Verdi® User Guide and Tutorial

Version O-2018.09-SP2, March 2019

SYNOPSYS®

Copyright Notice and Proprietary Information

Third-Party Software Notices

Verdi® Automated Debug Platform includes or is bundled with software licensed to Synopsys under free or open-source licenses. For additional information, see the *third_party_notices.txt* file in the *INSTALL_PATH/doc* directory of the Verdi software installation.

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 Software Licensing Notices

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

Third-Party Links

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

www.synopsys.com

Contents

About This Book

Purpose

This book is designed to allow you to quickly become proficient in the Verdi platform. This manual focuses on the most commonly used commands without going into detail on everything. For detailed descriptions of individual commands, please refer to the appropriate chapter of the *Verdi and Siloti Command Reference Manual*.

The manual should be read from beginning to end, although you may skip any sections with which you are already familiar.

- If you are new to the Verdi platform, begin with the *User Interface*, *Launching Techniques* and various *Tutorials* chapters. After you are familiar with the individual modules, review the *Application Tutorials* chapter.
- If you are familiar with the Verdi platform but want to learn new ways to apply it, review the *Application Tutorials* chapter.

Audience

The audience for this manual includes engineers who are familiar with languages and tools used in design and verification such as Verilog, VHDL, SystemVerilog, simulators, timing analyzers, and transactions. The application of these languages may be for System-on-Chip (SoC), Application Specific Integrated Circuit (ASIC), and Field Programmable Gate Array (FPGA) designs.

This document assumes that you have a basic knowledge of the platform on which your version of Verdi platform runs: Unix or Linux, and that you are knowledgeable in design and verification languages, simulation software, and digital logic design.

Book Organization

The *Verdi User Guide and Tutorial* is organized into the following chapters:

- *About This Book* provides an introduction to this manual and explains how to use it.
- *Introduction* provides an overview of the Verdi platform and introduces its unique debugging tools, capabilities, and methodology.
- *User Interface* provides details regarding the interface, including toolbars, icons, and commands.
- *Before You Begin* provides details on setting up the environment and demo cases.
- *Launching Techniques* provides details on different methods for starting the Verdi platform.
- *nTrace Tutorial* gives step-by-step instructions on *nTrace*.
- *nSchema Tutorial* gives step-by-step instructions on *nSchema*.
- *nWave Tutorial* gives step-by-step instructions on *nWave*.
- *nState Tutorial* gives step-by-step instructions on *nState*.
- *SmartLog Tutorial* gives step-by-step instructions on *SmartLog*.
- *OneSearch Tutorial* gives step-by-step instructions on *OneSearch*.
- *Temporal Flow View Tutorial* gives step-by-step instructions on *nTrace*.
- *Debug a Design with Simulation Results Tutorial* ties together all the modules in a simple debug scenario.
- *Appendix A: Supported Waveform Formats* lists the supported waveform formats.
- *Appendix B: Supported FSM Coding Styles* lists the supported finite state machine (FSM) coding styles.
- *Appendix C: Enhanced RTL Extraction* describes instance array, for loop statements, and creating detailed extracted schematics.
- *Appendix D: Additional Transaction Example* includes additional information for generating and extracting transactions.

Conventions Used in This Book

The following conventions are used in this book:

- *Italic font* is used for module names, emphasis, book titles, section names, application names, and design names within paragraphs.
- **Bold** is used to emphasize text and highlight titles, menu items, and other Verdi terms.
- `Courier type` is used for program listings and text messages that the Verdi platform displays on the screen. You can also use for file paths, and file names.
- **NOTE** describes important information, warnings, or unique commands.
- **Menu -> Option** identifies the path used to select a menu command.
- Left-click or Click means click the left mouse button on the indicated item.
- Middle-click means click the middle mouse button on the indicated item.
- Right-click means click the right mouse button on the indicated item.
- Double-click means click twice consecutively with the left mouse button.
- Shift-left-click means press and hold the <Shift> key then click the left mouse button on the indicated item.
- Drag-left means press and hold the left mouse button, then move the pointer to the destination, and release the button.
- Drag means press and hold the middle mouse button on the indicated item then move and drop the item to the other window.

Related Publications

- *Installation & System Administration Guide* - explains how to install the Verdi and Siloti systems.
- *Verdi and Siloti Command Reference Manual* - gives detailed information on the Verdi and Siloti command sets.
- *Verdi and Siloti Quick Reference Guide* - provides a quick reference for using the Verdi and Siloti systems with typical debug scenarios.
- *Linking Novas Files with Simulators and Enabling FSDB Dumping* - gives detailed information on linking Novas object files with supported simulators for FSDB dumping and the related dumping commands.
- *nAnalyzer User Guide and Tutorial* - detailed information on using the *nAnalyzer* Design Analysis module.
- *nECO User Guide and Tutorial* - detailed information on using the *nECO* Automated Netlist Modification module.
- *Release Notes* - for current information about the latest software version. Refer to the *View release notes* link on the product downloads page.
- Language Documentation

  Hardware description (Verilog, VHDL, SystemVerilog, and so on) and verification language reference materials are not included in this manual. For language related documents, refer to the appropriate language standards board (www.ieee.org, www.accellera.org) or vendor (www.synopsys.com, www.cadence.com) websites.

Introduction

Overview

The Verdi® Automated Debug Platform is an advanced solution for debugging your digital designs that increases design productivity with complex System-on-Chip (SoC), ASIC, and FPGA designs. Traditional debug tools rely on structural information alone and the engineer's ability to infer the design behavior from its structure. The Verdi platform provides powerful technology to help you comprehend complex and unfamiliar design behavior, automate difficult and tedious debug processes, unify diverse and complicated design environments, and infer the dynamic behavior of a design over time.

In addition to the standard features of a source code browser, schematics, waveforms, state machine diagrams, and waveform comparison (for comparing simulation results in FSDB format), the Verdi platform includes advanced features for automatic tracing of signal activity using temporal flow views, assertion-based debug, power-aware debug, and debug and analysis of transaction and message data. All of this is available in a graphical user interface using the Qt platform that supports multiwindow docking and is easily customizable.

The Verdi platform enable engineers to locate, isolate, understand, and resolve bugs in a fraction of the time of traditional solutions. This maximizes the efficiency and productivity of expensive engineering resources, significantly reduces costs, and dramatically accelerates the process of getting silicon to market.

Technology Overview

A technology base has been constructed that is optimized for design exploration, understanding, and debugging. The Verdi platform's unique architecture features powerful compilers, interfaces, databases, analysis engines and visualization tools in an integrated system for complete debugging.

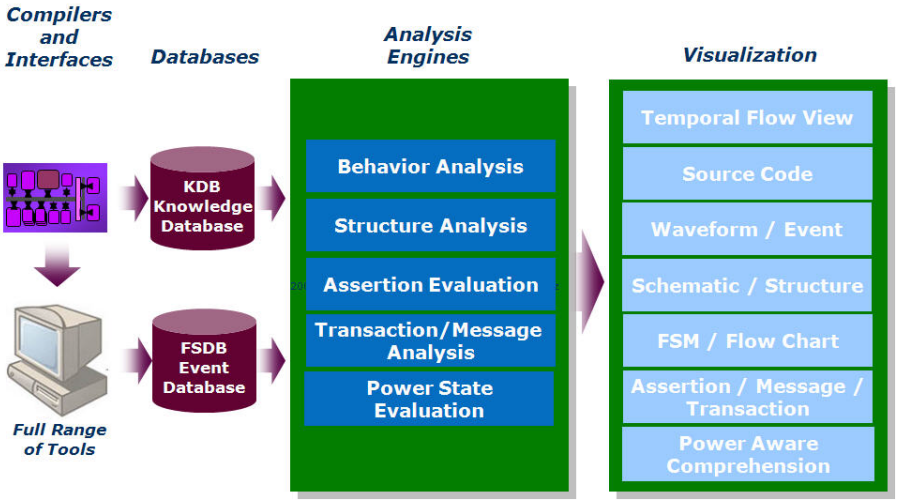


Figure: Verdi Technology Overview

Compilers, Interfaces and Interoperability

The Verdi platform has compilers for the most common design/verification languages and provides several interfaces for standard simulators.

