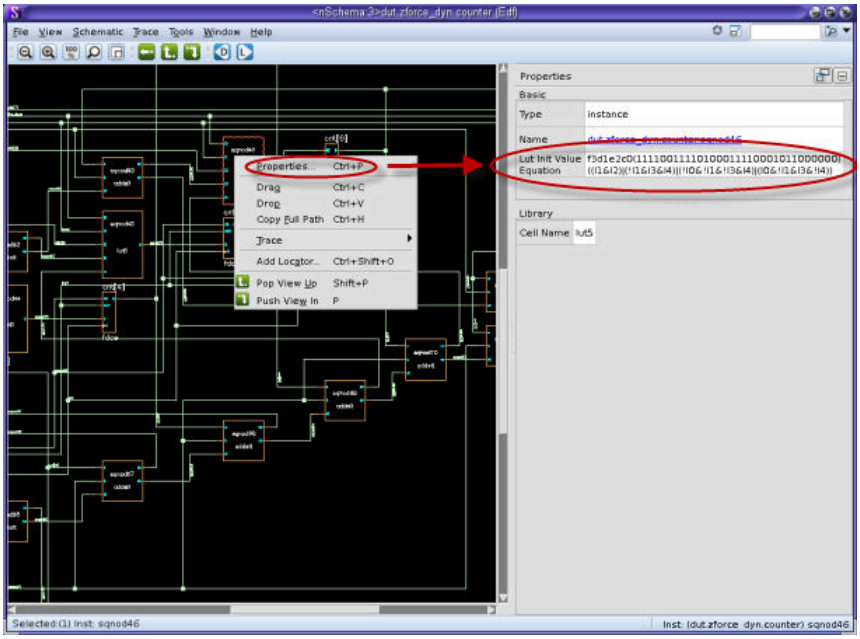
Feedback

Schematic Menu Options

The following options are available in the **Schematic** menu in the **nSchema** window:

- **Properties**: Invokes the dialog box that displays the basic information (instance/net name) of the selected object.

For example, for the "lut" property data, "Lut Init Value" and the corresponding "Equation" is displayed in the **Properties** dialog box. The "Lut Value" includes hexadecimal and binary as illustrated in the following figure:



- **Find in Current Scope**: Invokes the dialog box that supports search functions for the current window's netlist object including net, port, instance port, instance, and module.
- **Find locator**: After you add the locator using **Add locator** right-click menu option, you can use this option to search the added locators.
- **Auto Fit Found Object(s)**: **nSchema** zooms in the object when user drops an object to this window.
- **Selection**: **nSchema** selects all netlist objects including instance/net/port.

Note:

All the menu options present in Verdi nSchema window are not supported for ZNL netlist display.

Feedback

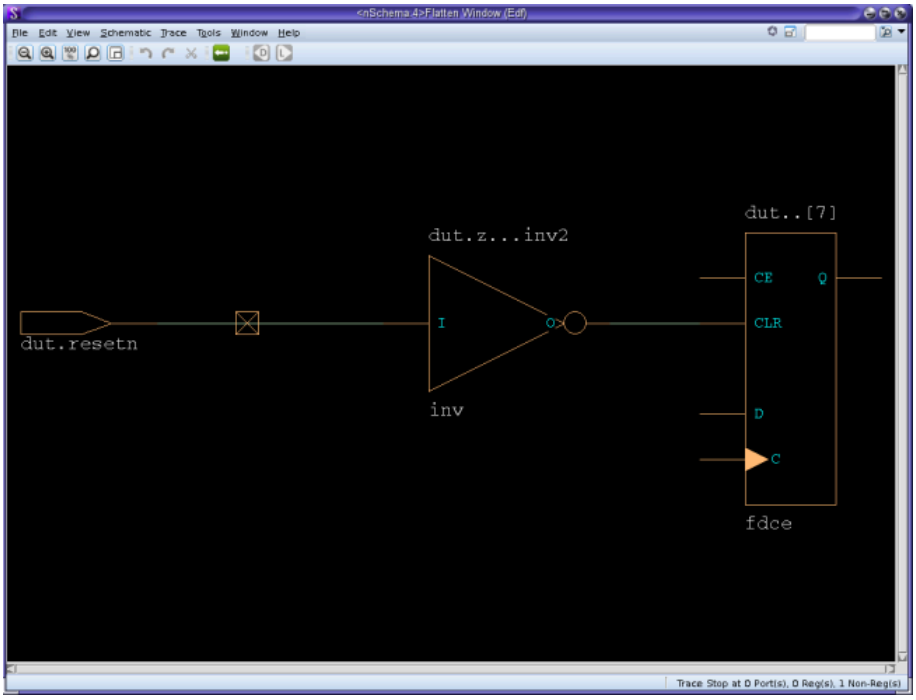
Flatten Schematic Window

To open the **ZNL Flatten Schematic** window for one signal, click the **New Flatten Schematic** right-click (RMB) option in the **Netlist Signal List** pane in **nTrace**.

To open **ZNL Flatten Schematic** window from an already open ZNL schematic window, click **Tools** > **New Schematic** > **Driver/Load/Connectivity**.

The following figure illustrates the **Flatten Schematic** window for ZNL:

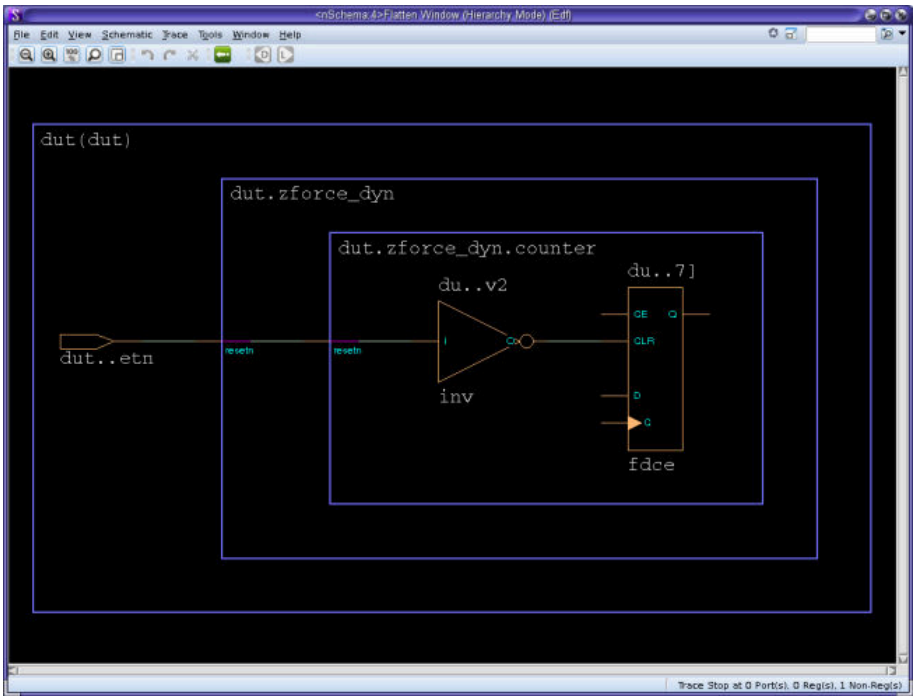
Figure 22 Flatten Schematic Window



Feedback

The following figure illustrates the **Hierarchy Flatten Schematic** window for ZNL:

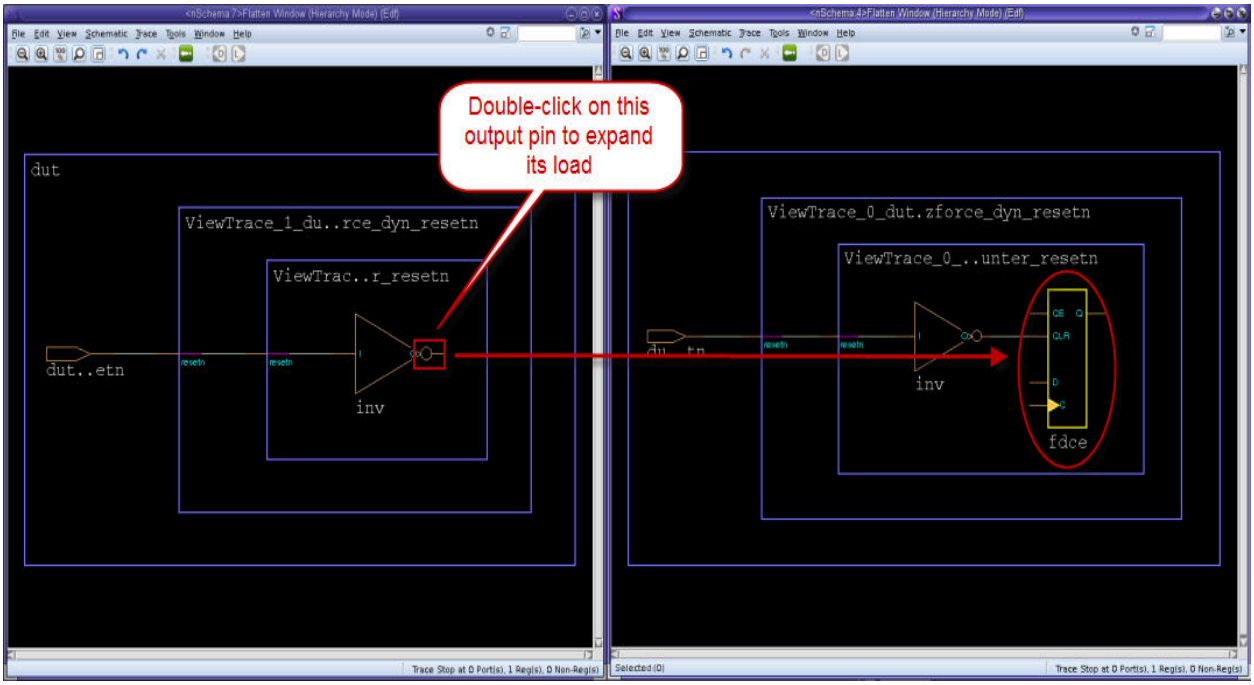
Figure 23 Hierarchy Flatten Schematic Window



Feedback

To expand the driver or load of a pin, double-click the pin in the **Flatten/Hierarchy Flatten** window, as illustrated in the following figure:

Figure 24 Expanding the Driver or Load of a Pin



Limitations

The following limitations apply for the ZNL netlist schematic window:

- Save/restore session function is not supported.
- **nECO** window is not available for the ZNL netlist.
- Clock domain/tree extraction is not available for the ZNL netlist.

Supported Tcl Commands

To create a new schematic from the specified ZNL netlist, use the `-edf` argument with the `schCreateWindow` Tcl command.

Syntax

```
-edf edf_net_list_name
```

Feedback

Examples

- To create a full ZNL schematic, use the following command:

```
schCreateWindow -win $_nSchem a1 -edf "EDFNetlist_0" -scope "dut"
```

- To create an incremental schematic about connectivity of one signal, use the following command:

```
schCreateWindow -win $_nSchema1 -edf EDFNetlist_0 -conn -signal
 "dut.clk"
```

- To create an incremental schematic about driver of one signal, use the following command:

```
schCreateWindow -win $_nSchema1 -edf EDFNetlist_0 -driver -signal
 "dut.clk"
```

- To create an incremental schematic about load of one signal, use the following command:

```
schCreateWindow -win $_nSchema1 -edf EDFNetlist_0 -load -signal
 "dut.clk"
```

7

Complex Event Language Support in nTrace

In ZeBu debug flow, to identify the internal state of the emulator, monitor program (state-machine) and a specific event or a sequence of events must be identified.

To meet this requirement, Complex Event Language (CEL) is introduced in ZeBu debug flow.

This section discusses the following subtopics.

Overview of CEL

Complex Event Language (CEL) is introduced to describe signals and conditions for monitoring a program (state machine) and for tracing a specific event or a sequence of events. The CEL helps you to describe the signals in an easier way than using the normal

hardware description languages such as Verilog, SystemVerilog, and VHDL. Therefore, the debug turnaround time is reduced and the debug efficiency is improved.

CEL is described using the context-free syntax, and a simple variant of Backus Naur Form (BNF), in particular:

- Lower-case words in roman form (may contain underlines) to denote syntactic categories.

  Example: `simple_expression`

- Boldface words to denote reserved words.

  Example: `module`

  Reserved words must be used only in those places indicated by the syntax.

- A production consists of a *left side*and a *right side*. The symbol *"::="* is read as "can be replaced by". The *left side* of a production is always a syntactic category and the *right side* is a replacement rule.

- A vertical bar to separate alternative items on the *right side* of a production unless it occurs immediately after an opening brace, in which case it stands for itself:

  ```
  action_statement ::= simple_action | if_then_else_statement
  action_statement_list ::= action_statement { ';' action_statement}
  ```

  In the first instance, an occurrence of "`action_statement`" can be replaced by "`simple_action`" or "`if_then_else_statement`". In the second case, the "`action_statement_list`" can be replaced by a list of "`action_statement`" separated by semicolon '*;*' (For the meaning of braces, see the next bullet).

- Square brackets to enclose optional items on the *right side* of a production; therefore, the two following productions are equivalent:

  ```
  if_then_else_statement ::= if_statement [ else_statement ]
  if_then_else_statement ::= if_statement | if_statement else_statement
  ```

- Curly braces to enclose a separated item or items on the *right side* of a production. The repetitions of an item occur from *left* to *right* as with an equivalent left-recursive rule. Therefore, the following two productions are equivalent:

  ```
  simple_action_list ::= simple_action { ';' simple_action }
  simple_action_list ::= simple_action | simple_action_list ';'
                         simple_action
  ```

  The term `simple_name` is used for any occurrence of an identifier that already denotes a declared entry.

For many aspects, CEL refers to *Verilog HDL Language Reference Manual*. For more information, see *Verilog HDL Language Reference Manual*.

Module Entity

The module entity is the primary monitor triggering description in CEL. It represents a Finite State Machine (FSM) that has well-defined inputs and single output (trigger signal), and a well-defined function and state-transitions. In the module of CEL, all the input signals are associated with a real signal in a design, which is described by HDL (Verilog, SystemVerilog or VHDL).

The FSM in the module

- fetches the input value from the associated signals in the HDL,
- performs the state-transition based on the defined state transition behavior in CEL, and
- triggers the output when certain state and condition are reached.

Module Declaration

Module declaration defines an interface between a given HDL design and the behavior of the FSM monitor.

Syntax

```
complex_event_entry ::= module_definition interface_definition
 FSM_definition_list
      endmodule
module_definition ::= module module_simple_name ';'
```

Interface Definition

The interface definition declares items that are common to all modules whose interface is defined by the given module definition.

Syntax

```
interface_definition ::= [ clock_definition ] [ reset_definition ]
 { interface_declarative_item }
interface_declarative_item ::=
   hdl_variable_definition
 | local_variable_definition
 | counter_definition
 | event_definition
```

Example

The following is an example for declaring a module.

Feedback

```
module toggle_count ;
   clock posedge top.clock ;
   reset posedge top.reset ;
   ….
endmodule
```

Clock and Reset Definitions

A clock/reset definition declares the clocking/resetting mechanism for the FSM of a specific-module.

Note:

Both the clock and reset definitions are optional.

When the clock definition appears, it means that the FSM is synchronous and all state transition happens on the clock edge. An asynchronous FSM can be described in CEL with no clock definition. This means that all the state transition condition are checked and computed when the input is changed.

The state value is set to the initial state when the reset condition occurs. When the reset condition occurs, CEL also clears (set the value to zero (0)) the local variables and the counters. If there is no reset definition in CEL, the value of the state variable is set to the initial state at the beginning of the simulation, and it can never be reset.

When the reset definition appears, the reset behavior is asynchronous. The initial state value is set to the state signal and the local variables and counters are set to zero (0) on the active reset edge.

Syntax

```
clock_definition ::= clock sampling_type hdl_hierarchy_name_clause ';'
reset_definition ::= reset sampling_type hdl_hierarchy_name_clause ';'
sampling_type ::= posedge | negedge | both
```

Where,

- `hdl_hierarchy_name` is the full path name to identify which signal in the HDL design is used for the `clock` or `reset` for the FSM defined in this module.

For both `clock` and `reset`, `posedge` (rising edge of the signal) or `negedge` (falling edge of the signal) of the active signal can be selected for the FSM behavior. The keyword, "both", means that the sampling is done on both the edges.

Feedback

Variable Definitions

CEL supports the following variable definitions:

- HDL variable definition: It defines the input variable, which associates to an `hdl_hierarchy_name` signal. The input variable is treated as an alias signal to represent the same signal from the `hdl_hierarchy_name`.
- Local variable definition: It declares a local variable, which can be used in the module of CEL and it does not associate to any signal in the HDL design that is being monitored by CEL.

Syntax

```
hdl_variable_definition ::= var var_simple_name
                            { '[' decimal_number ':' decimal_number ']' }
                 hdl_hierarchy_name_clause  ';'
local_variable_definition ::= var var_simple_name
                             { '[' decimal_number ':' decimal_number
 ']' }';'
hierarchy_name_clause ::= hierarchy_name | '"' hierarchy_name '"'
```

Counter and Event Definitions

The counter definition is identical to the local variable definition except that the counter variable is an unsigned 64-bit integer. In addition, only one counter variable can be associated with inc (increment), dec (decrement), and reset (set value to zero) command to control the counter value. The counter size is 64-bit.

The event definition defines an event variable and associates with an event expression, which is constructed by the HDL, local, and counter and other event variables. The value changes on the variables in the expression triggers the computation of the expression, and the event variable value is updated immediately.

The size (bit length) of all event variables is one bit. The size (bit length) of the event expression must also be one bit.

Note:

It is illegal if the size of event expression is not one bit.

Only one definition for each variable is allowed for all types of variables, such as HDL variable, local variable, counter variable, and event variable.

Note:

It is for a same variable defines more than once.

Syntax

```
counter_definition ::= counter counter_simple_name ';'
event_definition ::= event event_simple_name ':= ' event_expression ';'
```

FSM Definition

The FSM definition list in a module can define one or more FSM definitions. Each FSM definition describes a FSM behavior to monitor the given HDL design. The different FSM definitions are executed in parallel.

Note:

Each FSM has only one copy (one state variable, and one state label for current state) at all time.

It is illegal for a FSM name has more than one FSM definitions.

The BNF for FSM definition list is displayed as follows:

```
FSM_definition_list ::= FSM_definition { FSM_definition }
FSM_definition ::=
                fsm fsm_simple_name '{' initial_statement
 state_statement_list '}'
        | notify_event_definition
initial_statement ::= initial state_label ';'
label ::= simple_name
notify_event_definition ::= notify event_simple_name ';'
```

The name of FSM is defined in the FSM definition. The initial statement describes the initial state label for the FSM. The initial value is assigned to the state variable when the simulation/emulation start or when the reset condition of the reset definition is set to "`true`."

For a single state FSM, to perform the combinational assertion behavior, CEL also supports notify event definition. You can use this simpler construct to toggle the notify signal for a combinational toggling condition.

State Statement List

The state statement list defines the behavior of an FSM. The state statement list contains one or more state statements. Each state statement defines a state label and state actions (action statement lists) associated with a state in the FSM. The state label is a simple name, which describes the state name for a state transition in the FSM. The state labels in different state statements of the same FSM definition cannot be duplicated. All the action statements of a state (label) must be defined in state statement list after the state label. The action statements in the same state label is be executed sequentially.