- **Compilers**: The Verdi platform provides compilers for the languages used in most design and verification environments, such as Verilog, VHDL and SystemVerilog (both design and verification code) and power code (CPF or UPF). As the code is analyzed and compiled, it is checked for syntax and semantic errors.
- **Interfaces**: The Verdi platform's readers import industry-standard VCD and SDF data from all simulators and timing tools. The results are read in from the detection tool and stored in the Fast Signal Database (FSDB). Direct dumping to FSDB through the object files linked to a verification tool (simulator) results in smaller waveform files and flexible access to post-simulation data.
- **Interoperability**: The Verdi platform's comprehensive, documented, and supported interfaces provide interoperability with all popular logic

simulators, as well as many formal verification and timing analysis applications. These interfaces also provide the ability to integrate other verification applications using Tcl and C-language application programming interfaces (APIs). Synopsys has partnered with dozens of design and verification companies to integrate their tools with the Verdi platform, which saves the time and expense of learning multiple interfaces by providing a consistent view throughout the entire verification and debug flow.

Databases

The Verdi platform provides two databases. All analysis engines and visualization tools use these databases.

- **Knowledge Database (KDB)**: As it compiles the design, the Verdi platform uses its internal synthesis technology to recognize and extract specific structural, logical, and functional information about the design and stores the resulting detailed design information in the KDB.
- **Fast Signal Database (FSDB)**: The FSDB stores the simulation results, including transaction data and logged messages from SVTB or other applicable languages, in an efficient and compact format that allows data to be accessed quickly. Synopsys provides the object files that can be linked to common simulators to store the simulation results in FSDB format directly. You can generate FSDB either from the provided routines or after reading and converting your VCD file. In addition, FSDB read/write API routines are provided for customers and partners to use.

Analysis Engines

Using the information from the KDB and FSDB, the Verdi platform provides a set of analysis engines for different applications, including:

- **Structure Analysis**: analyze design structure to show how components are connected.
- **Behavior Analysis**: analyze design and simulation results to display design operation over time.
- **Assertion Evaluation**: answer questions and search for details about design operation from a previous simulation.
- **Transaction/Message Analysis**: analyze transaction and message (log) data in the FSDB file and visualize in *nWave* and a spreadsheet view.

- **Power State Evaluation**: evaluate the power state based on the power intent description in the CPF/UPF and the values of related signals in the FSDB file.

Graphical User Interface

The graphical user interface uses the Qt platform and provides the following functions:

- A *Welcome* page summarizing the available resources in a single location.
- History support enabling easy restoration of previous sessions.
- Typical work modes with predefined window layouts making the debug content easy to locate.
- A unique Spotlight function searches for a command without exhaustively searching through all the drop-down menus.
- Several customization options:
  - System frames and toolbar icons can be undocked, moved to a new location, and then docked again.
  - A pane can be maximized by double-clicking the pane banner so the content is more visible. Shrinking to the original size is another double-click.
  - The visible toolbar icons can be selected through a menu option.
  - Bind key values and drop-down menu names and locations can be customized through a provided customization form.

Visualization

The Verdi platform provides unparalleled temporal visualization capabilities in the form of the *Temporal Flow View*. This revolutionary tool extracts and displays multicycle temporal behavior from the design data and simulation results.

In addition, the Verdi platform includes state-of-the-art structure visualization and analysis tools: *nTrace* for source code, *nWave* for waveforms, *nSchema* for schematic/ logic diagrams, and *nState* for finite state machines (FSMs). These tools focus on analyzing the structure of the design in the form of the signal relationships in the RTL, physical connections in schematic/logic diagrams, states and transitions in FSM bubble diagrams, and value changes in waveforms.

The *Property Tools* window in the Verdi platform provides integrated support for assertions and enables quick traversal from an assertion failure to the related

design activity. While the *Transaction/Message Analyzer* enables debug and analysis at higher levels of abstraction from transaction or log information saved to the FSDB file. The *Power Manager* window provides visualization of the power intent and supports cross-probing with other Verdi platform windows.

All of these views are fully integrated. For example, you can select any portion of the design source code and instantly generate corresponding hierarchical or flattened logic diagrams. You can rapidly explore a design and its verification results by clicking on context-sensitive hyperlinked objects and signals in any of the views. You can quickly and easily change the current view to locate and isolate the specific information necessary to understand any portion of the design and resolve any problems.

User Interface

Overview

The Verdi® Automated Debug Platform has a highly customizable graphical user interface with a contemporary look. The following figures illustrate the look of the Verdi platform.

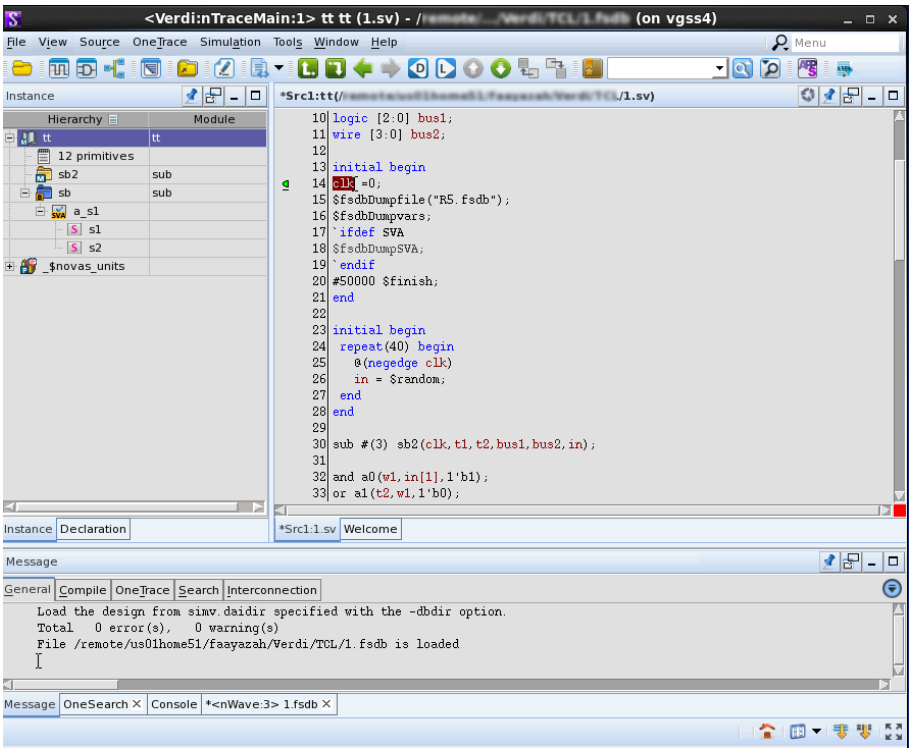


Figure: Verdi Main Window

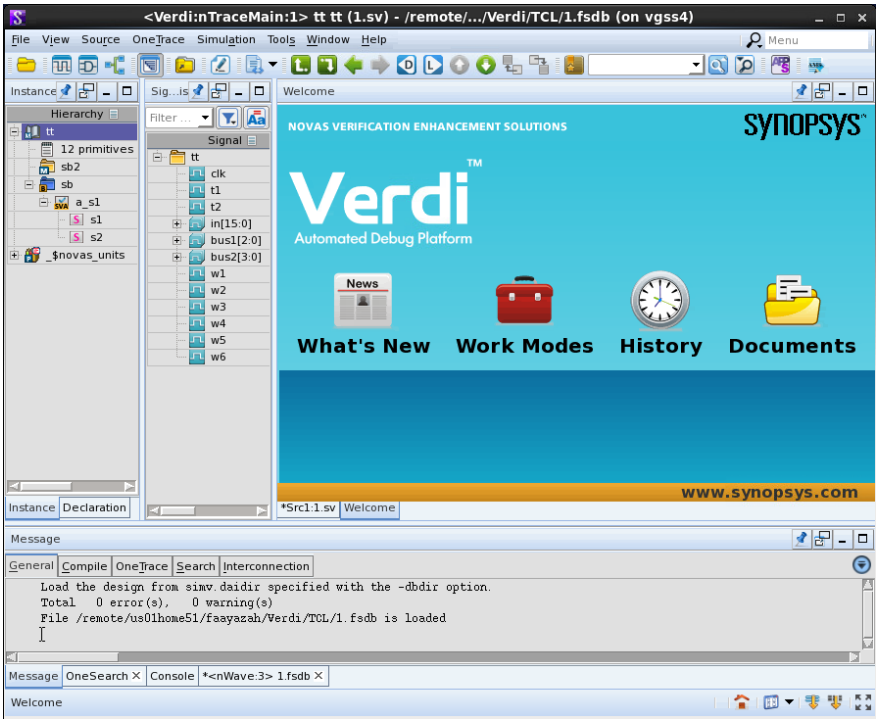


Figure: Verdi Window With Welcome Page

The Verdi platform has a large number of commands, including many that are invoked through mouse clicks or drags rather than selecting from pull-down menus at the top of each window. Read this chapter to become familiar with the interface conventions of the Verdi platform before proceeding further.