Feedback

```
state_statement_list ::= state_statement { state statement }
state_statement ::= state state_label '{' action_statement_list '}'
action_statement_list ::= action_statement { ';' action_statement}
action_statement ::= simple_action | if_then_else_statement
label ::= verilog_identifier
```

Where,

- The `action_statements_list` contain two type of actions as follows:
  - A simple action performs the behavior without any condition.
  - The if-then-else statement performs the execution branch-based on the given condition.
- A `label` is a name of an identifier, which specifies the state name in the FSM definition.

Simple Action

Six types of statements are present for the simple action, as follows:

- `goto`
- `reset`
- `inc`
- `dec`
- `notify`
- `$display`

Note:

The reset, inc, and dec statements are the simple action statements for the counter.

Syntax

```
simple_action ::= goto state_label
                | reset counter_simple_name
                | inc counter_simple_name
                | dec counter_simple_name
                | local_variable_simple_name ':=' expression
                | notify
                | $display '(' string  [ argument_list ] ')'
argument_list ::= ',' expression { ',' expression }
```

Feedback

The following table explains about each action:

Action	Description
`goto`	Defines the state transition and assigns the given state label to the state variable on the latest event at the current simulation time. This behavior assures that an action statement in a next state is not executed until the next sampling time and no race condition occurs.**Note:** *It is not recommended that the state label of the goto statement is not defined in the state statement of the same FSM definition. In addition, in one sampling time, only one state assignment (transition) statement can be executed in the CEL.*
`reset`	Sets the counter variable to zero (0).
`inc`	Increases the value of the counter variable by one (1).
`dec`	Decreases the value of the counter variable by one (1).
`notify`	Triggers an assertion event to the notify signal of the FSM. In CEL, each FSM definition has an implicated notify signal.
`$display`	Preforms a similar syntax and behavior as `$display` system task in Verilog HDL language. The message in `display` statement outputs to the console when the CEL is running.

The local variable assignment statement is basic mechanism for placing the result value from expression to the local variable.

The local variable and counter have only one driver so that the local variable on the LHS in a single FSM only. It is not recommended to use the local variable in more than on FSM.

$display Command Options

In a CEL file, the `$display` command supports the following options:

- `* %lu`: Use if the value of width is 64 or less;
- `* %s`: Use if the value of width is greater than 64

Example

Consider that a CEL file has the following variables:

```
var rt_clk_reg TOP.rt_clk_reg; //width = 1
var write_reg_32[31:0] TOP.write_reg_32[31:0]; //width = 32
var write_reg_64[63:0] TOP.write_reg_64[63:0]; //width = 64
var write_reg_128[127:0] TOP.write_reg_128[127:0]; //width = 128
```

To display these variables properly, use the $display command as follows:

```
$display("rt_clk_reg = %lu", rt_clk_reg);
$display("write_reg_32 = %lu", write_reg_32);
$display("write_reg_64 = %lu", write_reg_64);
$display("write_reg_128 = %s", write_reg_128);
```

If-Then-Else Statement

An if-then-else statement executes one or none of the enclosed sequences of statements (action list) based on the value of one or more corresponding conditions. If the condition clause evaluates to one (1'b1), then the sequence of the statements (action list) in the "then" part is executed. Otherwise, it executes the "else" part of statements (action list) if else-action-list appeared.

Syntax

```
if_then_else_statement ::= if_statement [ else_statement ]
if_statement ::= if '(' condition_clause ')' then simple_action_list
else_statement ::= else else_action_list
condition_clause ::= event_var_simple_name
 [binding_method constant_value]
binding_method ::= occurs  |  sustains
simple_action_list ::= simple_action
                     | '{' simple_action { ';' simple_action }
 '}'else_action_list ::= simple_action_list | if_then_else_statement
```

Note:

The if-then-else-statement can be nested, but the CEL only allows it in the else action list. You can use the nested if-then-else-statement to construct the "case" (in Verilog) or "switch" (in C) like behavior in CEL.

An event variable in the condition clause is evaluated (executed) on the sampling time. The sampling time of the condition clause is synchronous when the clock definition is defined, and it can also be asynchronous when the clock definition is not defined in the CEL.

In the synchronous sampling, the event expression of the event variable is evaluated on every cycle when the clock definition is occurred. In the asynchronous sampling, the event expression of the event variable is evaluated whenever any variable in the event expression had value change (level sensitive).

Methods Available in an If Statement

In a condition clause in the if statement, there are two optional binding methods that can be described in the condition.

Feedback

When the condition clause has no binding method, the value and value-change time of the condition clause is identical to the event variable in the condition.

When the binding method in the condition clause, extra rules must be satisfied before the condition clause to be evaluated to `1`(`true`). A constant value after the binding method is a positive integer and it describes how many times of the event variable must be evaluated to `1` to fill the extra rules of the condition clause.

The following table explains the binding methods.

Method	Description
`occurs`	The number of times for the event variable that is evaluated to `1` (`true`) at different sampling times is equal to the constant value. Or The number of times for the event variable returns `0` (`false`) at the sampling time when the event variable is evaluated to non-1 (0, x or z).
`sustains`	The integer constant value has two meanings. In the synchronous mode, the clock definition is declared in the CEL and this constant refers to the number of sampling time. In the asynchronous mode, no clock definition is declared in the CEL. That is, the real delay is declared with the time unit, which is declared by the design, dumped waveform file or specified by the user in the tool.The `sustains` clause counts the number or delay of the event variable that is evaluated to `1` at different sampling times until the number or delay is equal or larger to the constant value. However, the `sustains` clause requests the event variable to hold its value to `1` at all the time in the period of the delay. It only checks the event value at the sampling time. Any non-1 value for the event variable results the condition expression to be evaluated to `0` and resets (clear) the delay counter to `0`.

The *If-Then-Else* statement has an implicit counter in the CEL evaluation engine to count the event or delay for each binding method of these two clauses. The counter to count the number of events is "`occurs`" and the delay for "`sustains`" in the condition clause resets (clear to `0`) at the beginning of the simulation time and when reset definition is executed. In addition, the counter is set to `0` when the value of the state variable is changed.

For more information, see Examples of CEL Waveform.

For example, in the following case, `statement1` is executed when event variable `ev1` is `1` and occurs three times first, or the `statement2` is executed if the `ev2` is `1` and occurs two times first.

Example

```
if (ev1 occurs 3) then statement1
else if (ev2 occurs 2) then statement2
```

Feedback

Expression

The syntax and rules of CEL expression are same as the Verilog HDL except that the CEL only supports a subset of the operations. All the behavior of the supported operators are identical to the definition of the Verilog HDL language.

For example, "~" is bit-wise negation and "!" is logical negation. The precedence order of the binary operators and the conditional operator (?:) follows the precedence order defined by the Verilog HDL.

The CEL follows the rules of the Verilog HDL to support the expression execution order and the expression bit lengths.

Similar to Verilog HDL, the CEL supports the bit-select and part-select for a variable, and follows the same syntax as Verilog HDL. A square bracket with one index number is a bit-select for a variable, and can select one bit of the variable. A square bracket with two index numbers and with colon symbol (':') in between is a part-select for a variable and can select one or more than one bit of the variable.

Syntax

```
expression ::= variable_expression
             | value
             | unary_expression
             | binary_expression
             | conditional_expression
             | '(' expression ')'
             | $rose '(' hdl_variable_expression ')'
             | $fell '(' hdl_variable_expression ')'
             | $stable '(' hdl_variable_expression ')'
variable_expression ::=
                    simple_name | simple_name '[' decimal_number ']'
                     | simple_name '[' decimal_number ':' decimal_number
 ']'
unary_expression ::= unary_operator expression

unary_operator ::= '~' | '!'
binary_expression ::= expression binary_operator expression
binary_operator ::= '+'
                  | '-'
                  | '<<'
                  | '>>'
                  | '<'
                  | '>'
                  | '<='
                  | '>='
                  | '=='
                  | '!='
                  | '&'
```

```
                    | '|'
                    | '^'
                    | '~^'
                    | '&&'
                    | '||'
conditional_expression ::= expression '?' expression ':' expression
```

The following table lists the functions supported in the expression and its description.

Function	Description
System Functions	
`$rose()`	Accepts only one bit variable expression. **Note:** If the length of any variable expression is more than one-bit, it is considered as illegal. The function output of `$rose()` is defined the same as the `posedge` operator behavior of Verilog HDL. The function output of `$fell()` is defined the same as the "`negedge`" operator behavior of Verilog HDL.
`$fell()`	
`$stable()`	Accepts variable more than one bit variable expression. The function output of `$stable()` is set to true when all the bits in the variable expression have no change (no edge, or the bit is evaluated false on both `$rose()` and `$fell()` functions).

Note:

In CEL, only hdl variable expression is accepted in all three system functions. It is illegal to place a local variable, a counter or an event variable in $rose(), $fell() and $stable() system functions.

The following table lists the operators supported in the expression.

Unary Operators	• “~” for bit-wise negation • *“!” for logical negation*
Binary Operators	• “+” for addition • “-“for subtraction • “<<” for left shift • “>>” for right shift • “<” for less than • “>” for greater than • “<=” for less than or equal to • “>=” for greater than or equal to

Feedback

- “==” for equal to
- “!=” for not equal to
- “&” for bit-wise and
- “|” for bit-wise or
- “^” for bit-wise exclusive or
- “~^” for bit-wise exclusive or
- “&&” for logical and
- “||” for logical or

Guidelines for Defining Names in CEL

This section describes the guidelines for various forms of names.

Three types of names are used in the CEL as follows:

- Simple name:

  A simple name for a named entity is the identifier associated with the entity using its definition (for example: module, FSM, counter, local variable definition), or another hierarchy signal associated with the entity using an alias definition (for example: variable definition).

- Hierarchy name:

  The hierarchy name is used to describe a full path name of a signal/variable object in the HDL (Verilog, system-verilog or VHDL) design. In pure Verilog design, the hierarchy name has the same definition as the full path name in Verilog LRM. In Mixed Verilog with VHDL or pure VHDL design, a special '@' character is added to handle the escaped name of VHDL.

  For example, in hierarchy name "`top.\verilog.x .@\vhdl.y\.sig`", the name/identifier of each scope or signal of this hierarchy name is:

  - `"top"`,
  - `"verilog.x"`,
  - `"\vhdl.y\"`, and
  - `"sig"`

    Where,

  - `"top" and "sig"` are the normal name (for Verilog or VHDL),
  - `"\verilog.x"` is an escaped name of Verilog
  - `"\vhdl.y\"` is an escaped name of VHDL

Feedback

- Extended name

  Note:

  Do NOT use a CEL keyword in the extended_name.

Syntax

```
simple_name ::= verilog_identifier
hierarchy_name ::= hierarchy_name '.' extended_name | simple_name
extended_name ::= variable_expression | verilog_escaped_identifier | '@'
 vhdl_escaped_identifier
```

The identifier of CEL follows the same lexical rules of Verilog HDL. An identifier is used to give a unique name to an object so that the object can be referenced. An identifier is any sequence of letters, digits, dollar sign ($), and underscore characters (_), and it is case-sensitive. However, the first character of an identifier must not be digit or dollar sign ($).

For the Verilog escaped identifier, it also follows the rules from Verilog HDL. It starts with the backslash character (\) and ends with a whitespace (whitespace can be a space, tab, newline and form feeds).

For the VHDL escaped name, it follows the rules of extended identifier from VHDL. It starts and end with a backslash characters (\), and any graphic characters are allowed. If a backslash must be used as one of the graphic character of an extended literal, it must be doubled. This is used to identify the object with an extended name in a VHDL design.

For the scope name in the hierarchy name, it cannot be a partial name (example: ABC[3:0]).

Note:

Do NOT have a partial name in the hierarchy name except the latest place, which represents a part of the bus signal.

Lexical Conventions

The lexical tokens follows the lexical rules of Verilog HDL for the following:

- **Whitespace**: Contains the characters for spaces, tabs, newlines and formfeeds
- **Comment**:
  - Starts with two characters '//' and ends with a newline, if it is an one line comment.
  - Starts with '/*' and ends with '*/', if it is a block comment.

Feedback

Note:

It cannot be nested.

- **Operator**: Single- or double-character sequences and used in expressions.

  For more information, see *Expression*.

- **String**: A sequence of characters enclosed by double quotation marks (") and contained on a single line.

- **Identifier**: See *Guidelines for Defining Names in CEL*.

- **Number (value)**:

```
value ::= decimal_number | binary_number | octal_number | hex_number
decimal_number ::= [ sign ] unsigned_number
                 | [ sign ] decimal_base unsigned_number
sign ::= '+' | '-'
decimal_base ::= ''d' | ''D'
unsigned_number ::= decimal_digit { '_' | decimal_digit }
decimal_digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8'
 | '9'
binary_number ::= [ size ] binary_base binary_digit { '_' |
 binary_digit }
binary_base ::= ''b' | ''B'
binary_digit ::= 'x' | 'X' | 'z' | 'Z' | '0' | '1'
octal_number ::= [ size ] octal_base octal_number { '_' |
 octal_number }
octal_base ::= ''o' | ''O'
octal_digit ::= 'x' | 'X' | 'z' | 'Z' | '0' | '1' | '2' | '3' | '4' |
 '5'
             | '6' | '7'
hex_number ::= [ size ] hex_base hex_digit { '_' | hex_digit }
hex_base ::= ''h' | ''H'
hex_digit ::= 'x' | 'X' | 'z' | 'Z' | '0' | '1' | '2' | '3' | '4' |
 '5'
           | '6' | '7'  | '8' | '9' | 'a' | 'A' | 'b' | 'B' | 'c' |
 'C'
           | 'd' | 'D' | 'e' | 'E' | 'f' | 'F'
size ::= unsigned_number
```

Variable Dumping

To support debug functionalities, all the CEL variables, such as, state, event, counter, and local, can dump the value to a waveform file.

Feedback

The naming rules for all CEL internal variables are follows:

- All the variables in CEL start with a special name "\$$CEL " to distinguish the CEL signal with the HDL signals.
- The second place in the hierarchy name of the variable dumping is the module name (the name from the module definition).
- CEL variable dumping only contains the local state variables and local counters, which are defined in the CEL module. The CEL variable dumping do not include the hierarchy name in the variable definition, which represents an alias name in the CEL from a HDL design. The waveform of hierarchy name in the variable definitions is fetched from the waveform of HDL design.
- For the counters, events, and local variables defined in a CEL module, the name is in the third place of the hierarchy name.

  There are three types of implicit variables in the FSM. They are:

  - State variable
  - State notifies variable
  - Implicit counter for each of binding method in condition clause

  All hierarchy name for the variables in the FSM has the name of FSM in the third place of the hierarchy name. A fixed name (lowercase) "`state`" is represented for all the FSM in the CEL. The complete state name is the name of FSM in the third place with the fixed state name "`state`" in the fourth place. These rules are applicable for notifies variable with the variable name in lowercase "`notify`".

  The implicit counter for binding method ("`occurs`" and "`sustains`" clauses) are named as "`_bm##`", where `##` is an integer number, which represents the order number of binding method in this FSM source code.

  The counter name can also be placed at the fourth place of the hierarchy name after the FSM name.

Example

```
module toggle_mod;
clock posedge top.clk;    // dump variable \$$CEL.toggle_mod.clk,
                          // no waveform for top.clock in HDL
var cpu1_status [3:0] top.cpu1.status;
                           // dump variable \$$CEL.toggle_mod.cpu1_status
                          // no waveform for top.cpu1.status in HDL
var cpu2_status [3:0] top.cpu2.status;
                          // dump variable \$$CEL.toggle_mod.cpu2_status
                          // no waveform for top.cpu2.status in HDL
var tmp [3:0];            // dump local variable \$$CEL.toggle_mod.tmp
```

Feedback

```
counter count;           // dump counter variable \$$CEL.toggle_mod.count
event ev1:= cpu1_status == 4'b0;
                         // dump event variable \$$CEL.toggle_mod.ev1
event ev2:= cpu2_status == 4'b0;
                         // dump event variable \$$CEL.toggle_mod.ev2
fsm cpu_fsm {      // dump state variable \$$CEL.toggle_mod.cpu_fsm.state
initial S0;
state S0 {
  if (ev1 sustains 3) then
                 // dump implicit variable
 \$$CEL .toggle_mod.cpu_fsm._bm1
  else if (ev2 occurs 3) then
                // dump implicit variable \$$CEL .toggle_mod.cpu_fsm._bm2
}
state S1 {
   if (ev1 sustains 2) then
               // dump implicit variable \$$CEL.toggle_mod.cpu_fsm._bm3
…
state Sn { notify }
              // dump implicit variable \$$CEL.toggle_mod.cpu_fsm.notify
}
endmodule
```

Data Types for CEL Variables

The data type for FSM state variable is Verilog enum type.

The data type for implicit and explicit counter variables is 64-bits Verilog bus (wire/reg) type.

The data type for all other variables is normal Verilog scale or bus (wire/reg) type.

Examples of CEL Waveform

An example to generate a CEL waveform for a module is follows:

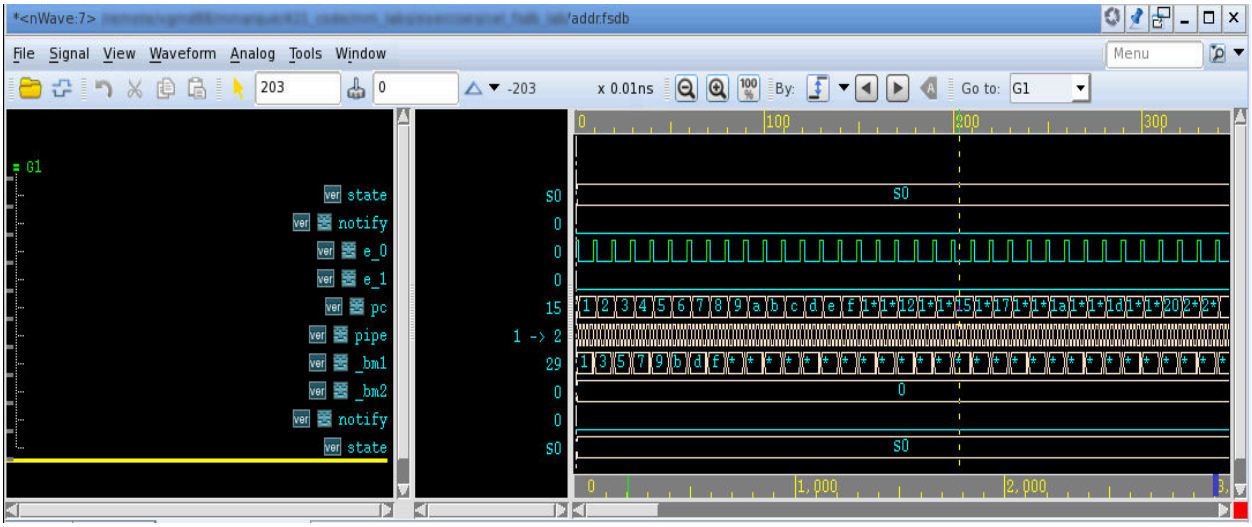
```
module toggle_ABC ;
   clock posedge top.clock ;
   var A[3:0] top.A ;
   var B[3:0] top.B ;
   var C[3:0] top.C ;
   event x:= A == 4'b1 ;
   event y:= B == 4'b1 ;
   event z:= C == 4'b1 ;
   fsm x_y_z {
      initial S0 ;
      state S0 { if (x) then goto S1 }  // wait for event x
      state S1 { if (y) then goto S2 }  // wait for event y
      state S2 { if (z) then goto S3 }  // wait for event z
       state S3 { goto S4 }             // wait to next clock edge
```

Feedback

```
        state S4 { notify }              // trigger the notify signal
    }
    endmodule
```

The following figure displays the CEL waveform.

Figure 25 CEL Waveform of a Module



CEL Waveform for *an*Occurs Clause

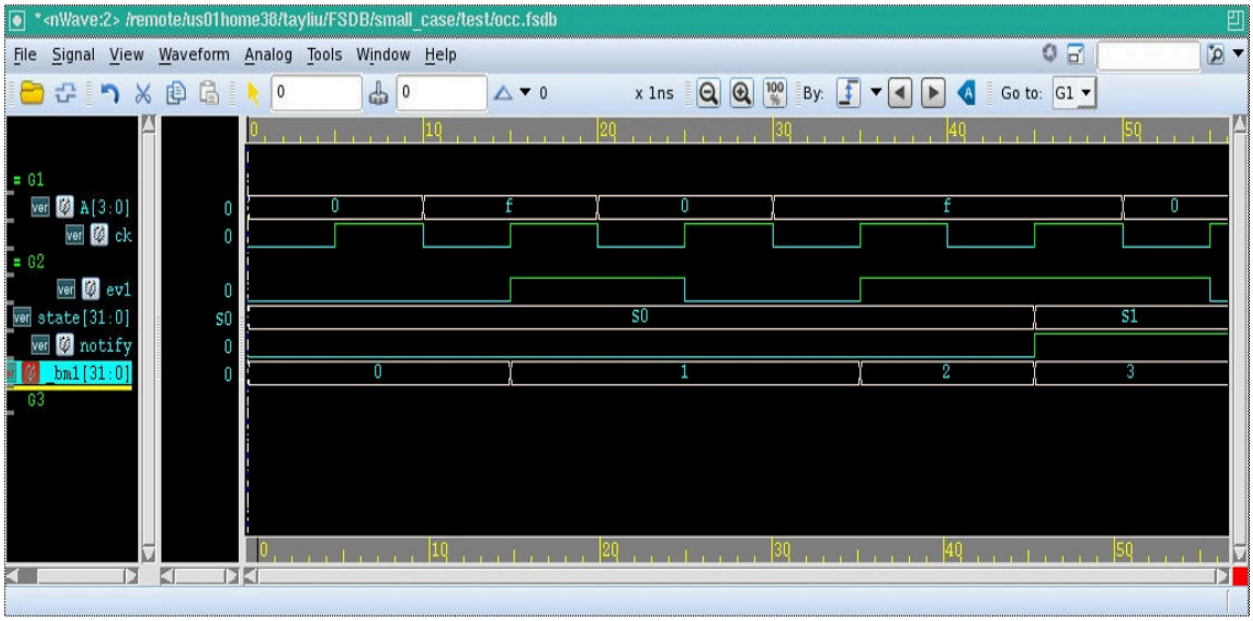
An example to generate a CEL waveform for an Occurs Clause is follows:

```
module ex_occurs ;
clock posedge top.clock ;
var A[3:0] top.A ;
event ev1:= A == 4'b1111 ;
fsm occ_1 {
        initial S0 ;
        state S0 { if (ev1 occurs 3) then goto S1 }
                                     // wait for event ev1 3 sampling times
        state S1 { notify }          // trigger the notify signal
}
endmodule
```

The following figure displays the CEL waveform for an Occurs Clause.

Feedback

Figure 26 CEL Waveform of an Occurs Clause



CEL Waveform for a Sustains Clause

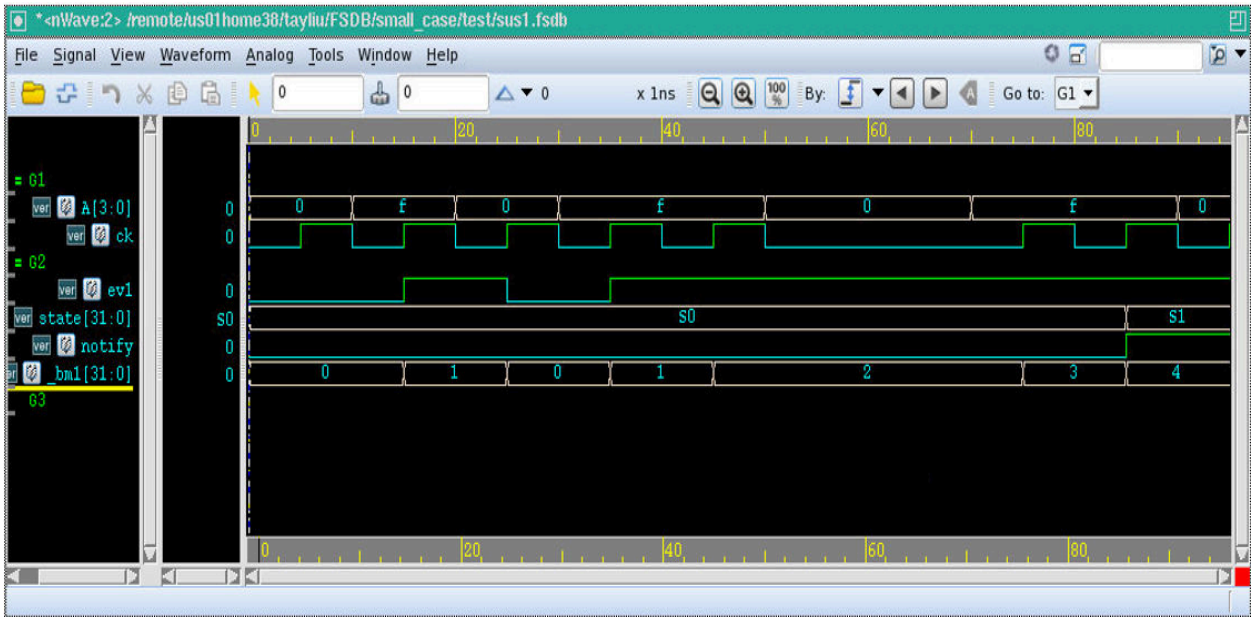
An example to generate a CEL waveform for a Sustains Clause is follows:

```
module ex_sustain ;
clock posedge top.clock ;
var A[3:0] top.A ;
event ev1:= A == 4'b1111 ;
fsm x_y_z {
        initial S0 ;
        state S0 { if (ev1 sustains 4) then goto S1 }
                                // wait for event ev1 sustains 4 sampling
 time
        state S1 { notify }  // trigger the notify signal
}
endmodule
```

The following figure displays the CEL waveform for a Sustains Clause.

Feedback

Figure 27 CEL Waveform of a Sustains Clause



Limitations

The limitations of CEL are listed as follows:

- Current CEL does not support multiple dimension array in the hierarchy name of the global variable.
- VHDL generate-block name is required "(" and ")" in the name, which is not supported by LRM.

The following features are not supported with ZeBu CEL.

- Clock Expressions: `clock_definition ::= clock sampling_type hdl_hierarchy_name_clause ';'`
- Only one FSM is supported for ZeBu CEL.
- Combinational Signals are not supported for CEL notifications.
- Binary operators, XOR and XNOR (^ and ~^), are not supported.
- Nested if-else statements unsupported.
- Concatenation is not supported in event expressions.

For example:

```
event e_0:=  ({count_up[7:0],count_down[7:0]} == 16'h3232);
```

- If you add more than 1k signals in the CEL file, the emulation performance degrades.

Displaying the CEL File in nTrace

In **nTrace**, the CEL file is displayed in the **Complex Event** window, which is introduced in **Tools** > **Emulation** > **Complex Event** to import the CEL files.

Alternatively, you can load the CEL file using the command line option, `-cel_file`, as follows:

```
novas -emulation -ultra -lca -cel_file top.cel
```

Note:

Importing multiple files at a time is not supported.

The following table provides the `Complex Event` menu command.

Note:

Only one CEL file can be displayed and debugged on the CEL source code debug window at a time. When you try to open a second CEL file, the previously/currently opened CEL file is closed. Subsequently, Verdi closes the primary (FSDB, SLVF or ZTDB) waveform file and then nWave window(s) reopens it with the same waveform file.

Table 3 Complex Event Menu

Menu	Description
File	
Open File	Allows you to select and specify a CEL file, and then imports the selected CEL.
Close	Closes the CEL window.
Source	
Active Annotation (X)	Shows/hides active annotation for HDL signals.
Tools	
Customize Menu/ToolBar	Invokes Customize Menu/Toolbar forms.

Feedback

Table 3 Complex Event Menu (Continued)

Menu	Description
Window	
Dock to -> New Container Window	Opens the Complex Event window in a new window.

Functions for Complex Event

The following basic functions are supported for complex event in the Complex Event window:

- Syntax color.
- Searching next/previous string pattern.
- Case insensitive searching.
- Jumping to a specific line.
- Selection on HDL signal, event, variable, counter, and FSM objects (see Figure 28). Multiple selection is available.
- Dragging selected objects; Verdi wraps the objects' full hierarchy name and string information.
- Active annotation for HDL signal, event, variable, counter, and FSM objects.
- Postsim mode only: Users need to import HDL dumping file or CEL dumping file.
- Search next/previous transition on selection object.

Note:

Do not allow selecting a transition type in the 2017.03 release.

- Highlight (set white background color) matched state when dumping file loaded.

  Note:

  Do not support dropping objects into the CEL window in the 2017.03 release.

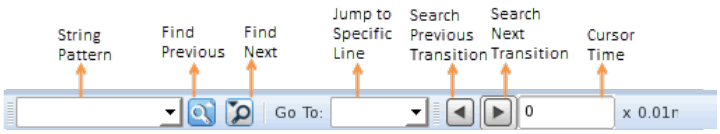
  Do not support file editing.

  Do not support file editing.

The following figure displays the toolbar present in the CEL window.

Feedback

Figure 28 CEL Tool bar



Appendix

This section discusses about the evaluation of a model using CEL.

Execution Order of a Model

The execution of a model consists of an initialization phase followed by repetitive execution of process statements in the description of a specific model. Each such repetition is called as a simulation cycle. In each cycle, the value of each variables, counters, events in the description are computed.

Note:

In CEL, the simulation cycle is defined as sampling time. If a reset definition is described in CEL, the simulation time of the reset signal with the correct edge-function is also considered as a sampling time.

At the beginning of the initialization, the current time is assumed to be the start time of a HDL waveform dumping or the beginning of the runtime for Runtime Trigger. For the *nWave* CEL post computation engine, the start time can also be the time of each waveform dump-on time (when the waveform starts the dumping from the dump-off state).

The Initialization Step

At the start time, all the local variable, implicit variable, notifies variable, and the counter are set to 0. Also, the state variable of the FSM is set to the initial state, which is specified in each FSM definition.

The Simulation Cycle

At each sampling time, CEL is executed based on a predefined order. When the clock definition is defined in CEL, the sampling time is described as the clock edge time of the edge-function. When the clock edge occurs, a sampling time for CEL also occurs.

Note:

If there is no clock definition in CEL, then the sampling time is defined by any value change of the sampling inputs HDL signals on the HDL definitions.

Evaluation of a Model

For each simulation cycle (sampling time), the CEL evaluation consists of the following steps:

1. Wait until all the events and evaluation of current simulation cycle (sampling time) to be finished in the HDL simulation.

   Note:

   In real implementation, CEL starts to compute this simulation cycle when the HDL simulation had advanced to the next simulation time, but CEL records all the evaluation on the current simulation time.

2. Obtain all the values of the HDL variables from the HDL hierarchy signals (by execution the HDL variable definitions).
3. Evaluate all the event variables based on the event definition description order in the code.

   Note:

   Each event definition is executed only once.

4. Execute the FSM definitions based on the FSM definition description order in the code.

   Note:

   In each of FSM definition, the action statements are executed only once in each simulation cycle (sampling time).

   At any execution cycle, only the action statements at the current state are executed. However, the action statements in more than one state are not evaluated.

5. Output the value change of all the CEL variables, events, states and notify.

8

Monitoring State Machine Variables in nWave

As **nTrace** supports the CEL in source code view, **nWave** must support to display the complex event in a post simulation mode when a signal is dragged and dropped or right-click menu option to add to waveform.

This section describes the following subtopics.

Bringing up the Waveform of CEL Variables

You can bring-up the CEL variables in **nWave** using one of the following methods:

- **When you have FSDB file for CEL:**In Runtime Trigger flow, you can dump-out the waveform of the CEL into an FSDB file (CEL-FSDB). First, the FSDB file of CEL waveform must be loaded to display it on the waveform. The usage is same as you debug the waveform FSDB file from the simulator.
- **When you do not have FSDB file for CEL:** In this case, you need to load the design waveform (SLVF) or use the interactive Waveform Reconstruction flow to obtain the design waveform from ZTDB. Verdi prepares the CEL variables and then you can use drag-and-drop method or right-click menu option to add CEL variables from **nTrace** window to **nWave**, or turn *on* annotation in source view. Verdi CEL postcomputation engine starts and tries to compute the CEL variables based on the signal value from the design. The computation starts from the current annotation file begin time.

Computing CEL Waveform Based on Design

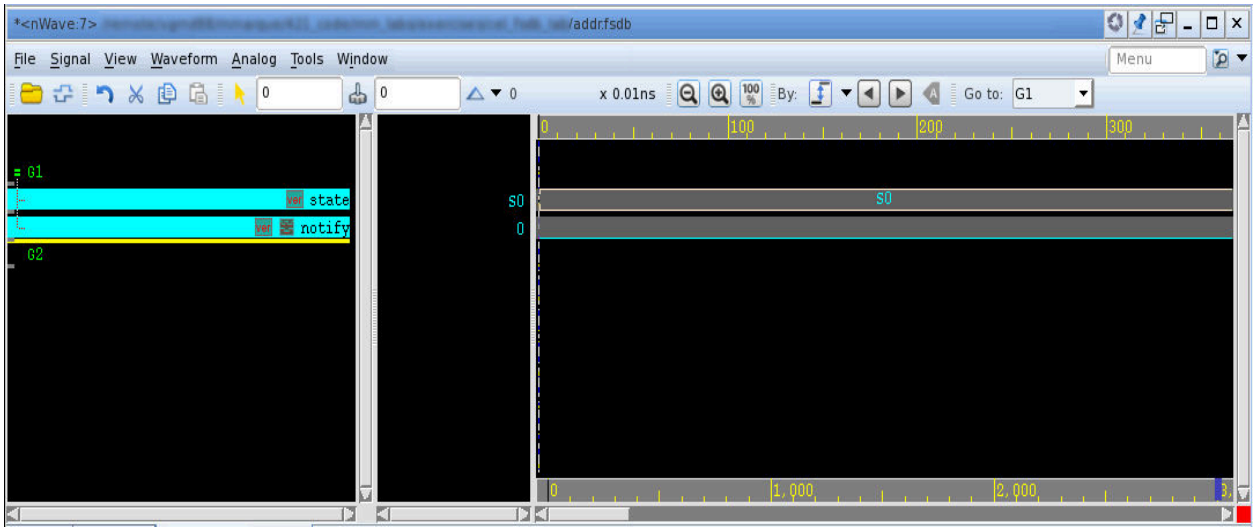
To compute the CEL waveform based on a design or HDL, perform the following steps:

1. Open the `.CEL` file in the **Complex Event** window.
2. Right-click the FSM signal in the source code view and then click **Add Object (s) to Waveform**. Or, drag-and-drop the FSM signal to the **nWave** window.

   The CEL variables in the source code view are added to the **nWave** window.

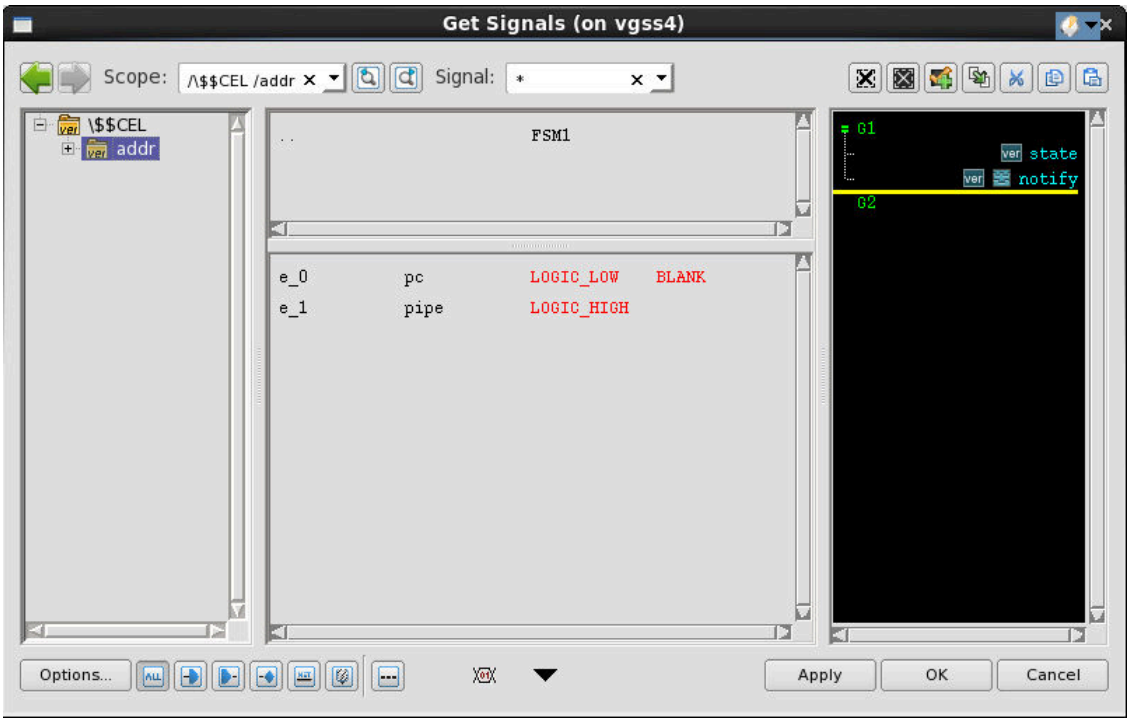
Feedback

Figure 29 CEL Variables Added to the nWave Window



3. Click **Signal** > **Get Signals**.

   The **Get Signal form** has a new scope named "`\$$CEL`", as follows:



For more information about the hierarchy and naming of CEL variables, see Complex Event Language Support in nTrace.

Feedback

Note:

If the event has unrecognized signals or syntax error, it cannot be computed and added into the nWave window.