This chapter covers the following topics:

- *Common User Interface Features*
- *nTrace User Interface*
- *nWave User Interface*
- *nSchema User Interface*
- *nState User Interface*
- *Flow View User Interface*
- *Transaction/Message User Interface*
- *nCompare User Interface*
- *nECO User Interface*
- *nAnalyzer User Interface*

Common User Interface Features

The features described below are common to *nWave*, *nTrace*, *nSchema*, *nState*, and *Flow View* components. Refer to the *User Interface Overview* section of the *Introduction* chapter in the *Verdi and Siloti Command Reference Manual* for additional information.

Frame Banner

The banner at the top of each pane identifies the application, frame number (such as *<nWave:2>*), and file, unit, or scope displayed in that pane. The asterisk (*) character appearing at the front of the banner indicates that the pane is the active one.

Double-click the pane banner bar to maximize a pane or to shrink the pane back to the previous size as shown in the following figure.

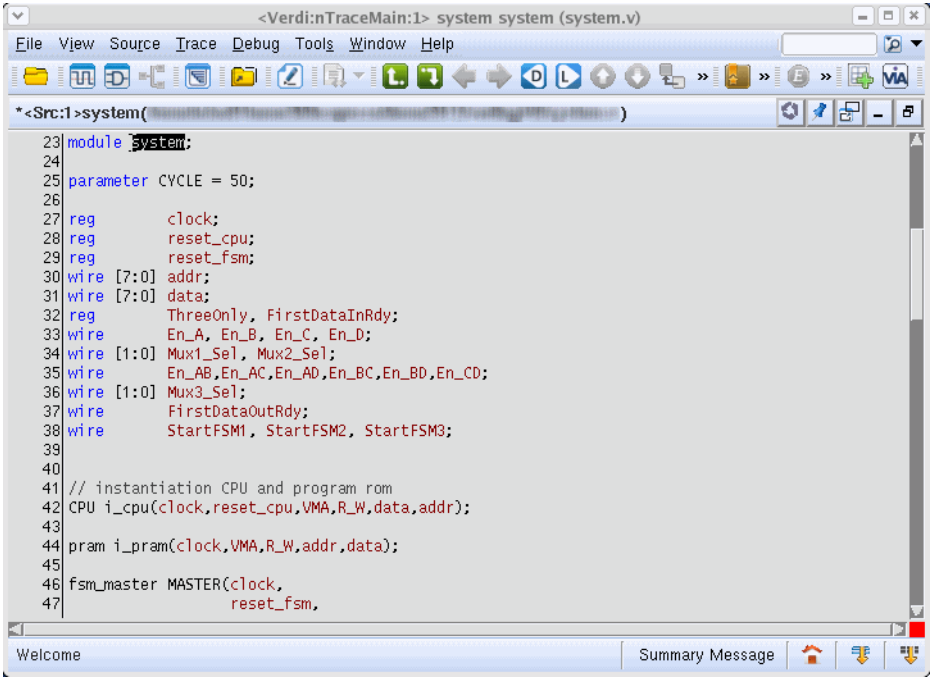


Figure: Maximize the Source Code Pane

Right-click the banner of a dockable pane to display a configuration option menu that lists all the available dockable panes and toolbar categories. Toggle the

option to hide/show the entire pull-down menu, dockable pane, or toolbar category.

Figure: Configuration Option Menu to Hide/Show Dock Panes/Toolbar Categories

Pull-Down Menu Commands

A pull-down menu bar is located just below the banner for panes that are also windows. Each menu item contains several commands that display when the menu is selected. The pull-down menu can be hidden or displayed by selecting the right-click **Menu** option invoked from the window banner, menu bar, or toolbar.

When the **Menu** option is toggled *off*, the pull-down menu of the pane or window is hidden. You can press the **Alt** key within the pane or window to display or hide the pull-down menu again. Also, when the cursor is clicked elsewhere in the pane or window, the menu is automatically hidden again.

When the **Menu** option is toggled *on* (the value *on* means always show the pull-down menu), the pull-down menu cannot be shown/hidden by pressing the **Alt** key.

For each sub-window or window in the Verdi platform, custom commands can be added using the **Tools -> Customize Menu/Toolbar** command.

Mnemonic Keys

The pull-down menus support **Meta** key invocation using mnemonics. The mnemonic for each item is indicated by an underline. For example, to display the **File -> Open** menu (meta -fo), press and hold the <Meta> key on your keyboard (the diamond key/<Alt> key on Sun keyboards or the <Alt> key on Windows' keyboards) and press the "f" key, then release the <Meta> key and press the letter "o" key.

Bind Keys

A command can be bound to either a keystroke or a mouse button. After the bind keys are defined, commands can be invoked with a keystroke or mouse click. For example, the **Source -> Active Annotation** command can be invoked using the "X" key (the defined bind key is the letter after the command in the menu). The bind keys of the menu commands can be customized using the **Tools -> Customize Menu/Toolbar** command.

Toolbars

A row of icons appears beneath the pull-down menu bar on panes that are also windows. These icons provide access to frequently used commands for the current window.

The available toolbar icons may be modified using one of the following methods:

- Enable/disable the icon category using the main window right-click option menu.
- Left-click to select the separator bar and then drag left or right to decrease or increase the associated space. When the space is decreased such that some icons are no longer visible, a double arrow (>>) symbol is displayed to the right of the category. Clicking this symbol displays the hidden icons.
- Left-click to select the gray bar and then drag up or down to undock the category and then move to a new location on the toolbar or to left/right/ below the window. Available slots are highlighted with a blue dashed line.
- Define/modify/add toolbar icons and categories using the **Tools -> Customize Menu/Toolbar** command. Refer to the *nTrace* chapter of the *Verdi and Siloti Command Reference Manual* for details.

Mouse Operation

The mouse is most often used to select objects by clicking the left mouse button. A range of objects can be selected by dragging with the left mouse button over the objects or by using the <Shift> key along with left-clicking. To add or remove individual objects to or from the selection, use the <Ctrl> key along with the left-click.

The Verdi platform also makes use of drag-and-drop to move information from one pane/window to another. Normally drag-and-drop is performed by pressing the middle mouse button or left mouse button to select the object, holding the button as the mouse is moved to a new location, and then releasing the button to "drop" the object into a new location.

The drag-and-drop operation can be performed between different pane types, for example, dragging a signal from the *nWave* pane and dropping it to the source code pane executes tracing connectivity of the selected signal. If the pane is in the background (displayed as a tab), moving the dragged object to the tab name and dropping it changes the tab to the foreground and drops the object. The resulting behavior is the same as if the object was dropped directly in the pane.

Right Mouse Button Menus

Right-click an object to display a menu with commands appropriate for that object type. These menus are described in detail in the *Right-Click Commands* sections of the *Verdi and Siloti Command Reference Manual*.

Undock/Dock

The main window of the Verdi platform consists of dockable panes that can be released (undocked) from the main window. The dockable panes can be docked to the main window again.