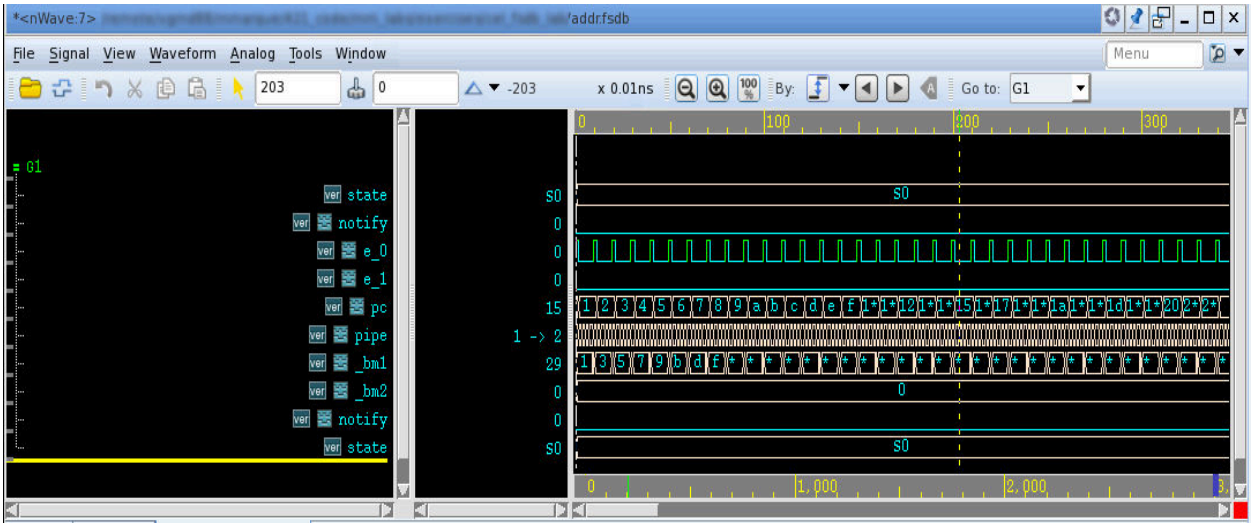
Example

The following is an example of `.CEL file` and the corresponding waveform in the **nWave** window.

```
module addr;
var pipe [4:0] TOP.pipe[4:0];
var pc [31:0] TOP.pc[31:0];
event e_0 := pipe == 5'b10000;
event e_1 := pc == 300;
fsm FSM1 {
    initial S0;
    state S0 {

        if (e_0 occurs 100) then goto S1 else goto S0
    }
    state S1 {
        if (e_1) then goto S2 else goto S1
    }
    state S2 {
        notify
    }
}
```

Figure 30 CEL Waveform Example



9

Viewing ZeBu Memory Using Verdi

Using Verdi, both HDL and EDIF memories can be viewed during **zRci** or **Simulation Acceleration** runtime. Verdi uses the memory command to get the complete memory list from **zRci** or **Simulation Acceleration** to display them in **nMemory** in a tabular view.

Note:

This information in this section is also applicable to Simulation Acceleration.

For details, see the following subsections:

Displaying Memory Content

In the simz interactive mode and the Simulation Acceleration Interactive Mode, **nMemory** can be enabled in **nTrace** and **nWave**. The following figures show the **Memory** options in **nTrace**, **nWave**, and right-click menu option from the Signal pane.

Feedback

Figure 31 Memory Menu in nTrace

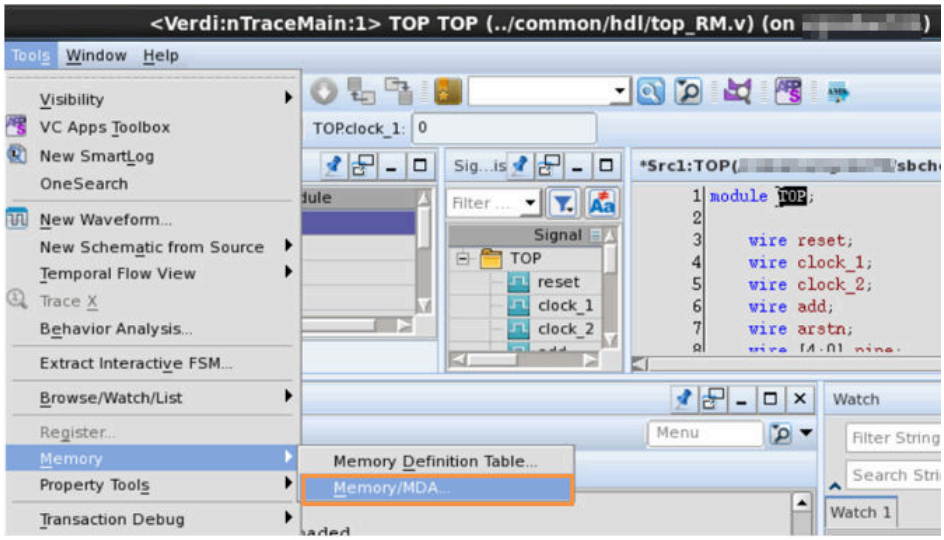


Figure 32 Memory Menu in nWave

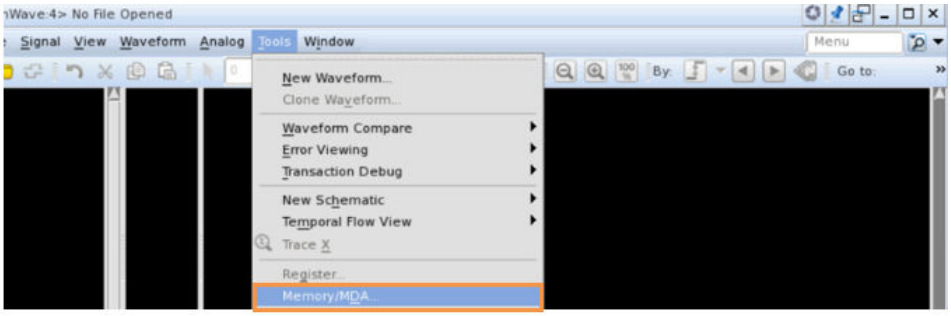
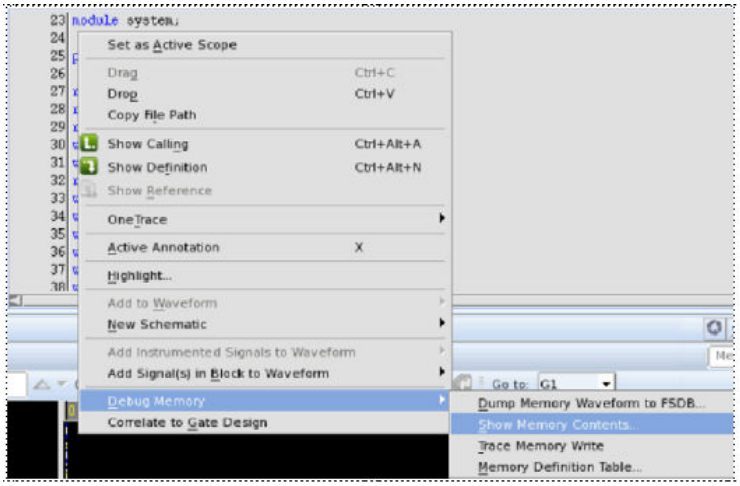


Figure 33 Memory Options From Signal Pane



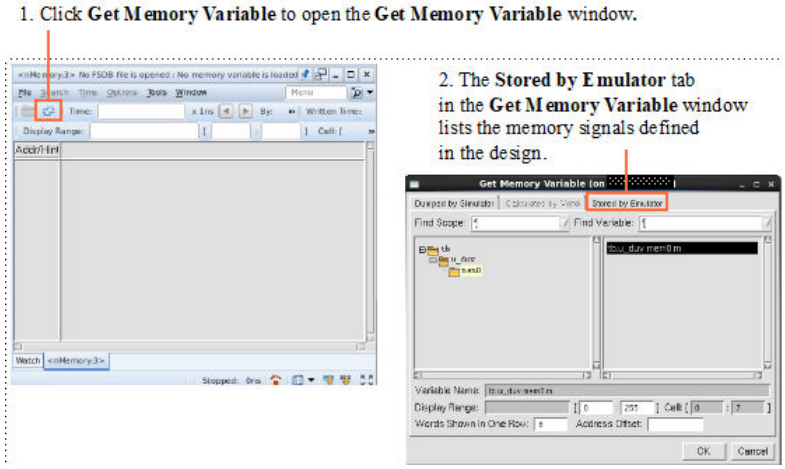
Feedback

nTrace/nWave

In **nTrace**, to get the list of memory signals from **zRci** and **Simulation Acceleration**, choose **Tools** > **Memory** > **Memory/MDA**.

If there are memory signals defined in the design, the signals are automatically retrieved and listed in the **Stored by Emulator** tab of the **Get Memory Variable** window.

Figure 34 Get Memory Variable Window



Note:

The Stored by Emulator tab is enabled only if the design has memory signals.

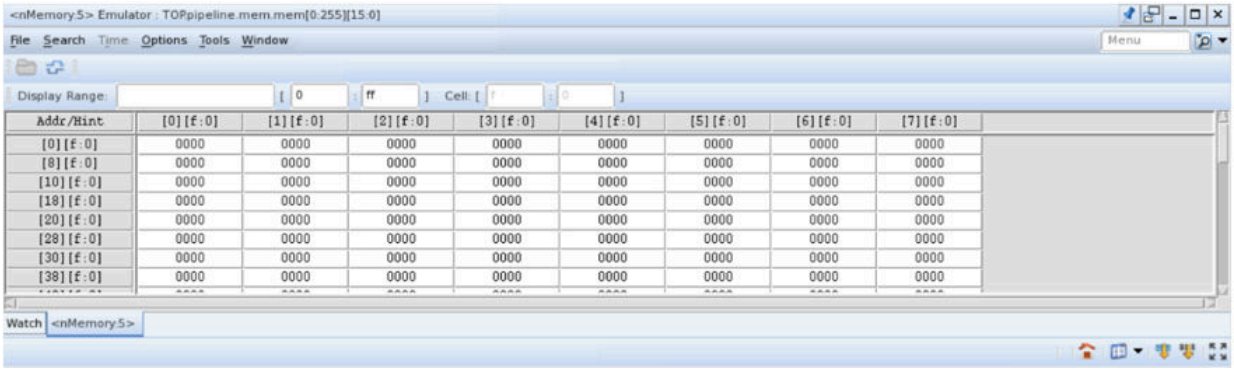
nMemory

In **nMemory**, you can select the memory variable, configure the address, or word number in a row, or address offset, then add to the **nMemory** window.

The **nMemory** window displays "**Emulator: memory variable name**".

Feedback

Figure 35 nMemory Window



Note:

"Time" menu and toolbar options, such as "time text", "next", and "previous" are disabled because the nMemory window shows only the current emulation runtime memory.

Note:

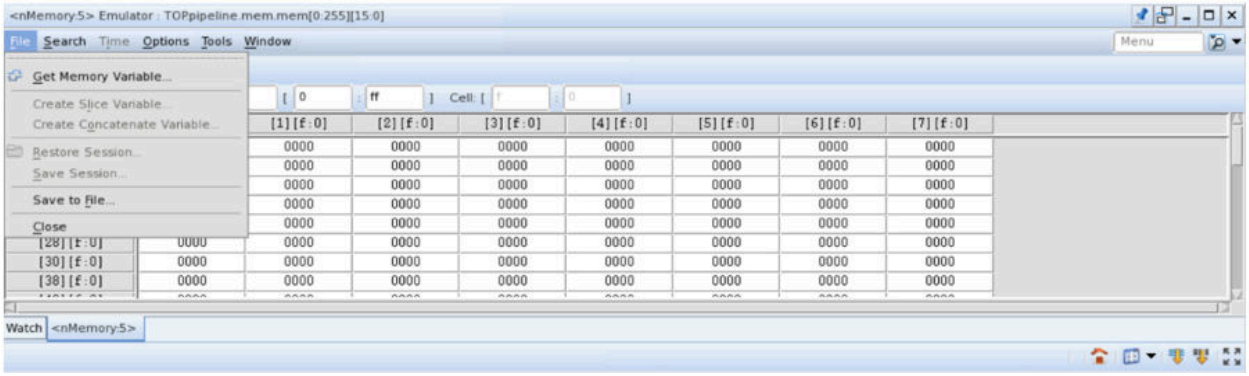
If the emulation is under the cycle mode, all "time" widgets in the toolbar are hidden.

If there is a change in the emulation runtime, **nMemory** updates the content accordingly.

In **nMemory**, the following menu commands are not supported:

- Create Slice Variable
- Create Concatenate Variable
- Restore Session
- Save Session

Figure 36 Unsupported Menu Commands



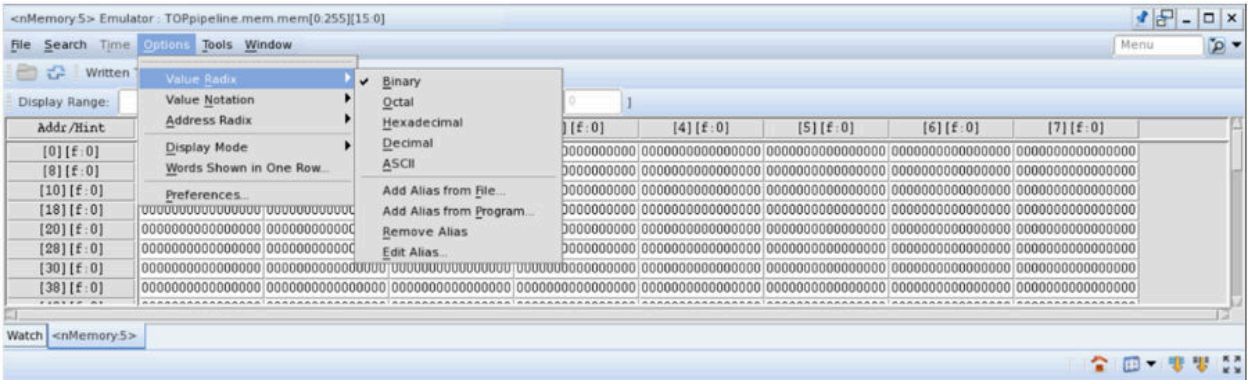
Note:

If you reload Verdi, the **nMemory** window will be empty.

If you restore a session, the **nMemory** window will not be restored.

The menu commands available in the **Options** menu are supported.

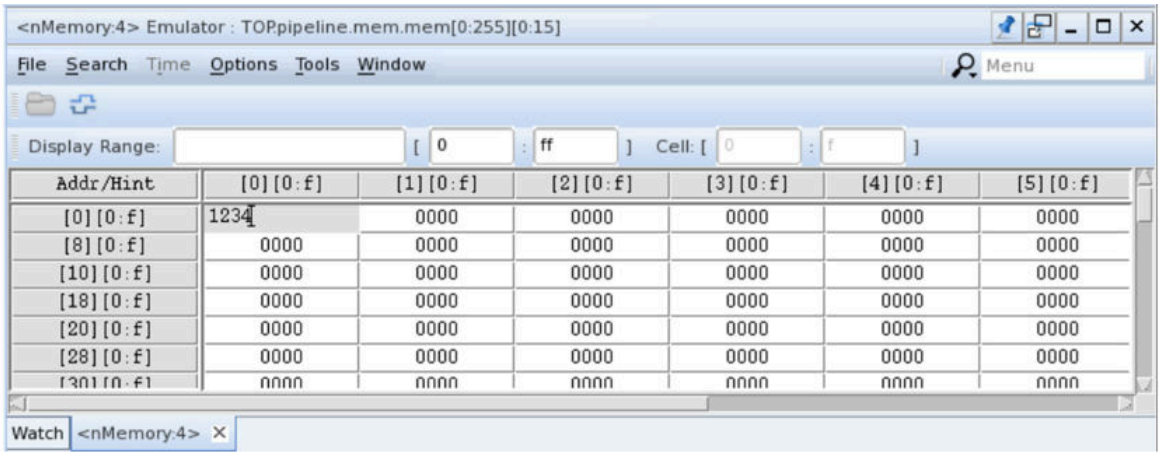
Figure 37 Options Menu



Modifying zRci Memory Content in nMemory

To modify **zRci** memory content in **nMemory**, perform the following steps:

1. Double-click the **zRci** memory cell to modify the memory content as shown in the following figure:



2. Type the value and then press "**Enter**" or leave the cell to write the memory into **zRci**.

3. Modify the memory with a current value radix, which is set using **Options** > **Value Radix**.

   The supported radix can be binary, octal, decimal, or hexadecimal. Otherwise, a message is displayed as follows.

   ```
   "Failed to write memory to emulation, support bin|oct|dec|hexa radix
    only. Check the Options->Value Radix and try again."
   ```

4. Modify the memory with a valid format.

   If you modify the memory with an invalid value format, a message is displayed as follows:

   ```
   "Failed to write memory to emulation. Check the value and try again."
   ```

10

ZeBu Design Manipulation Using Verdi

Verdi supports the ZeBu design manipulation based on Knowledge Database (KDB)-Engineering Change Order (ECO) to view the original source code. In addition, Verdi annotates symbols for the modified netlist in the source code.

ZeBu compilation generates the changelist database for design manipulation. Verdi-elaborator fetches KDB from the original design with the changelist to modify the KDB netlist.

nTrace and **nSchema** uses the manipulated KDB to annotate the symbols for ZeBu.

Prerequisites

- Set the following environment variable to use this feature: *setenv VERDI_ZDM*
- Specify the *-lca* option in the commands.

Subtopics:

Command-Line Options for Design Manipulation

To perform design manipulation in the command-line mode, use the following commands:

1. `%verdi -lca -emulation`

   Verdi displays **Emulation** in the menu bar.

2. `% verdi -lca -emulation --zebu-work <zebu.work> --cldb <CLDB>`

a. Automatically import KDB from `simv.daidir` and set the root scope.

b. Open the **Design Manipulation** table view and specify `<CLDB>` to open the specified CLDB file. Otherwise, open the default CLDB file (`…zcuiUC.work/vcs_splitter/simv.daidir/zebu_cldb.db`).