Every dockable pane has its own banner or title bar. The dockable panes can be moved from one dock area to another by dragging the pane banner. A dockable pane can attach above, below, left, right or over another dockable pane. A tab is created when you dock a pane over another dockable pane. Refer to the following figures for examples.

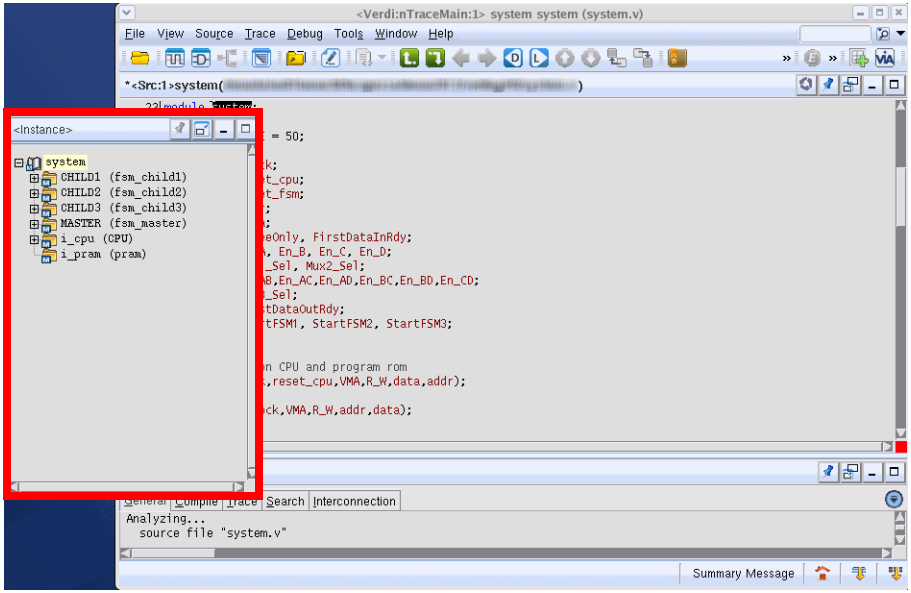


Figure: Undock Design Browser Pane from Main Window

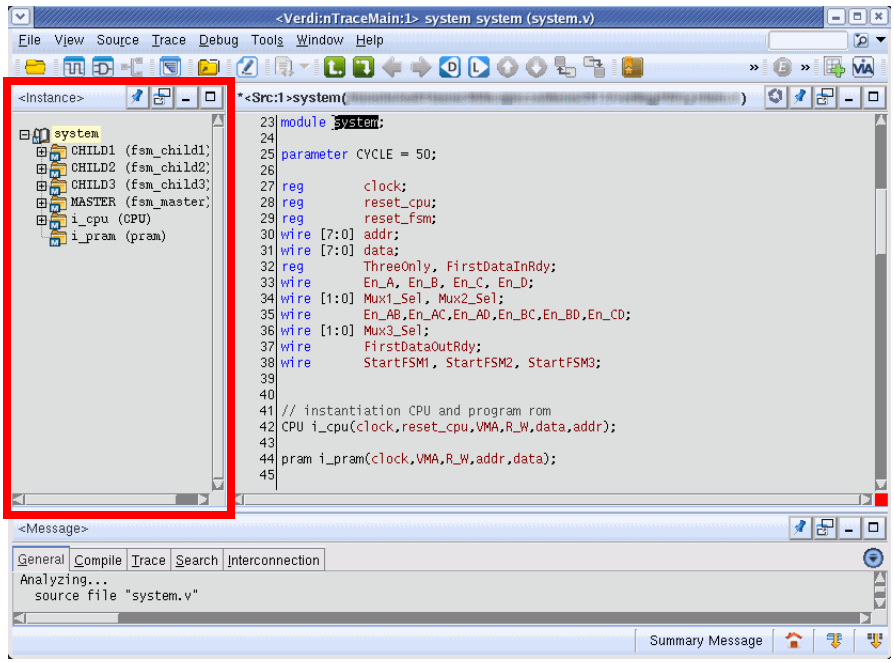


Figure: Re-Dock Design Browser Pane to Main Window

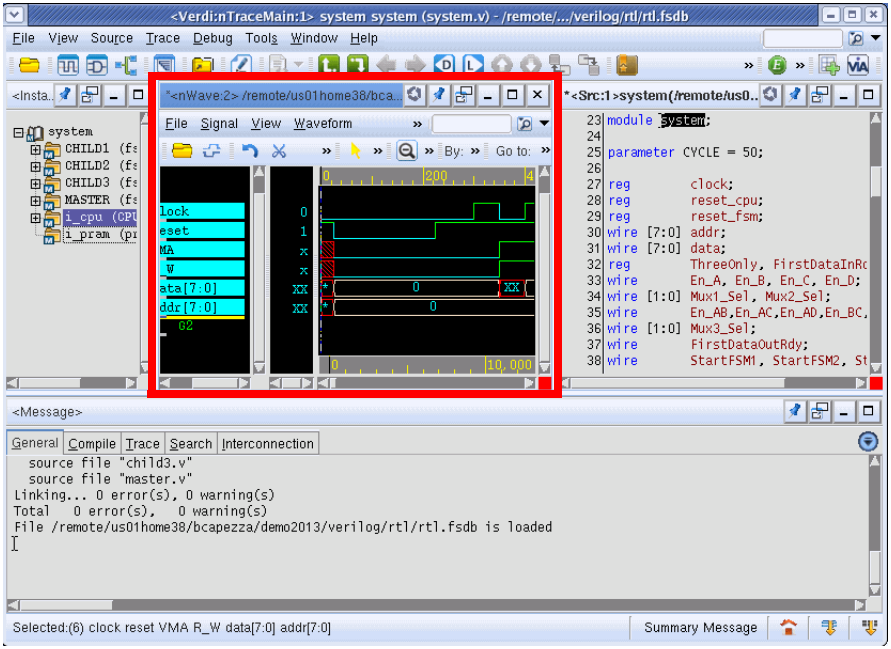


Figure: Dock nWave to the Right of the Design Browser Pane

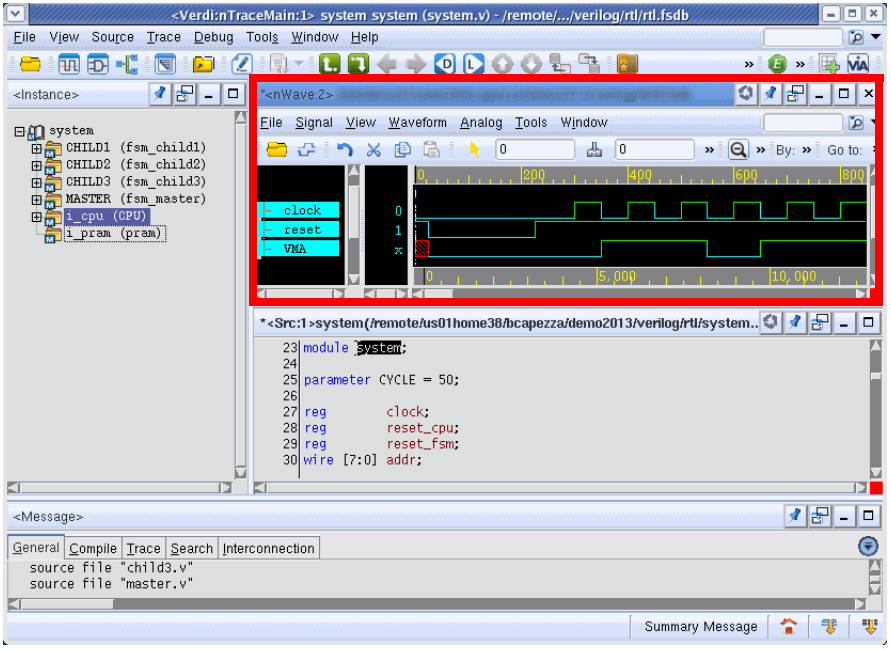


Figure: Dock nWave above the Design Source Code Pane

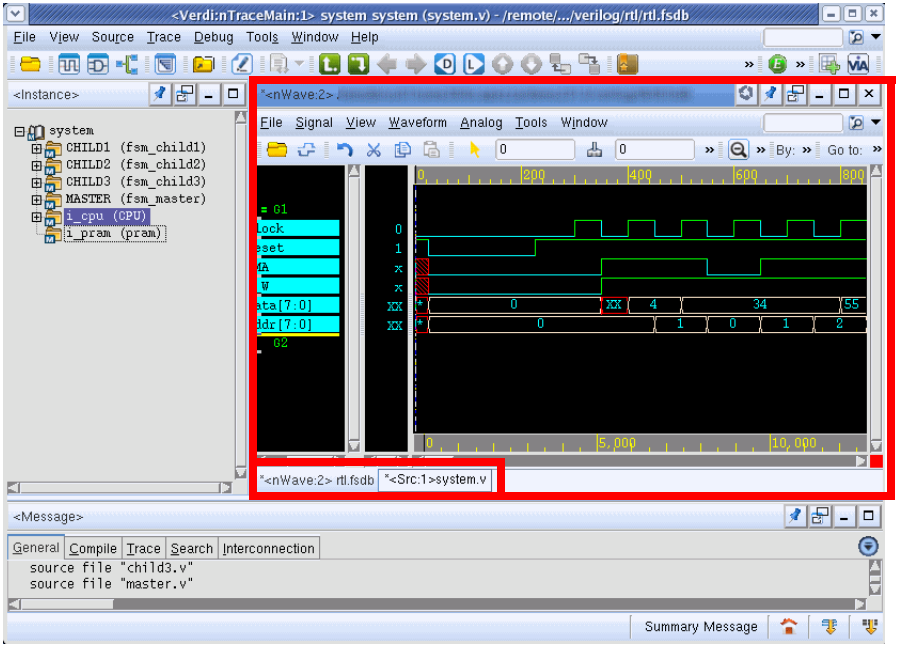


Figure: Dock nWave over the Design Source Code Pane to become a Tab

A pane can also be docked/undocked by clicking the **Dock/Undock** icons on the pane banner. Some major dockable panes, like *nWave* and *nSchema*, can be released to become stand-alone windows. Other panes (such as, *message* and *source code* panes) that belong to the *nTrace* main window can also be released to become widgets.

Refer to the *Icons for User Interface Overview* section in the *Introduction* chapter of the *Verdi and Siloti Command Reference Manual* for more information.

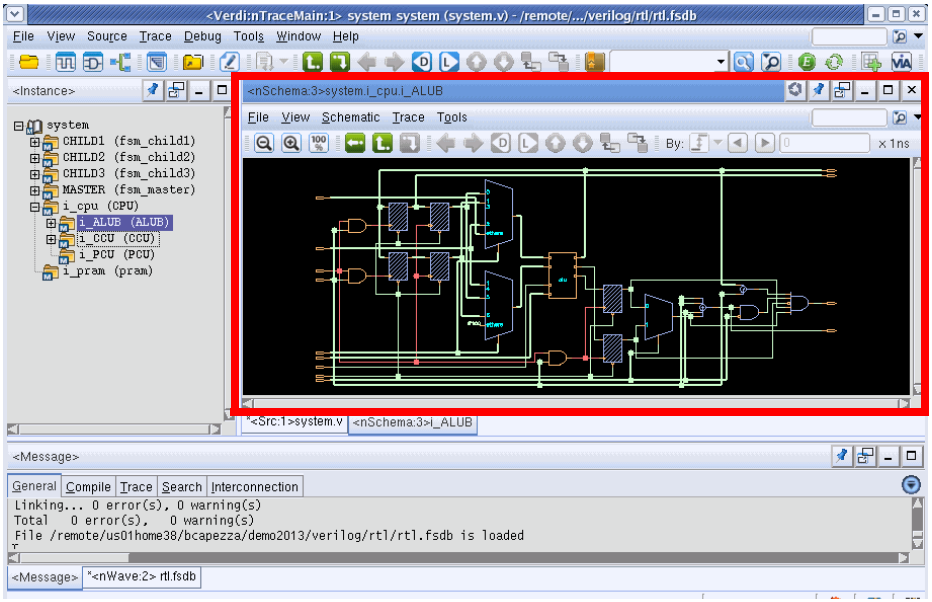


Figure: nSchema Docked as a Pane

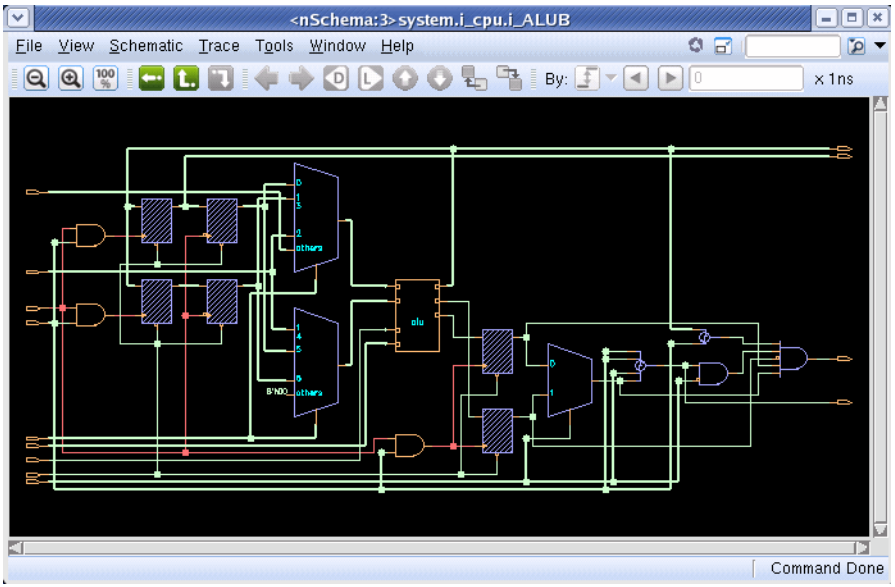


Figure: nSchema Undocked as a Window