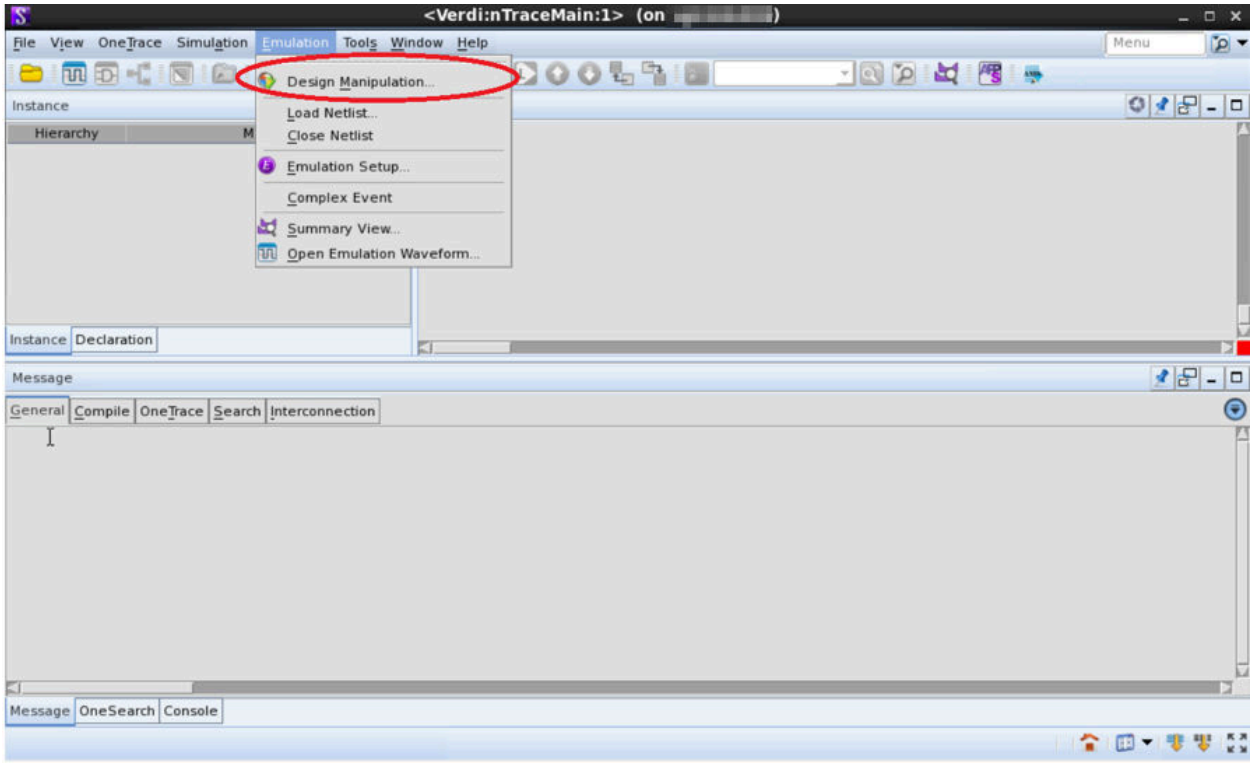
3. `%verdi -emulation -help`

`--cldb <CLDB name>`: Specify the CLDB file name.

Design Manipulation With Table View

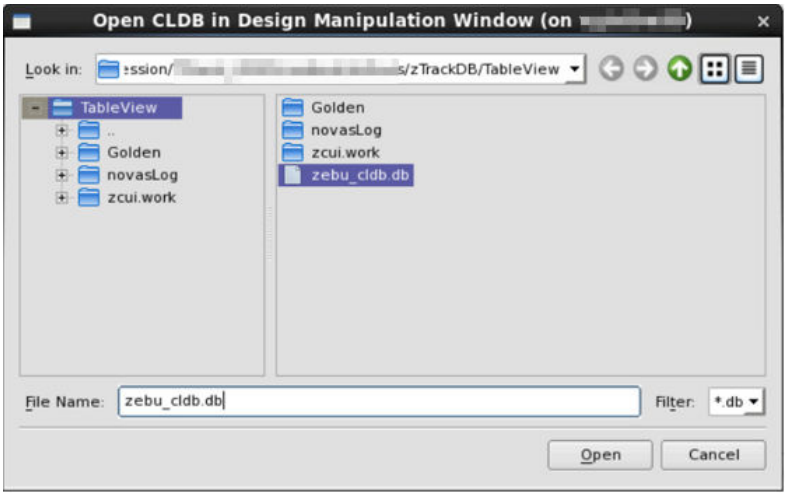
In Verdi **nTrace**, the design manipulation command is available under the **Emulation** menu.

Figure 38 nTrace With Design Manipulation Menu Command



To specify CLDB and load it, choose **Emulation** > **Design Manipulation**. The **Design Manipulation** window appears as follows:

Figure 39 Design Manipulation Window

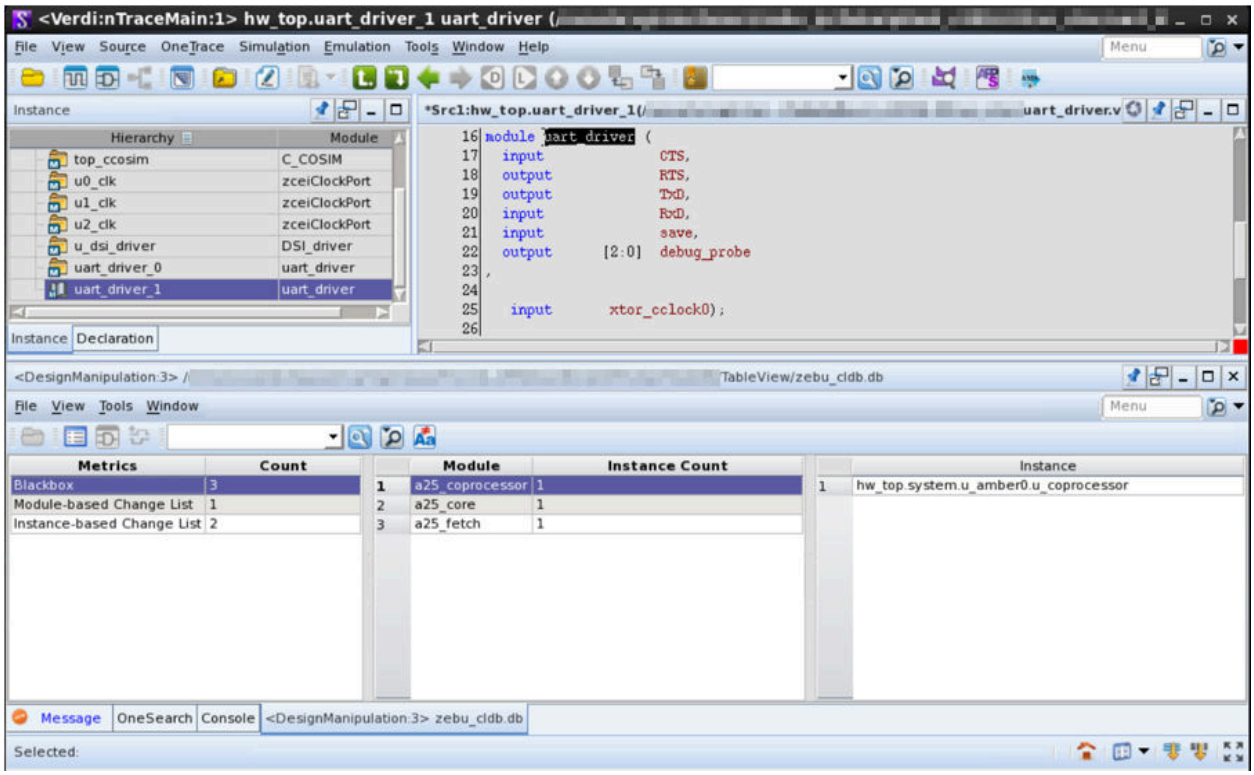


When you choose the database and click **Open**, the selected database is uploaded in the **Design Manipulation Window**. The **Design Manipulation Window** lists the number of

Feedback

black boxes, module-based changelist, and instance-based changelist, as shown in the following figure:

Figure 40 Design Manipulation View of a Design



The metrics are displayed in table view.

Feedback

Black-Box Module and Instance Changelists

Blackbox displays module and instance in two tables as follows:

Figure 41 Blackbox Module Information in Table View

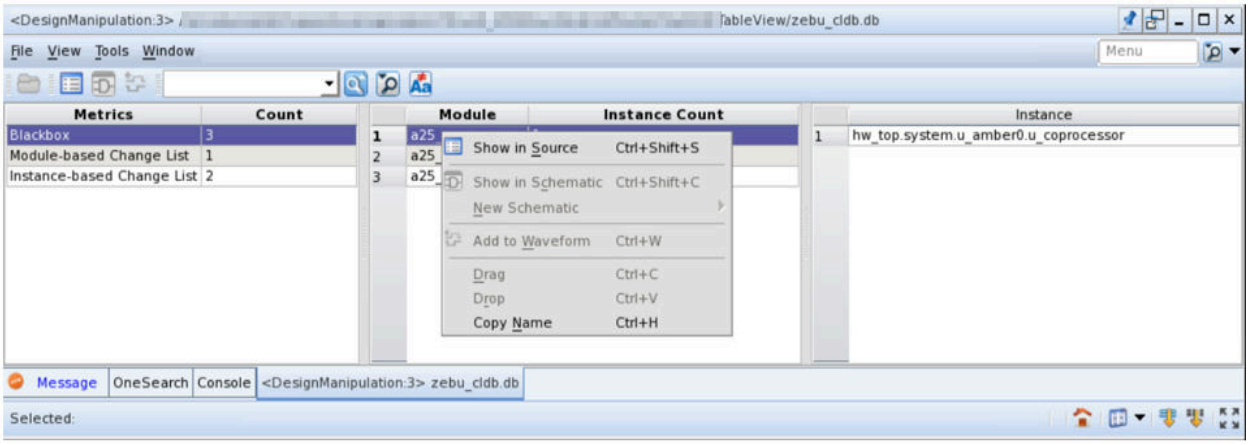
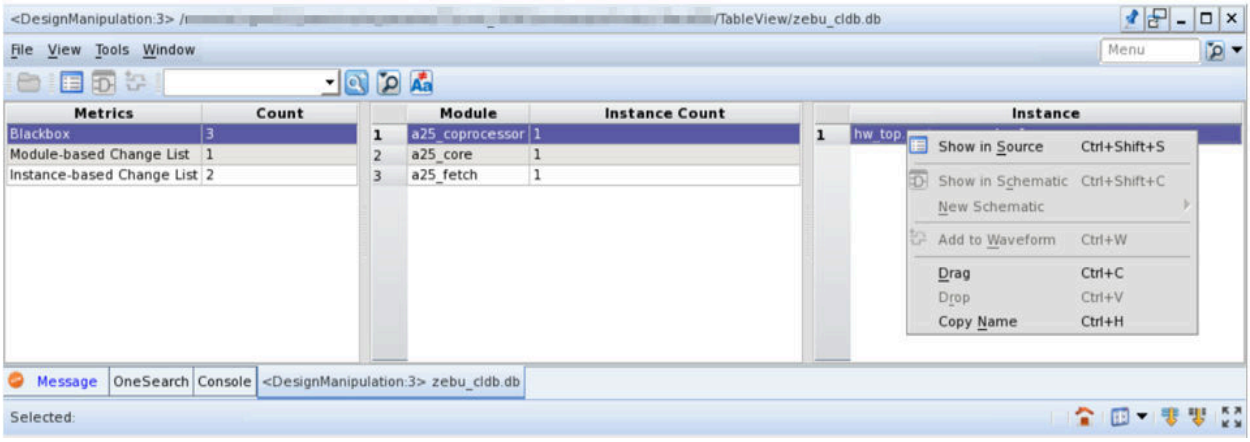


Figure 42 Blackbox Instance Information in Table View



To view the source and copy the name, use the right-menu options **Show in Source** and **Copy Name**, respectively for a module in the black box.

To view the source, drag and copy the name, use the right-menu options **Show in Source**, **Drag**, and **Copy Name**, respectively for an instance in the black box.

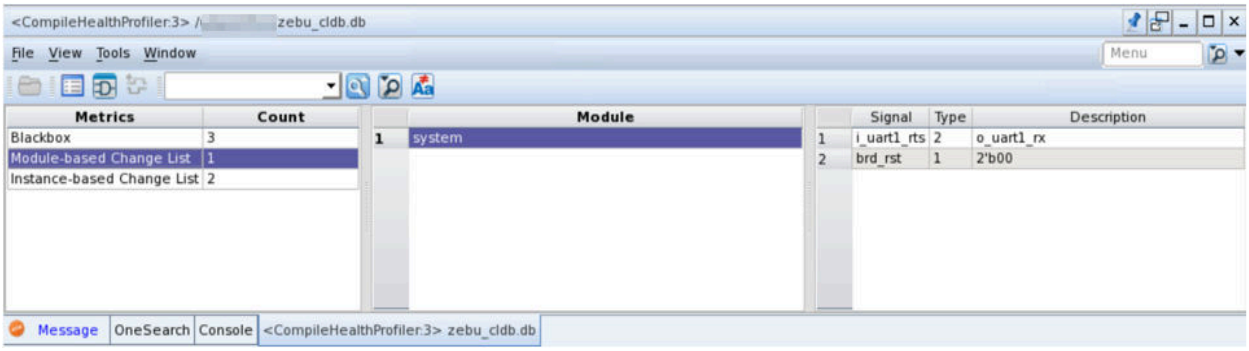
You can also double-click both module and instance tables to run the **Show in Source** command.

Feedback

Module-Based Changelists

The module-based changelists table displays module and signal changelist details as follows:

Figure 43 Module-Based Changelist Details in Table View



To view the source, schematic, and copy the name in the **Module** table, use the right-menu options **Show in Source**, **Show in Schematic**, and **Copy Name**, respectively.

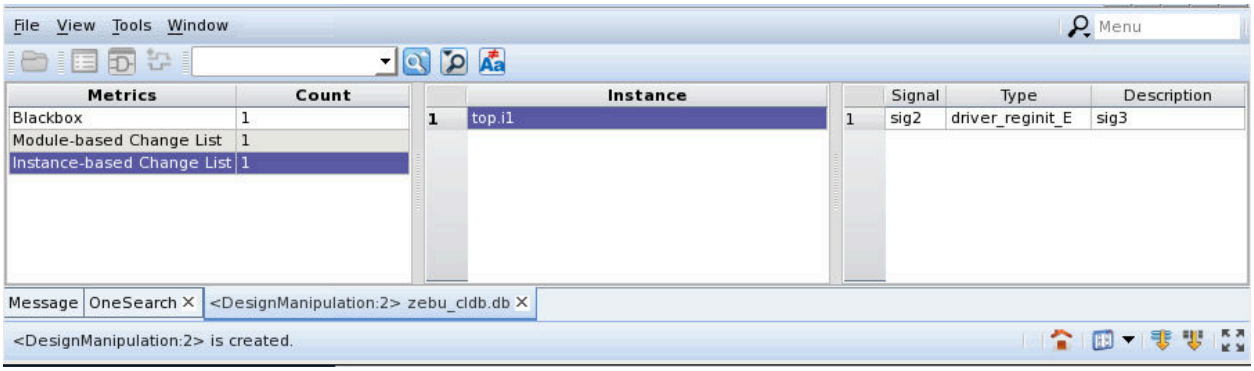
Note:

The right-menu option is not supported in the Signal table view.

Instance-Based Changelists

The instance-based changelists table displays instance and signal changelist details as follows:

Figure 44 Module-Based Changelist Details in Table View



To view the source, schematic, drag, drop, and copy the name in the Instance table, use the right-menu options **Show in Source**, **Show in Schematic**, **Drag**, **Drop**, and **Copy Name**, respectively.

Feedback

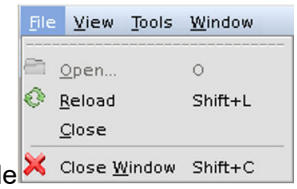
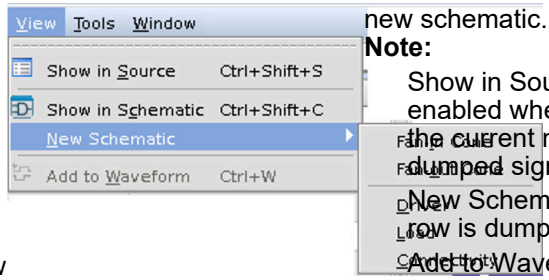
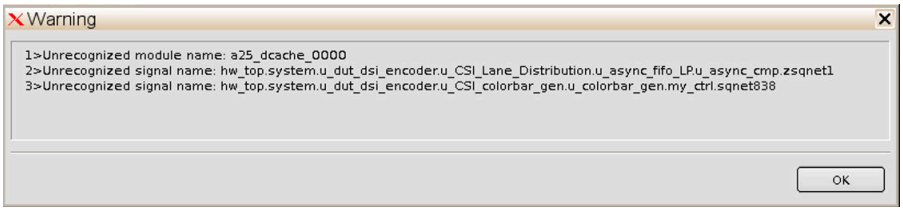
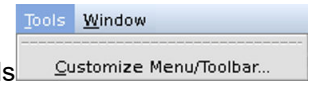
You can also double-click the instance to run the **Show in Source** command.

To view the source, schematic, new schematic, drag, and copy the name in the Signal table, use the right-menu options **Show in Source**, **Show in Schematic**, **New Schematic**, **Drag**, and **Copy Name**, respectively.

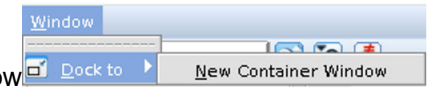
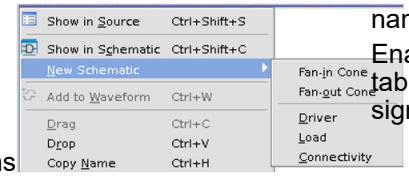
Design Manipulation Menu and Toolbar Options

Menu Bar

The following table lists the menu commands available in the **Design Manipulation**window.

Menu Commands	Description
File 	Enables to reload and close the **Design Manipulation** window.
V iew 	Enables to view the source, schematic, and open a new schematic. **Note:** Show in Source and Show in Schematic are enabled when Verdi has imported a design, and the current metric item table has selected a row of dumped signals or a module. New Schematic menu is enabled when a selected row is dumped with a signal name. Add to Waveform is enabled when Verdi has loaded a waveform, and the current metric item table has selected a row of dumped signals. If Verdi does not recognize any modules or signals, a warning message is displayed as follows: `"Unrecognized module/signal name: %s"` 
Tools 	

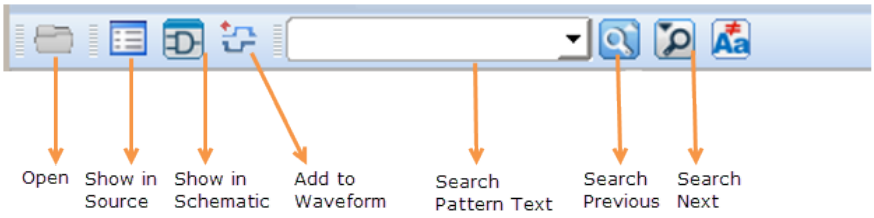
Feedback

Menu Commands	Description
Win  dow	
Right-menu opti  ons	Enables the **Drag** command only when a selected row is dumped the module instance name or the signal name. Enables the **Drop**command only when the current table is dumped with the module instance name or the signal name.

Tool bar

The following figure displays the toolbar options.

Figure 45 Design Manipulation Toolbar



Design Manipulation with nSchema

Based on the ZeBu Design Manipulation (ZDM), there is some netlist change on **nSchema** for a ZeBu design manipulated signal as follows:

- If the net name is shown, "(z)" is included after the net name in cyan on the net.
- If the net name is not shown, "z" is highlighted in a position, where the net name is shown in cyan on the net.
- **nSchema** highlights the net lines in cyan.
- The signal type is named as "ZDM Signal" and can be found in the **Preference-Color/ Font-Type** drop-down list.
- **nSchema** is applied on a full and flatten window.

Feedback

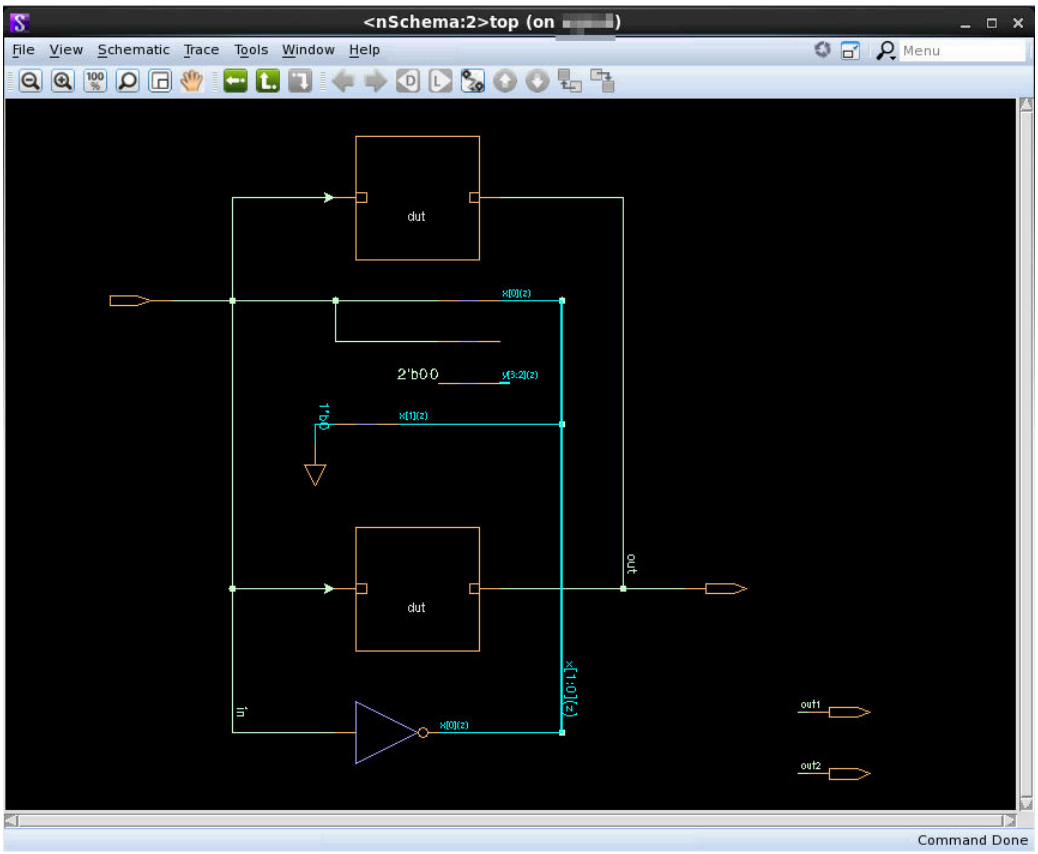
Example

Consider the following ZDM configuration to view the top scope view.

```
zgate fd -clock_name driverClk -module dut -signal x[0]
zgate fd -clock_name driverClk -rtlname {top.dut1.x[1]}
zgate fd -clock_name driverClk -module dut -signal y.a[0]
zgate fd -clock_name driverClk -module top -signal y
zgate fd -clock_name driverClk -rtlname {top.y[1]}
zgate fd -clock_name driverClk -rtlname {top.x}
```

Top scope view: "z" is included after the net name in cyan as follows:

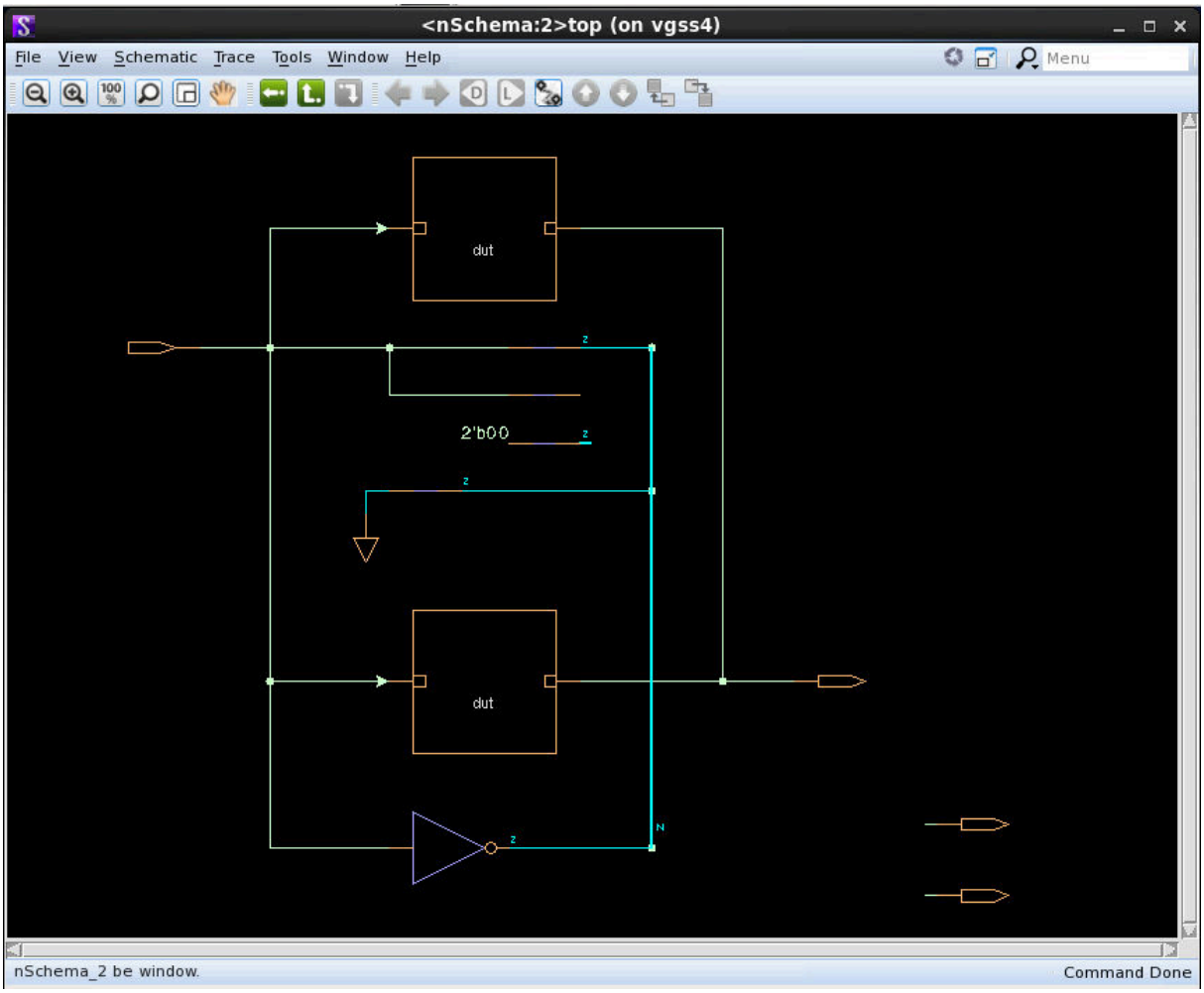
Figure 46 Top Scope View When Net Name Is Shown



Feedback

Top scope view: "z" position and the net is in cyan as follows:

Figure 47 Top Scope View When Net Name Is Not Shown



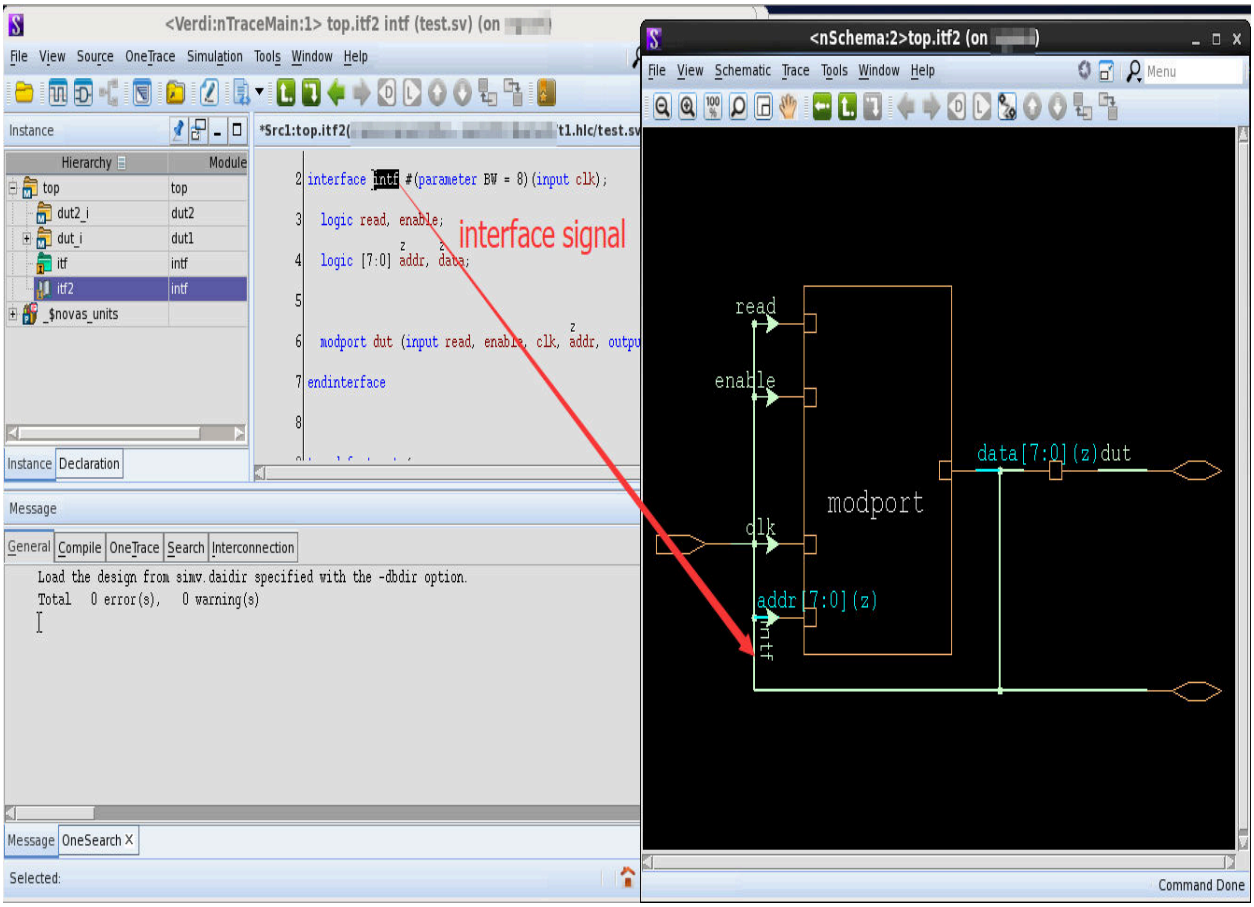
Limitation

For an interface in the design, **nSchema** shows it as a signal. However, the signal cannot be highlighted with the ZDM feature. For signals in interface, the behavior is same as others and can be highlighted as ZDM signal.

In the following figure, the interface "`intf`" in scope "`top.itf2`" cannot be highlighted with the ZDM feature in **nSchema**.

Feedback

Figure 48 Interface Signal With ZDM Feature in nSchema



Design Manipulation With Temporal Flow View

HDB references the change-list database, and contains a set of APIs for other component to get ZeBu design attributes. Temporal Flow View (TFV) shows "Z" on zGate if the attribute can be obtained from HDB API.

Feedback

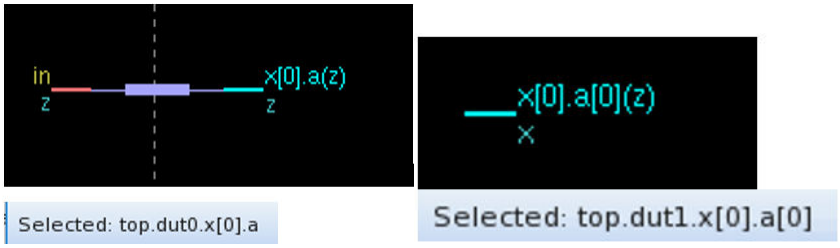
TFV provides support for the following:

- Shows "Z" above the output pin and marks "Z" in bright blue.
  - If an output pin has both black box and zGate notation, only zGate notation is shown on the output pin.
  - TFV marks "Z" only for the net that is modeled in our lower level engine. If you specify zGate on an immediate hierarchy name that IS NOT modeled in our lower-level engine, it can be directly traced through it.
- Support instance-based and module-based configuration in the UTF file.

TFV for Module-Based Signals

In this example, assume that *zgate fd -clock_name driverClk -module dut -signal x[0]* is specified in the UTF file. Signals *x[0]* in the instances created by module DUT are marked as "Z". In this case, two instances *dut0* and *dut1* are created by DUT, and their signals *x[0]* are marked with "Z".

Figure 49 TFV for Module-Based Signals



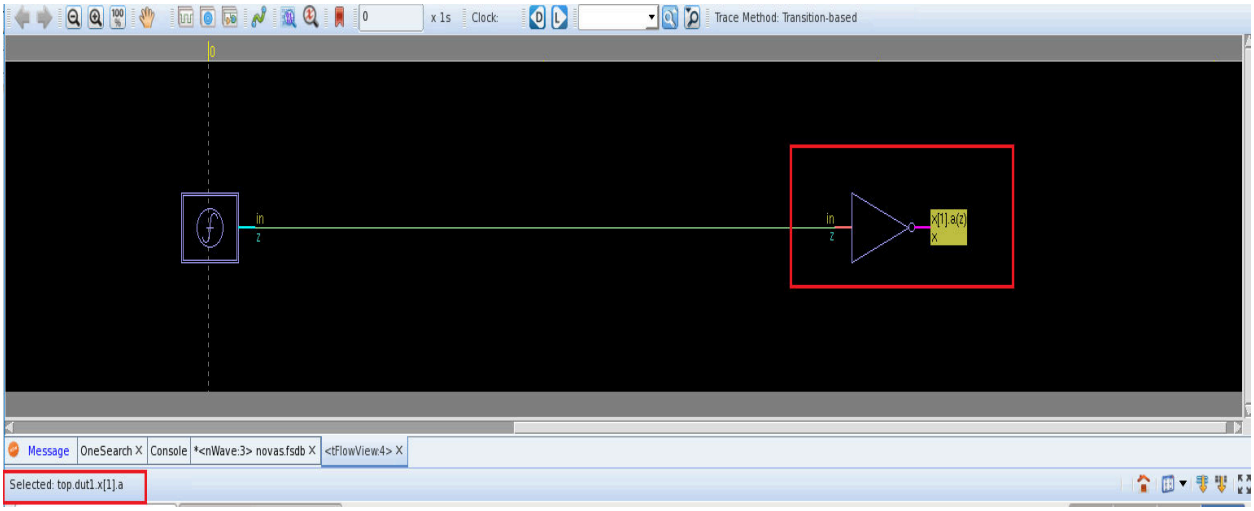
TFV for Instance-Based Signals

In this example, assume that `zgate fd -clock_name driverClk -rtlname {top.dut1.x[1]}` is specified in the UTF file.

The following figure shows overall view of TFV when Z is specified for zGate.

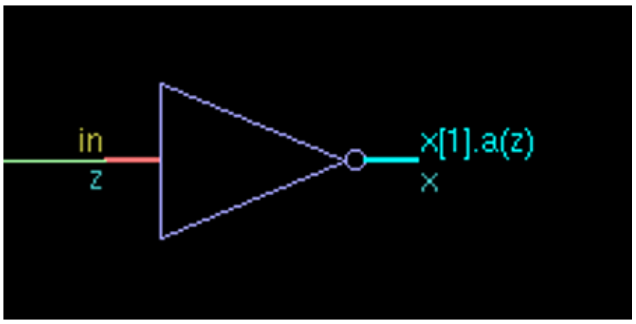
Feedback

Figure 50 TFV: Overall View for Instance-Based Signals



The following figure shows the detail view of the pink rectangle in the preceding figure.

Figure 51 Detail View of an Instance-Based Signal



The following figure highlights the full hierarchy name for the signal that is derived by zGate, which is shown in red rectangle in the overall view.

Figure 52 Hierarchy Name for a Signal

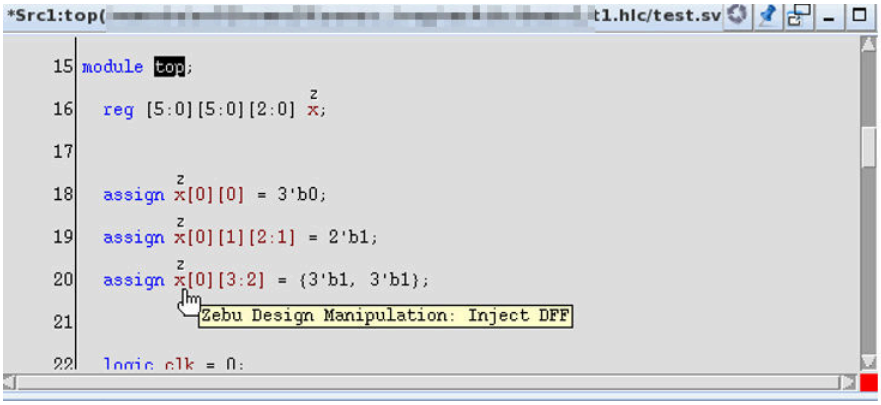
Feedback

Design Manipulation With Source and Signal View

Source View

The Source view shows "z" above ZDM variables; This annotation is shown after current existent upper annotation, such as power, AMS, and so on. It allows you to turn off ZDM upper annotation. In addition, the Source view shows a tip if you hover the pointer on ZDM variables.

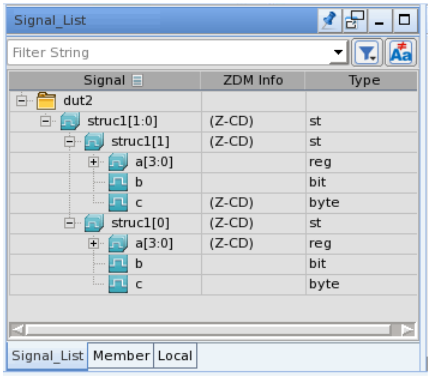
Figure 53 Source View



Signal List

Signal list shows ZDM information in the "**ZDM Info**" column, "*Z-DFF*" for FF injection, and "*Z-CD*" for driver change. ZDM information is shown after power information, such as "(*iso in*) (*Z-DFF*)". A signal can be treated as ZDM if its element/partial slice is ZDM.

Figure 54 Signal View



Feedback

Tcl Commands

The following Tcl commands are supported for the Design Manipulation window operation.

- `chpCreateWinAct`
- `chpCloseWinAct [-win winId]`
- `chpOpenAct [-win winId] dbname`
- `chpCloseAct [-win winId]`
- `chpReloadAct [-win winId]`
- `chpFindNextAct [-win winId] search_pattern`
- `chpFindPrevAct [-win winId] search_pattern`
- `chpMatchCaseAct [-win winId] [-on]`
- `chpShowInSrcAct [-win winId]`
- `chpShowInSchAct [-win winId]`
- `chpAddToWaveAct [-win winId]`
- `chpDragAct [-win winId]`
- `chpDropAct [-win winId]`
- `chpCopyNameAct [-win winId]`
- `chpTblAct [-win winId] -tbl tblName -select rowIndex`

Visualizing Design Modifications Using Verdi

The original source code and the schematic view are annotated to notify the design behavior changes due to UTF commands. The annotation is done through “z” symbol (z=modified by ZeBu) on top of the signal name. The following UTFs are supported:

- All flip-flop insertions through the `zgate` command.
- Force assign commands with constant driver, register driver or change in design driver with another signal.
- Black box UTF. Instead of “z”, a special symbol is used in the instance view. The black box is grayed out in **nSchema**.