Right-click on any pane banner to display a configuration option menu that lists all the available dockable panes and toolbar categories. Toggle the option to hide or show the entire pull-down menu, any dockable pane, or toolbar category.

The layout of the main framework can be saved or restored by invoking the **Window -> Save/Restore User Layout** command. To switch to the previous or next layout, invoke **Window -> Previous Layout** or **Next Layout** commands respectively.

Refer to the *Window/Frame Right-Click Options* sections of the *Verdi and Siloti Command Reference Manual* for details.

On-line Help

The *nTrace* main window and the stand-alone *nWave*/*nSchema* windows provide online help, which can be accessed through the **Help** menu.

nTrace User Interface

When you start the Verdi platform, the *nTrace* main window displays and serves as the main window from which other frames/windows are created. When you import a new design into the *nTrace* main window (using the **File -> Import Design** command), the Verdi platform closes existing *nWave* and *nSchema* panes/windows started from the open session.

The *nTrace* main window contains three re-sizable panes:

- Design browser pane
- Source code pane
- Message pane

An example *nTrace* main window is shown below:

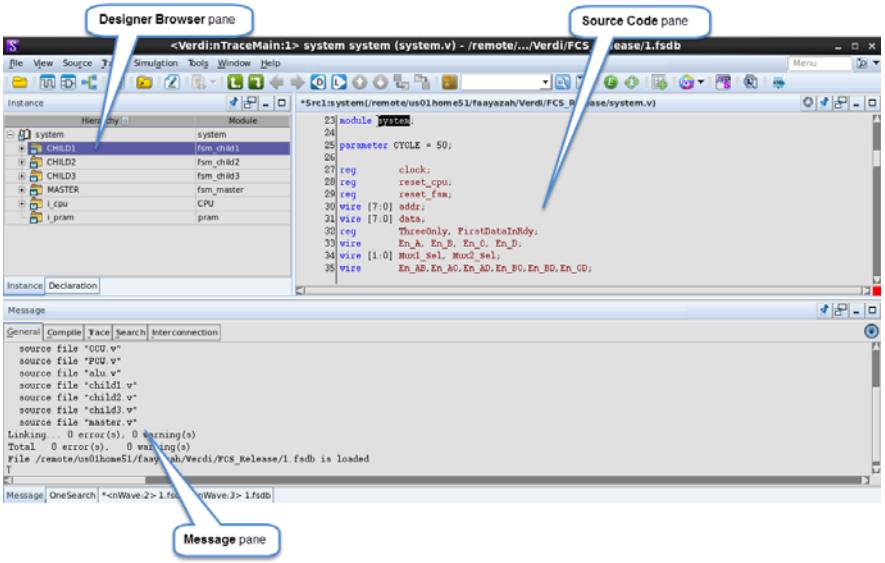


Figure: nTrace Example Window

When you open a design with the Verdi platform, the HDL source code of the top-level unit is displayed in the *Source Code* pane.