Feedback

Enabling Visualization of Design Modifications Using Verdi

To activate this feature in ZeBu, set the following environment variable before compiling the design with **zCui**:

```
setenv SNPS_VCS_INTERNAL_UTF_VERDI_ANNOTATION 1
```

To activate this feature in Verdi, use the following command and click **Emulation** > **Design Manipulation**:

```
verdi -emulation -lca --zebu-work <zebu.work>
```

Verdi Command-Line Options

To perform design manipulation in the command-line mode, use the following command:

```
%verdi -lca -emulation --zebu-work <zebu.work> --cldb
```

It automatically imports KDB from `simv.daidir` and sets the root scope. Also, it automatically opens the **Design Manipulation** table view with the default CLDB file.

Supported ZDM Types

The following ZDM types are supported and available on the **Signal detail** table:

- "change driver to constant by force assign";
- "change driver to another design net by force assign";
- "DFF injection by *zgate*";
- "change driver to write back register by force assign";
- "insert runtime programming register to force signal to given value by dynamic *zforce*";
- "insert dedicated top port to force signal to given value by static *zforce*";
- "inject corresponding wire with runtime programming register by dynamic *zinject*";
- "inject corresponding wire with a created top port by static *zinject*";

The **Signal** column shows the original signal string in the change list command.

Visualizing Design Modifications in Verdi

The following table explains the various views available in Verdi for each design modification.

Feedback

Design Modification View	RTL view	Signal List View (ZDM Information)	Design Manipulation View (Type)
`zforce` (module or instance based)	Shows “z” in RTL	Shows “Z-F” in ZDM information	Insert a runtime programming register to force the signal to a given value using dynamic `zforce`. Insert a dedicated top port to force the signal to a given value using static `zforce`.
`zinject` (module or instance based)	Shows “z” in RTL	Shows “Z-I” in ZDM information	Inject a corresponding wire with the runtime programming register using dynamic `zinject`. Inject a corresponding wire with a created top port using static `zinject`
`zgate` (module or instance based)	Shows “z” in RTL	Shows “Z-DFF” in ZDM information	DFF injection using `zgate`
Blackbox (module based only)	Special symbol on instance view; grey out instance on *nSchema*;RTL is visible, but debug features are greyed out;Signal list is visible	<Empty>	Shows `<libname@module>`. Shows module, instance count and `libname`
Force `assign -value 0/1`(module or instance based)	Shows “z” in RTL	Shows “Z-CD” in ZDM information	Change driver to constant by force assign”

Feedback

Design Modification View	RTL view	Signal List View (ZDM Information)	Design Manipulation View (Type)
Force `assign -type reg` (module or instance based)	Shows “z” in RTL	Shows “Z-CD” in ZDM information	Change driver to write back register using force assign
Force `assign -source_rtl -rtlname` (module or instance based)	Shows “z” in RTL	Shows “Z-CD” in ZDM information	Change driver to another design net using force assign

11

Simulation Acceleration Interactive Mode

Verdi interactive debug supports the Simulation Acceleration flow. With Simulation Acceleration interactive debug, you can debug both software (testbench) and hardware (DUT) in the Verdi interactive debug environment. Simulation Acceleration interactive debug also allows you to output testbench signals and DUT signals into one ZTDB file.

Note:

For more information on Simulation Acceleration, see the Simulation Acceleration User Guide.

For more information, see the following subsections:

- Compilation Requirements
- Invoking Simulation Acceleration Interactive Debug
- Simulation Acceleration Interactive Debug Windows
- Generating Waveform Output in Simulation Acceleration Interactive Debug
- Limitations

Note:

For more information on displaying memory content, see the “Viewing ZeBu Memory Using Verdi” chapter.

Compilation Requirements

To enable Simulation Acceleration interactive debug, compile the design using the following switches:

- *-kdb*
- *-lca*
- *-debug_access+all*

Invoking Simulation Acceleration Interactive Debug

First compile the design in ZeBu flow. To start the coemulation in interactive mode, log on to an emulator host and execute the following command.

```
% verdi -simBin <simv_executable> -i -simXL -emulation -lca
-simDelim <Simulation_Acceleration_run_time_options>
```

Where:

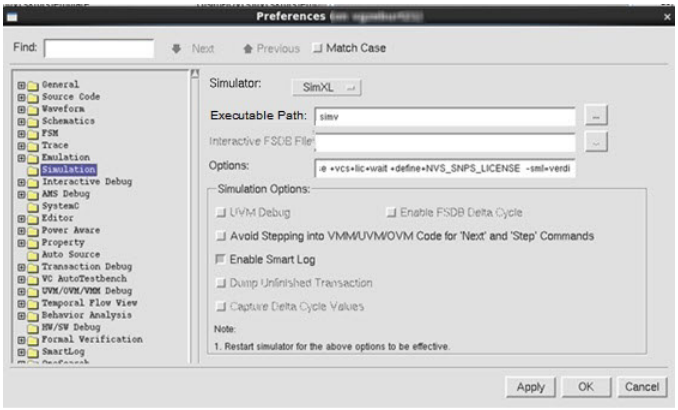
- *-simBin simv_executable*: Specify the simulation executable name, for example simv.
- *-i*: Specifies that Verdi is launched in interactive mode.
- *-simXL*: Specifies that Verdi launches Simulation Acceleration automatically.
- *-lca*: Launches the Limited Customer Availability (LCA) mode.
- *-simDelim*: Specifies Simulation Acceleration runtime options

Example

```
%verdi -simBin simv -i -emulation -lca -simXL -simDelim
 +UVM_TESTNAME=axi_test_wrap +UVM_VERBOSITY=UVM_LOW
```

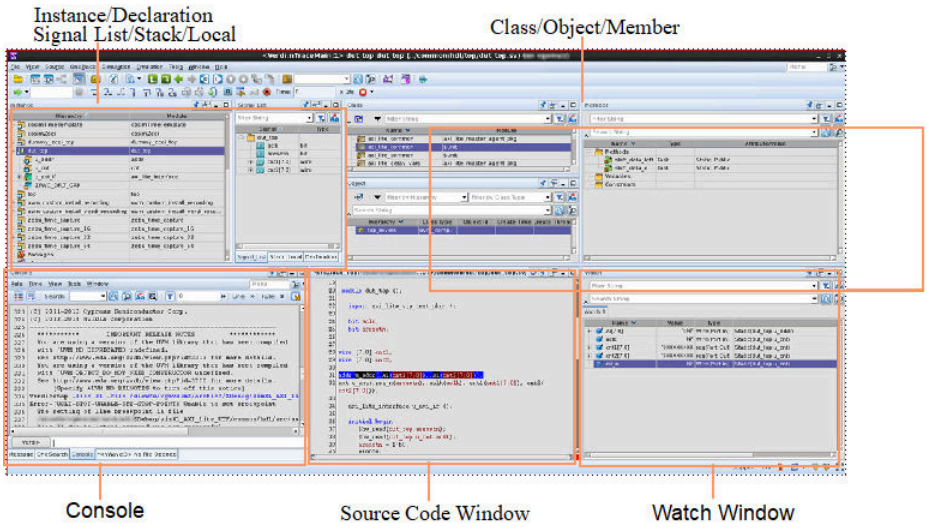
Simulation Acceleration Interactive Debug Windows

Based on the options specified in the command line, the preferences are autopopulated in the **Preferences** dialog box and Simulation Acceleration is automatically invoked. The following screenshot shows the Simulation Acceleration mode selected in the **Preferences** dialog box.



The following figure shows the main windows of Simulation Acceleration interactive debug.

Feedback



Simulation Acceleration interactive debug supports the following features:

- Local, watch, class, object, member, stack views
- Next, step, run, step in thread/constraint solver/testbench
- Breakpoints
- Go to active file/line, restart, quit, kill

For more information, see the following subsections:

- Source Code
- Stack View
- Local View
- Class and Member Windows
- Object and Member Windows
- Simulation Acceleration Interactive Toolbar
- Verdi Console - Verdi Command-Line Bar

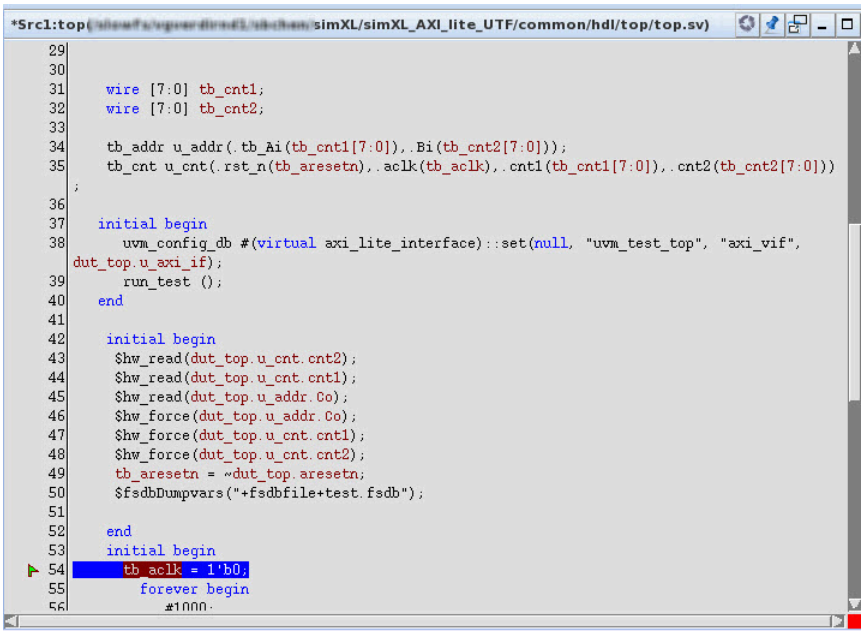
Source Code

From the Source Code window, you can:

- View both testbench and DUT source code
- Set breakpoint on testbench code

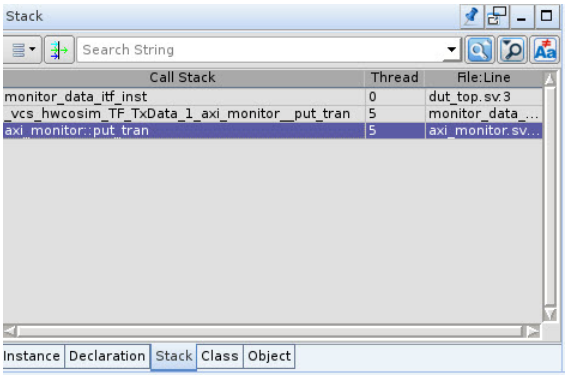
- Drag-and-drop signals to Watch Window
  - Testbench signals
  - DUT signals that allow testbench access (setting up inside UTF or through `$hw_read/$hw_force` system tasks)
- Drag-and-drop signal to dump file

The following screenshot shows the Source Code window.



Stack View

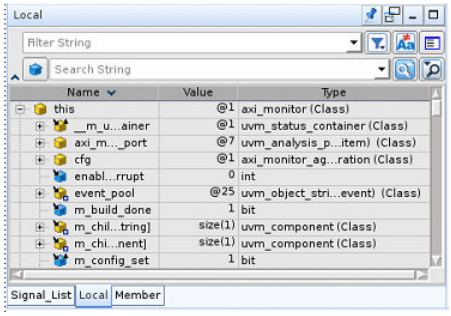
The Stack view for simulation shows call-stack of the VCS kernel. The following screenshot shows the Stack view



Feedback

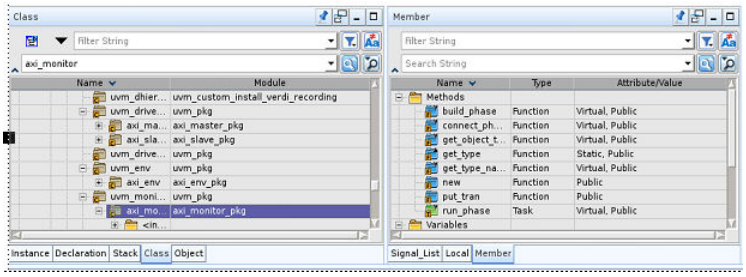
Local View

The Local view for simulation shows local variables of the VCS kernel. The following screenshot shows the Local view.



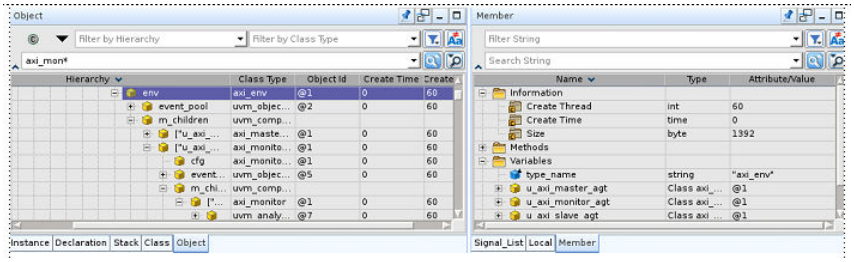
Class and Member Windows

Class and Member windows show the testbench class and members. The following screen shot shows the Class and Member windows.



Object and Member Windows

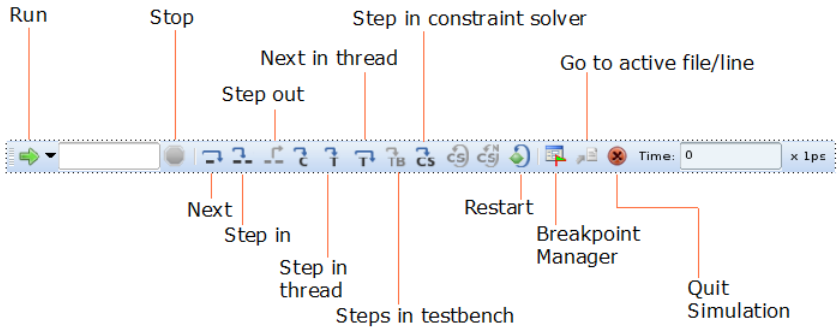
Object and member windows show the testbench object and the members. The following screenshot shows the Object and Member windows.



Feedback

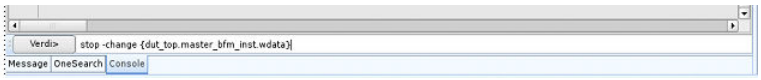
Simulation Acceleration Interactive Toolbar

To support debug in Simulation Acceleration mode, a new Simulation Acceleration interactive toolbar is added. See the labels in the following figure to know more about this toolbar.



Verdi Console - Verdi Command-Line Bar

Simulation Acceleration interactive debug also supports users to run UCLI command in the Verdi command-line bar:



Note:

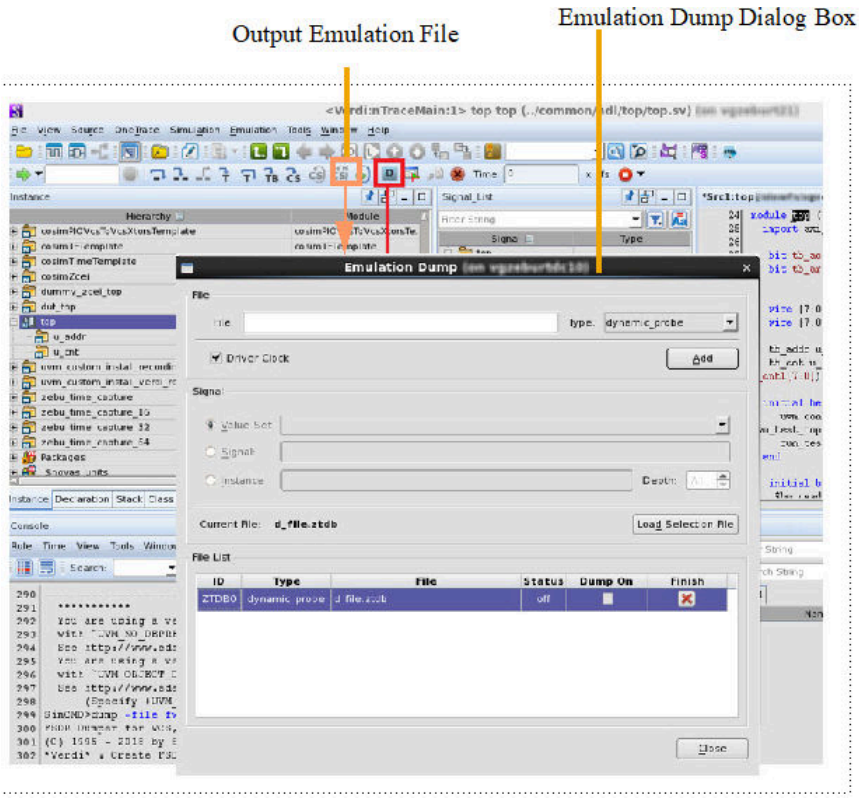
For more information on Verdi windows and toolbars, see the Verdi User Guide.

Feedback

Generating Waveform Output in Simulation Acceleration Interactive Debug

You can output both testbench and DUT into one ZTDB file. To output a ZTDB file, perform the following steps:

1. Click the **Output Emulation File** icon (D icon). The **Emulation Dump** dialog box appears.



The following table describes the GUI elements found in the *Emulation Dump* dialog box.

Table 4 Description of Emulation Dump Dialog Box

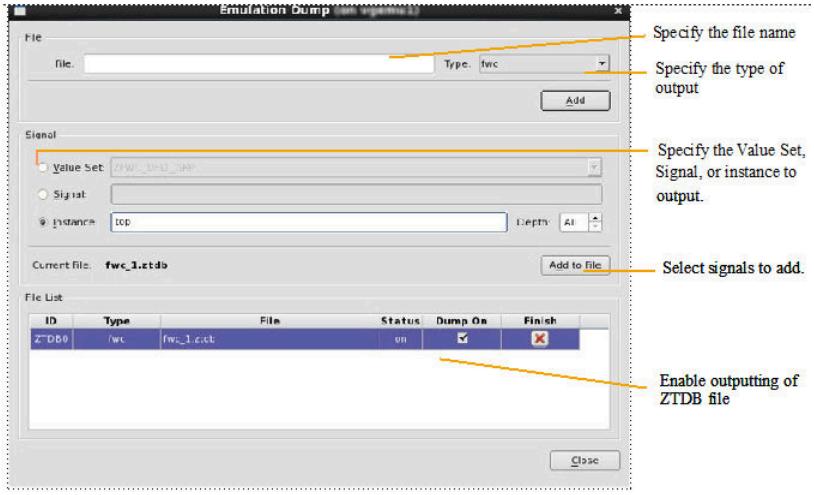
GUI Element	Description
File (line edit)	Add new file name.
Type	Specify new file dump type (FWC or dynamic_probe).
Driver Clock	For the dynamic_probe type, you can select the Driver Clock check box.