The top-level unit is shown as the root of the design hierarchy in the design browser (refer to the *nTrace Design Browser Pane* section below).

The message pane reports errors or other information related to the Verdi platform's operation.

nTrace Design Browser Pane

Located on the left side of the *nTrace* main window, the *Design Browser* pane displays the design hierarchy and provides a way to navigate through the hierarchy (see *nTrace Example Window* figure above).

The *Instance* tab in the *nTrace Design Browser pane* consists of the following two columns:

- Hierarchy
- Module

The **Hierarchy** column can be sorted in ascending, descending, or type order by clicking the heading of the column. By default, the columns are sorted in the type order.

The **Module** column can be sorted in ascending or descending order by clicking the heading of the column.

You can also filter and search the *Instance* tab for a specific node. See *Design Browser Frame Right-Click Options* section in the *Verdi and Siloti Command Reference Guide* for more information.

This window contains the following icons and symbols:

Symbol	Name	Description
 	Plus Minus	Click these symbols to either expand (plus) or collapse (minus) the display of the selected unit's hierarchy.
	Opened-folder	Indicates that the relevant design scope is active and the related source code is displayed in the Source Code frame. The letter on the folder is a mnemonic for the scope type, such as M for module, L for library, T for task, and F for function.
	Closed-folder	Indicates the relevant design scope is non-active. As in the **Opened-folder** symbol, the letter on the closed-folder symbol indicates the type of design scope.
	Icon with a bookmark	Indicates the relevant design scope is set with a bookmark.

Symbol	Name	Description
	Highlighted Design Scope	The highlighted design scope is selected and acting as the target scope for further relevant operations in the design browser.

nTrace Source Code Pane

The source code pane appears on the right side of the *nTrace* main window. Multiple source code files can be displayed as multiple tabs. If a module is described in multiple source files, multiple tabs are opened to display the complete source code when the node is set as an active scope. Each tab is undockable.

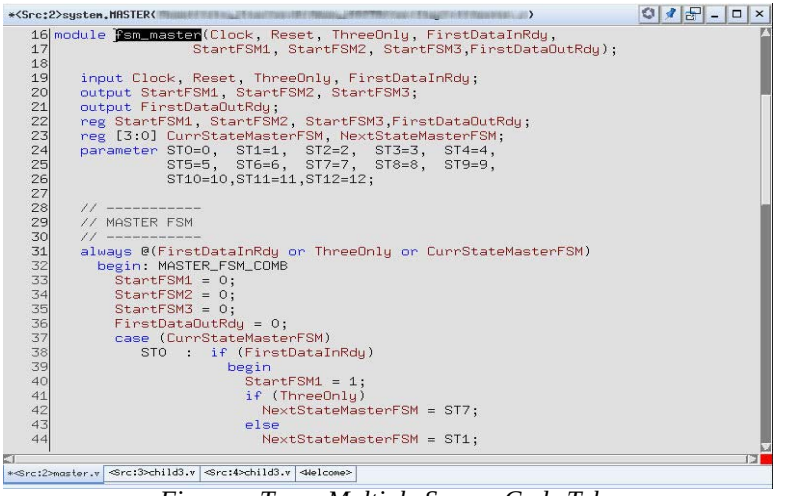


Figure: nTrace Multiple Source Code Tabs

The source code view displays the source code for the active unit in the design browser. This window is divided into the following two areas:

- Source Code Area
- Indicator Area

Source Code Area

The source code area contains the HDL source code. The Verdi platform color-codes the source code to differentiate syntax elements. You can set the syntax colors to your preferences. Some colors change during debugging. For example, signals that are traced are displayed in green to highlight the trace history. You can reset all the traced signals' colors to their default settings using the **Trace -> Reset Traced Signal's Color** command.

Indicator Area

The indicator area contains line numbers and graphical indicators that result from load tracing, driver tracing, connectivity tracing, and bookmarking. The following table lists and describes the symbols used in the indicator area:

Symbol	Definition
	**Driver** - result from last trace command. Multiple drivers are possible.
	**Active driver** - selected driver from last active trace command or current **Show** command.
	**Load** - result from last trace command. Multiple loads are possible.
	**Possible driver** - result from the possible analyzed traced result.
	**Unanalyzed driver** - result from the unanalyzed traced result.
	**Bookmark** - marks a selection for easy referral.
	Pass through load - result from the pass-through driver traced results.
	Pass through driver - result from the pass-through load traced results.
	Pass through connectivity - result from the pass-through connectivity traced results.

The indicator area also shows interactive simulation controls such as break points and current active statement arrows.

nTrace Message Pane

The message pane at the bottom of the *nTrace* main window contains *General*, *Compile*, *Trace*, *Search,* and *Interconnection* tabs.

You can drag-and-drop a signal from the *Source Code* pane to the *Trace* or the *Search* tab to list the results of **Trace Driver** or the search results respectively.

A *Find* field appears above the message tabs when the **Find** toolbar icon is clicked. Refer to the *Message Frame* section in the *nTrace* chapter of the *Verdi and Siloti Command Reference Manual* for details.

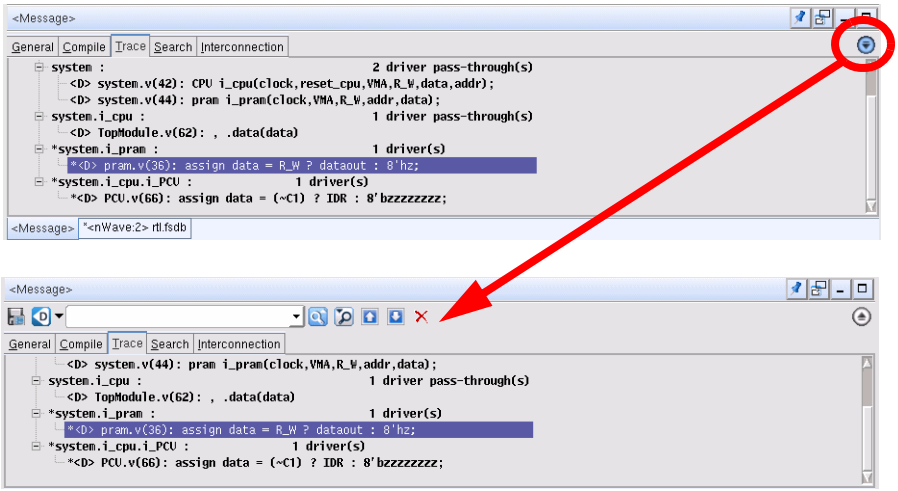


Figure: nTrace Find Bar on the Message Frame

nTrace Toolbar Icons

Refer to the *Toolbar Icons and Fields* section in the *nTrace* chapter of the *Verdi and Siloti Command Reference Manual* for information regarding available toolbar icons.

NOTE: The default toolbar can be modified through the **Tools -> Customize Menu/Toolbar** command.

---

nWave User Interface

You can open a new *nWave* pane from the *nTrace* main window by clicking the **New Waveform** icon or choosing the **Tools -> New Waveform** command. An *nWave* pane can be released from the main window to become a stand-alone window by clicking the **Undock** toolbar icon. An example *nWave* stand-alone window is shown below.

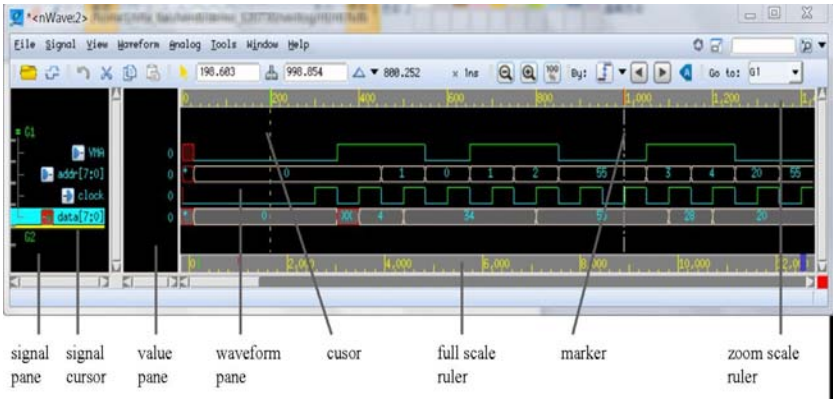


Figure: Example nWave Standalone Window

An example docked *nWave* pane is shown below.

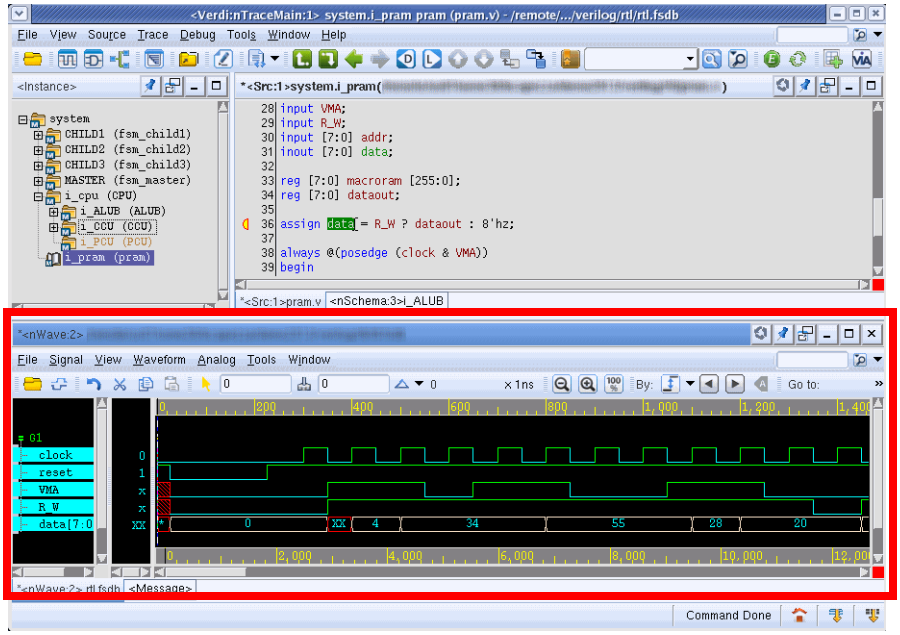


Figure: Example nWave Dock Pane

The *nWave* pane/stand-alone window consists of three re-sizable sub-windows (also known as panes):

- Signal pane
- Value pane
- Waveform pane

nWave Signal Pane

The *Signal* pane displays signals and group names on the left side of the *nWave* display. You can use the *Signal* pane to select and manipulate signals and groups of signals. Three types of objects appear in the *Signal* pane:

- Signal name
- Signal cursor
- Group name

Signal Name

A signal name appears to the left of its waveform. In addition to identifying the waveforms, the signal names are selectable areas; clicking on a signal name selects that signal for manipulation. The signal name can be displayed as either

a full hierarchical name or a local name. By default, *nWave* right-justifies the signal name. However, you can change the justification. If a name is too long, use the horizontal scroll bar or adjust the window size to see the entire name.

Signal Cursor

The signal cursor marks the insertion point for signal commands: **Add**, **Move**, **Paste**, **Overlay Signals**, and **Create Bus**. Middle-click to set the signal cursor.

Group Name

You can place similar signals in the same group. The group name can be changed from the default of G1, G2, and so on.

nWave Value Pane

The *Value* pane is next to the *Signal* pane and displays the value of each signal at the cursor time in the *Waveform* pane. You can select the display format for signals. For example, they can be displayed as hex, octal, binary, decimal value, or user-defined alias text.

Preferences for what is displayed (for example, leading zeros, marker value), can be set through the *Value* pane menu or the **Tools -> Preferences** command.

For any value change of a signal, *nWave* displays the old value to the new value in the *Value* pane indicating that the value is changed from 0 to 1 or 1 to 0. If the value (such as the value change of the long bus value) is not fully visible due to the width of the *Value* pane, move the cursor on top of that value in the *Value* pane, and the value is displayed in the tip window.

nWave Waveform Pane

The *Waveform* pane appears to the right and displays the waveforms. In addition, the *Waveform* pane contains the following objects:

- Cursor
- Marker
- Zoom scale ruler
- Full scale ruler

Cursor

The cursor is used to show the current simulation time for all windows and to provide one end point for delta time calculations. To set the cursor, left-click the moues button. The toolbar displays the cursor time.

Note the following when setting the cursor:

- The setting affects the time display (and, therefore, the results) in all panes/windows that display values.
- If you click inside the *Waveform* pane and choose the **Waveform -> Snap Cursor to Transitions** command (“s” key), the cursor can only be set where there is a signal transition.
- If you de-select the **Waveform -> Snap Cursor to Transition** command, you can set the cursor to any location.

Marker

The marker is used to provide the second point of a delta calculation. To set the marker, click the middle mouse button. The toolbar displays the amount of time between the cursor and the marker (the delta time).

Note the following when setting the marker:

- If you click inside the *Waveform* pane and choose the **Waveform -> Snap Cursor to Transitions** command (“s” key), the marker can only be set where there is a signal transition.
- If you de-select the **Waveform -> Snap Cursor to Transition** command, you can set the marker to any location.
- If you choose the **Waveform -> Fix Cursor/Marker Delta Time** command (“x” key), cursor or marker is spaced at the same delta time.
- If you de-select the **Waveform -> Fix Cursor/Marker Delta Time** command, cursor or marker is not spaced at the same delta time.

NOTE: After you set the cursor time and the marker time, right-click to zoom and fit the waveform display to the time range between the cursor and the marker times.

Zoom-Scale Ruler

The zoom-scale ruler appears at the top of the *Waveform* pane and displays the current displayed time range.

Full-Scale Ruler

The full-scale ruler appears at the bottom of the *Waveform* pane. This ruler displays the time range of all the results (not just the displayed portion) and indicates where the cursor and marker positions are in this range. You can change cursor and marker times by clicking on the full-scale ruler. Selecting a range (dragging with the left mouse button) zooms the display so that the selected area zooms the display in the *Waveform* pane to the selected area.

nWave Toolbar Icons

Refer to the *Toolbar Icons and Fields* section in the *nWave* chapter of the *Verdi and Siloti Command Reference Manual* for information regarding available toolbar icons.

NOTE: The default toolbar can be modified through the **Tools -> Customize Menu/Toolbar** command.

Get Signals

The *nWave* window does not display any signals by default; signals are added by dragging from other windows or by selection in the *Get Signals* form (using the **Signal -> Get Signals** command). Signals are displayed hierarchically based on the design unit selected in the design hierarchy box on the left side of the form.