Feedback

GUI Element	Description
Value Set	Add hardware value set to file.
Signal	Add software signals to file.
Instance	Add software signals to file.
Load Selection File	Load selection file for dynamic probe type file.
File List	Display current active dump file.
Dump On	Enabled/disabled ZTDB outputting.
Finish	Close ZTDB file

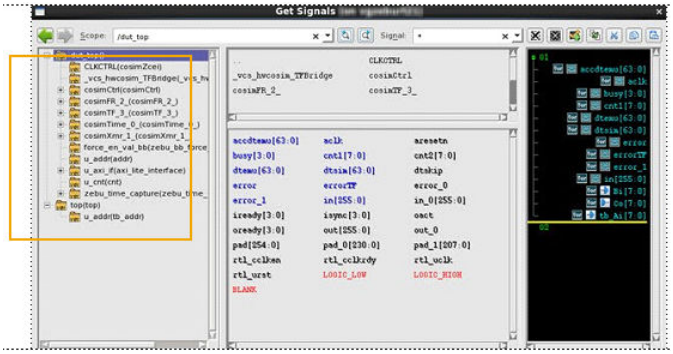
2. In the **Emulation Dump** dialog box, perform the following:
   - Specify the file name and type to output.
   - Specify the **Value Set**, **Signal**, or **Instance** to output.
   - Select file and add signals to file. FWC and dynamic probes are supported.
   - Enable outputting of ZTDB file.

The following screen shot shows how to perform these steps in the **Emulation Dump** dialog box.

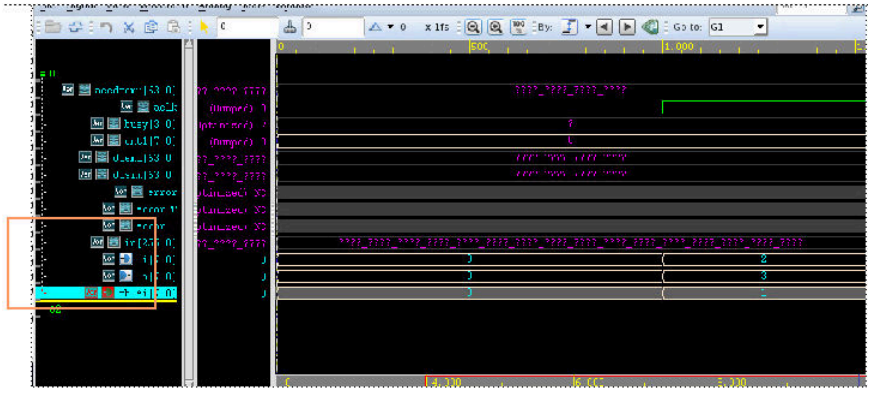


After the waveforms are outputted, open the **Get Signals** dialog box to view the ZTDB. The hardware and software signals are all in one ZTDB file, as highlighted in the following screen shot.

Feedback



The following image shows the corresponding FWC waveform.



Limitations

The following functionalities are not supported in Simulation Acceleration Interactive Debug using Verdi:

- Reverse debug
- UVM aware debug
- Active Annotation
- Rebuild and restart
- Go to value assignment
- Save/Restore state
- Checkpoint functions
- CBug functions

- Next in C/C++
- Save/Rewind checkpoint

12

Analyze FSDB Using the zwdutils Utility

FSDB can be analyzed more efficiently using the FSDB utilities `fsdbreport`, `fsdb2saif`, and `fsdbextract`.

- The `fsdbreport` utility generates an ASCII text file of value change lists for specified signals from an FSDB file.
- The `fsdb2saif` utility converts an FSDB file into a backward SAIF format.
- The `fsdbextract` utility modifies an existing FSDB file after some process. This utility extracts signals, scopes, and time periods of interest to a new FSDB file without repeating the time-consuming simulations. Depending on the dumped signal, the full or partial range of a bus signal are extracted. Also, the FSDB file can be split into multiple files by size constraint.

The `zwdutils` utility is created to support most of the options of `fsdbreport`, `fsdb2saif`, and `fsdbextract` utilities on ZeBu Waveform Database (ZWD) waveform format as mentioned in the following sections.

For more information, see the following subsections: Use Model and Limitations.

Use Model

`zwdutils` is a utility which supports `fsdbreport`, `fsdb2saif`, and `fsdbextract` utilities on ZWD waveform format and `zwdutils` utility only supports ZWD as input.

Note:

To enable and use the `zwdutils` utility in Verdi, you must specify the -lca option during compilation.

For more information, see the following subsections:

- fsdbreport Utility on ZWD
- fsdb2saif Utility on ZWD

Feedback

fsdbreport Utility on ZWD

Usage

```
zwdutils --tool=report <zwd_file_name> [Options]
```

Options

Options	Explanation
`-a alias_name`	Defines the alias for the output signal.
`-af alias_file`	Specifies an `nWave` waveform alias file.
`-bt time[unit]`	Specifies the begin time of the report. If omitted, the begin time of the ZWD file is used. The time unit is `Ms`, `Ks`, `s`, `ms`, `us`, `ns`, `ps`, or `fs`. The default time unit is `ns`.
`-cn column_number`	Defines the number of columns for the report, including the time column. Column number is set to be 0 or an integer larger than 1. When setting to `0`, the signal name and its value are not printed in the format of a table, but line by line. **Note:** This option is ignored if the `-csv` option is also specified.
`-csv`	Saves the output report file in the CSV format. If this option is specified with `-cn` and `-w`, these options are ignored. **Note:** If the selected signals contain stream, coverage, or SVA type signals, the `-csv` option is ignored and non-csv format is output.
`-et time[unit]`	Specifies the end time of the report. If omitted, the end time of the ZWD is used. The time unit is `Ms`, `Ks`, `s`, `ms`, `us`, `ns`, `ps`, or `fs`. The default time unit is `ns`.
`-exclude_scope "scope1" ["scope2" … "scopeN"]`	Excludes signals under the specified scopes. Each scope is enclosed with double quotes. To exclude subscopes of the specified scopes, the wildcard character "*" is appended at the end of the scopes.
`-exp expression`	Reports values when the expression changes to `true (==1)`.

Feedback

Options	Explanation

`-f config_file`	Specifies a text file which defines all the options except `-h` and `-f`. The pound (#) sign is added to the beginning of a line as a comment line.
	Example:
	`##comment`
	` -bt 10 -et 100 -s "/system/i_cpu/*"`
	` #comment`

`-h | -help`	Prints the help message.
`-level level_depth`	Specifies the number of levels to be dumped under the specified scope. Use this option with the `-s` option. When set to `0`, all signals below the specified scope is dumped.

`-levelstrobe "expression"`	Dumps values when the expression holds true (level sampling). If the expression is not true, the time point and value are not dumped. The expression can consist of one strobe signal (for example, `"a==1"`) or multiple strobe signals (for example, `"a==1 && b==1"`). The `-strobe` and `-levelstrobe` options cannot be used together.
	Example:
	`zwdutils --tool=reportverilog.zwd -levelstrobe "/`
	`system/clock==1 && /system/sig==1"`
	` -s /system/data`
	`/system/addr`

`-log filename`	Specifies the output log file name. The default file name is `err.log`.
`-nolog`	Disables generation of the `zwdreportLog` log directory.
`-nocase`	When included, the mapping of signal names is not case-sensitive. The default is case-sensitive.
`-o reported_file_name`	Specifies the output report file name. If the `-o` option is not specified, the default is `report.txt`.
`-of [b|o|d|u|h]`	Defines the output display format as binary, octal, decimal, unsigned decimal, or hexadecimal. Its default value is `binary`. When this option is specified before the `-s` option, it is applied globally. When this option is specified after a signal, it is applied locally.
`-period period_time`	Dumps values at each specified time.

Feedback

Options	Explanation

`-precision precision_value`	Defines the precision (the number of decimal places to include) of output values for analog signal types. This option is ignored for digital signal types.
	Example:
	Original output without the `-precision` option: `> zwdutils --tool=report test.tr0.zwd -s "I(vcc"`
	`Time(1.0)  I(vcc`
	`==========  ==========`
	`0.000000e+00  -10.0pA`
	`5.000000e-11  23.4nA`
	`1.000000e-10  25.0nA`
	`3.000000e-10  30.4nA`
	`1.100000e-09  41.0nA`
	`2.100000e-09  43.3nA`
	`3.100000e-09  43.4nA`
	- Report with the `-precision` 7 option: `> zwdutils --tool=report test.tr0.zwd -s "I(vcc" -precision 7 -w 20 -o report_with_precision_5.txt`
	`Time(1.0)  I(vcc`
	`==========  ====================`
	`0.000000e+00  -1.0000000e-11A`
	`5.000000e-11  2.3453000e-08A`
	`1.000000e-10  2.5039000e-08A`
	`3.000000e-10  3.0451002e-08A`
	`1.100000e-09  4.1056001e-08A`
	`2.100000e-09  4.3309001e-08A`
	`3.100000e-09  4.3463999e-08A`
	`4.100000e-09  4.3423999e-08A`
	`5.100000e-09  4.3402000e-08A`
	`6.100000e-09  4.3387999e-08A`
	`7.100000e-09  4.3374001e-08A`

`-pt time_precision`	Defines the time precision (the number of decimal places to include) of the output value for analog signal types. This option is ignored for digital signal types. **Example:** `zwdutils --tool=report hspice.zwd -s "/v(_be0" -pt 5 -o 1.txt`

Feedback

Options	Explanation

`-s {signal_name [options]}`	Specifies the signals or scopes to be reported. When specifying a scope name, the wildcard character "*" is appended to the end with double quotes. At least one `-s` must be included.
	Note:
	For system tasks or system functions, when a scope or signal name begins with `'$'` (for example, `$root`), it denotes a variable in the Unix shell, and gives a warning that the variable root value is not obtained. To avoid this error usage, use single quotes (') to replace double quotes ("). For example: `zwdutils --tool=report test.zwd -s '/$root/En_a' -o test_report.txt`
	Note:
	If the output format needs to be specified for multiple signals, the output specification must immediately follow each of the desired signals.
	For example:
	`zwdutils --tool=report test_fsdb.zwd -o multi_scope.txt -s "/U_core_top*" -precision 5 -w 17 "/U_pad_ring*" -precision 6 -w 20 -bt 555 -et 555`
	`-precision 5 -w 17` belongs to the `"/U_core_top*"` signal.
	`-precision 6 -w 20` belongs to the `"/U_pad_ring*"` signal.
	If the following are specified, the `-precision 6 -w 20` only belongs to `/U_pad_ring`:
	`zwdutils --tool=report test_fsdb.zwd -o multi_scope.txt -s "/U_core_top*" "/U_pad_ring*" -precision 6 -w 20 -bt 555 -et 555 -cn 20.`

Options	Explanation
`-shift | -shiftneg time[unit]`	Specifies to `shift (plus)` or `shiftneg (minus)` the report time when the strobe signal matches the specified value. Use this option with the `-strobe` option. **Example:** `zwdutils --tool=report verilog.zwd -shift 10 -s "system/VMA" -strobe "system/clock==1" -o 1`
`-skip_value_no_transition`	If the value of signal has no transition, it is skipped in the output report. **Note:** This option must be specified with `-csv` option.

Feedback

Options	Explanation
`-strobe "expression"`	Reports values when the value of the strobe signal changes to the specified value (edge sampling).
`-verilog | -vhdl`	Specifies the output format as Verilog or VHDL format.
`-w column_width`	Defines the width of the signal column. For the strobe, `level_strobe` or expression signal, if the width is less than the maximum time width, the width is automatically expanded to the maximum time width. **Note:** This option is ignored if the `-csv` option is also specified.

Example

1. Assign the begin time (1000ps) and end time (2000ps) for the report.

```
%zwdutils --tool=report verilog.zwd -s /system/addr -bt 1000ps -et
 2000ps
```

2. Report a slice of a bus signal. For example, if there is the `addr[7:4]` bus, bits 7, 6, 5, and 4 are extracted with the following command:

```
%zwdutils --tool=report verilog.zwd -s "/system/addr[7:4]"
```

3. Report signals in the signal list using different formats.

```
%zwdutils --tool=report zwd/vhdl_typecase.zwd -nocase -s
 top/A_SIMPLE_REC.FIELD3
-a simple.field3 -w 15 TOP/A_COMPLEX_REC.F1.FIELD3 -a
 complex.f1.field3
-w 20 top/a_std_logic_vector -af sean2.alias -of a -o output.txt
-bt 1000 -et 2000
```

   `simple.field3` is reported as the alias for the `top/SIMPLE_REC.FIELD3` signal, `complex.fl.field3` is reported as the alias for the `TOP/COMPLEX_REC.F1.FIELD3` signal, and the alias values in `sean2.alias` are reported for the `top/a_std_logic_vector` signal.

4. Report a scope and its descendants. Multiple scopes may be specified.

```
%zwdutils --tool=report rtl.dump.zwd -bt 10 -et 100 -s
 "/system/i_cpu/*" -level 3 /system/i_pram/clock -cn 0
```

   Up to 3 levels of scopes below `/system/i_cpu` and the `/system/i_pram/clock` signal between 10-100ns are reported. The results are printed line by line.

5. Report the results for the specified strobe point using `-strobe`.

Feedback

```
%zwdutils --tool=report verilog.zwd -strobe "/system/clock==1"
 -s /system/data/system/addr
```

Only when the value of the `/system/clock` signal is 1, the values of the `/system/data` and `/system/addr` signals are reported.

6. Report the results when the expression value changes to true.

```
%zwdutils --tool=report verilog.zwd -exp "/system/addr=='h30
 & /system/clock==1" -s /system/data
```

fsdb2saif Utility on ZWD

Usage

```
zwdutils --tool=saif <zwd_file_name> [Options]
```

Options

Options	Explanation
`-bt time[unit]`	Specifies the begin time of the report. If omitted, the begin time of the ZWD file is used. The time unit is `Ms`, `Ks`, `s`, `ms`, `us`, `ns`, `ps`, or `fs`. The default time unit is `ns`.
`-et time[unit]`	Specifies the end time of the report. If omitted, the end time of the ZWD is used. The time unit is `Ms`, `Ks`, `s`, `ms`, `us`, `ns`, `ps`, or `fs`. The default time unit is `ns`.
`- flatten_genhier`	Eliminates blocks created by generate constructs and appends the block name into the signal name as a prefix.
`-h | -help`	Prints the help message.
`-o reported_file_name`	Specifies the output SAIF file name.
`-s`	Specifies the scopes or sub-scopes to be converted to the output SAIF file. This option cannot be used to specify a signal name and wildcards are not supported.
`-to_stdout`	Redirects the content of the output SAIF file to stdout and the generation of the output SAIF file is disabled even if option `-o` is specified.
`-vhdl`	Converts 9 MVL to 4 states value with the following table instead of skipping VHDL signals. `U X 0 1 Z W L H - (9 MVL) X X 0 1 Z X 0 1 X (4 states)`

Feedback

Examples:

1. Converts ZWD into a backward SAIF format. `%zwdutils --tool=saif verilog.zwd -o verilog.zwd.saif`
2. Converts ZWD into a backward SAIF format with 9 MVL to 4 states conversion for VHDL signals. `%zwdutils --tool=saif verilog_and_vhdl.zwd -vhdl`
3. Converts ZWD into a backward SAIF format with a specified time range.`%zwdutils --tool=saif verilog.zwd -bt 10ps -et 1000ps -o verilog.zwd.saif`
4. Specify the scope name to convert. `%zwdutils --tool=saif rtl.zwd -s "/system/i_cpu" -o rtl.zwd.saif`

fsdbextract Utility on ZWD

Usage

```
zwdutils --tool=extract <zwd_file_name> [Options]
```

Options

Options	Explanation
`-bt time[unit]`	Specifies the begin time of the report. If omitted, the begin time of the ZWD file is used. The time unit is `Ms`, `Ks`, `s`, `ms`, `us`, `ns`, `ps`, or `fs`. The default time unit is `ns`.
`-Compact`	Creates value changes in compact format. This option may reduce file size but takes longer to create the ZWD file.
`-et time[unit]`	Specifies the end time of the report. If omitted, the end time of the ZWD is used. The time unit is `Ms`, `Ks`, `s`, `ms`, `us`, `ns`, `ps`, or `fs`. The default time unit is `ns`.
`-exclude_scope "scope1" ["scope2" … "scopeN"]`	Excludes signals under the specified scopes and sub-scopes. Multiple scopes may be specified.
`-f config_file`	Specifies a text file which defines all the options which are omitted. If the options are used both in the command line and the `config_file`, the command line options override the command in the `config_file`.
`-h | -help`	Prints the help message.
`-level n`	Extracts signals from specified scope and its descendants. Use this option with the `-s` option.

Feedback

Options	Explanation
-log **filename**	Specifies the output log file name. The default file name is err.log.
-memory_will_alloc_in_mega n	Specifies the memory which is allocated to run zwdutils --tool=extract. The memory unit is in mega bytes.
-nolog	Disables generation of the zwdextractLog log directory.
-nocase	When included, the mapping of signal names is not case-sensitive. The default is case-sensitive. Use this option with the -s option.
-o	Specifies the extracted output file name.

Options	Explanation
-s	Specifies the signals or scopes to be extracted. When specifying the signal and scope name, the wildcard characters "*, ?, and %" are allowed. At least one -s option must be included. The design hierarchical delimiter must be used.
-sigOnly	Specifies the signals whose value changes are extracted.
-time_shift time[unit]	Specifies the time offset for the extraction. The specified time can be positive or negative. The time unit is Ms, Ks, s, ms, us, ns, ps, or fs. The default time unit is ns. If the time offset is not specified, the minimum time of the time range is shifted to 0.
-tscale time[unit]	Specifies the time scale unit for the extracted ZWD file. The specified time must be positive. The time unit is Ms, Ks, s, ms, us, ns, ps, or fs. The default time unit is ns.
-top_signals	Extracts signals that do not belong to any scope.
-verilog | -vhdl	Specifies scope is in Verilog or VHDL format. Use this option with the -s option.

Examples:

1. Extract signals.

```
%zwdutils --tool=extract verilog.zwd -s /system/clock -o sig.zwd
```

Feedback

Note:

Surrounding double quotes (") are necessary if any signal contains one of the following characters: '*', '%' or '?".

2. Extract all signals in the specified scopes and their subscopes.

```
%zwdutils --tool=extract verilog.zwd -s /system/i_cpu -level 0 -o
 scope.zwd
```

3. Extract multiple signals and scopes.

```
%zwdutils --tool=extract verilog.zwd
 -s /system/VMA /system/i_cpu/i_ALUB -level 0 -o ss.zwd
```

4. Extract signals and scopes within a specified time range.

```
%zwdutils --tool=extract verilog.zwd -s /system/VMA /system/i_cpu -bt
 100 -et 1000 -o ss.zwd
```

5. Time positive shift

```
%zwdutils --tool=extract verilog.zwd -time_shift 10ns -o result.zwd
Time negative shift
%zwdutils --tool=extract verilog.zwd -time_shift -10ns -o result.zwd
Time shift to zero
%zwdutils --tool=extract verilog.zwd -time_shift -o result.zwd
Time scaling
%zwdutils --tool=extract verilog.zwd -tscale 3.5ns -o result.zwdd
```

Limitations

This feature has the following limitation:

- `zwdutils --tool=extract` utility cannot filter only the scope provided. However, it can extract all the scopes created even if the scope is not specified.

For example:

The following instances are dumped in the `verilog.zwd` ZWD file.

```
/Top/A/s1
/Top/A/B/s2
/Top/A/B/C/s3
/D
```

If `Top.A.s1` instance is specified as follows:

```
%zwdutils --tool=extract verilog.zwd -s Top.A.s1 -level 0 -o ss.zwd
```

The scope B, C, and D are also dumped in the new `ss.zwd` ZWD file.