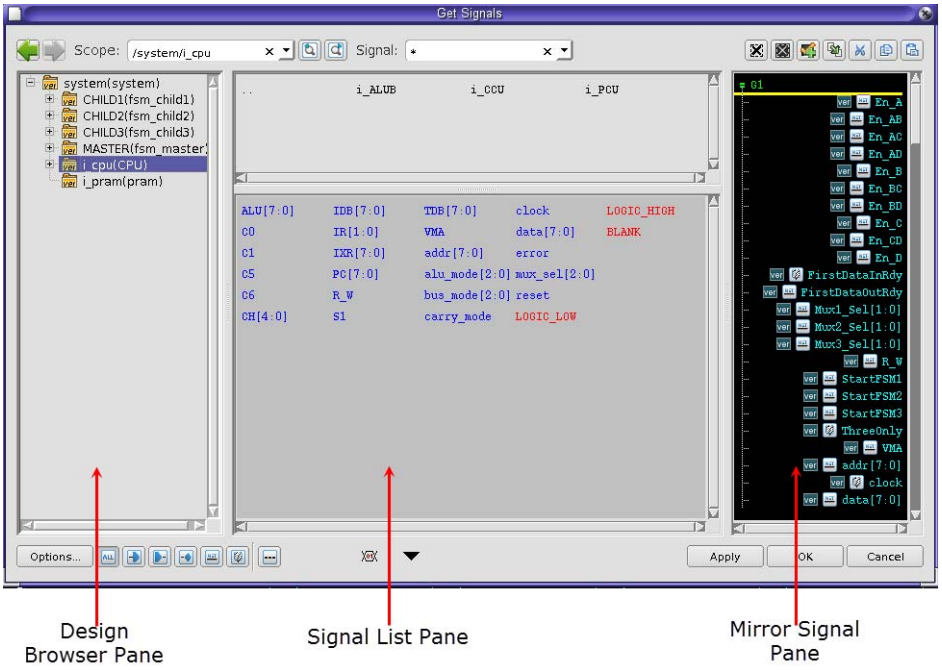


Figure: nWave Get Signal Form

To select signals, navigate the design tree in the *Design Hierarchy* pane to find the desired signals, then either drag them to the mirror signal pane in the right-side pane (which mirrors the signals in the *Waveform* pane) or select the signals and click **Apply**. When you have selected all of the signals of interest, click **OK**.

NOTE: The design hierarchy of the simulation files may not match that of the currently opened design source. You can display waveform data independent of the design that is loaded.

The mirror signal pane allows you to manipulate the arrangement of the signals displayed in the *Get Signals* form without immediately affecting the waveform pane. After finishing the signal arrangement, click **Apply** to synchronize the waveform pane. Click **OK** to apply the arrangement and close the form.

Refer to the **Get Signals** command description in the *nWave* chapter of the *Verdi and Siloti Command Reference Manual* for details.

nWave Mouse Operations

The following tables list the *nWave* mouse actions:

Mouse Action	nWave - Signal Pane
Left-click	De-select the current selected signals/group and select the signal under the mouse button.
Left-click drag-and-drop	Select the object first and then drag-and-drop the object to the new location using the left mouse button on the indicated item.
Left-click the plus/minus icon of a group containing signals	Unhide/hide the signals of the selected group.
Left-click a Group	De-select the current selected signals/group and select the group under the mouse button.
Middle-click	Set the Signal Cursor position for the destination of commands: Move, Paste, Add, Overlap and Create Bus.
Right-click	Open a context-sensitive menu that provides some commands, which apply to the signal/group under the mouse button.
Double-click a bus	Expand or collapse bus member.
Double-click a power domain signal	Expand to three member signals to display value changes for power state, power nominal (for CPF; power nominal will be power alias when a UPF file is loaded), and power voltage.
Double-click an interface sub-group	Expand or collapse the node.
Drag & Drop	Move the selected signals to the Signal cursor position. **NOTE**: The Signal Cursor position is moved along with the dragged mouse pointer.
Drag & Drop a signal to a schematic frame/window	Display the schematic in which the signal is found and select it.
Drag & Drop a signal to a source frame/window	Trace signal's connectivity and highlight the result by symbols in the indicator area of the source code pane.
Drag & Drop a signal to a Temporal Flow View	Highlight the corresponding signal if it exists. Add the signal and driving instance as a reference if it doesn't exist. The global cursor time is used to identify the signal.
Drag-left	Area selection for multiple signals.
Drop an interface signal	Adds an expanded sub-group node with all interface signals at current position.
Shift-left-click a signal	Add to selection list for multiple signal selection.

Mouse Action	nWave - Value Pane
Right-click on bus or signal	Open a context-sensitive menu that provides some commands, which apply to a bus (such as Radix, Notation) or a signal (such as Edit Alias, Remove Alias).

Mouse Action	nWave - Waveform Pane
Left-click	Set the Cursor position.
Middle-click	Set the Marker position.
Right-click	Zoom to time range between the Cursor and Marker position.
Double-click	Find the signal's driver statements in source code frame.
Drag & Drop	Move the selected signals to the Signal cursor position. **NOTE**: The Signal Cursor position is moved along with the dragged mouse pointer.
Drag & Drop a signal to a schematic frame/window	Display the schematic in which the signal is found and selected.
Drag & Drop a signal to a Temporal Flow View window	Highlight the corresponding signal if it exists. Add the signal and driving instance for the current cursor time if it doesn't exist.
Drag & Drop a signal to a source frame	Trace signal's connectivity and highlight the result by symbols in the indicator area of the source code frame.
Drag-left horizontally on a waveform window, full scale ruler and zoom scale ruler	Zoom into the time range of the dragged time interval.
Drag-left vertically on an analog signal	Zoom into the value range of the dragged value interval.
Ctrl + Mouse Wheel	Zoom-in or zoom-out the time range.
Right-click a signal waveform	Open a context-sensitive menu that shows Temporal Flow View debug commands (that is, Temporal Flow View, Trace This Value, Show Fan-in, and so on.)

nSchema User Interface

You can open a new *nSchema* pane from the *nTrace* main window by clicking on the **New Schematic** icon or using the **Tools -> New Schematic from Source -> New Schematic** command. The schematic for the active unit in the design browser frame (*nTrace*) will be displayed in the *nSchema* frame, as shown in the example below.

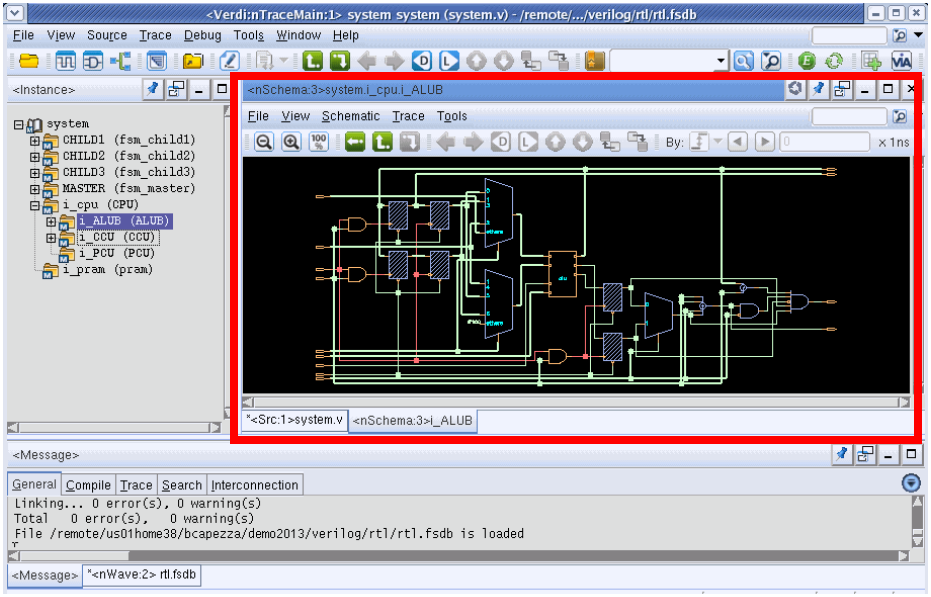


Figure: Example nSchema Frame

The schematic window displays the schematic generated from the corresponding HDL source code and provides another design view for debugging. You can debug the design using menu commands or mouse operations.

In *nSchema*, VDD, VCC, VEE, POWER and PWR net names are treated as supply nets and VSS, GND and GROUND net names are treated as ground nets. These nets are case insensitive. When a signal is treated as a power/ground global signal, trace actions are skipped.

An *nSchema* pane can be released from the main window to become a stand-alone window by clicking the **Undock** toolbar icon. An *nSchema* stand-alone window is shown below.

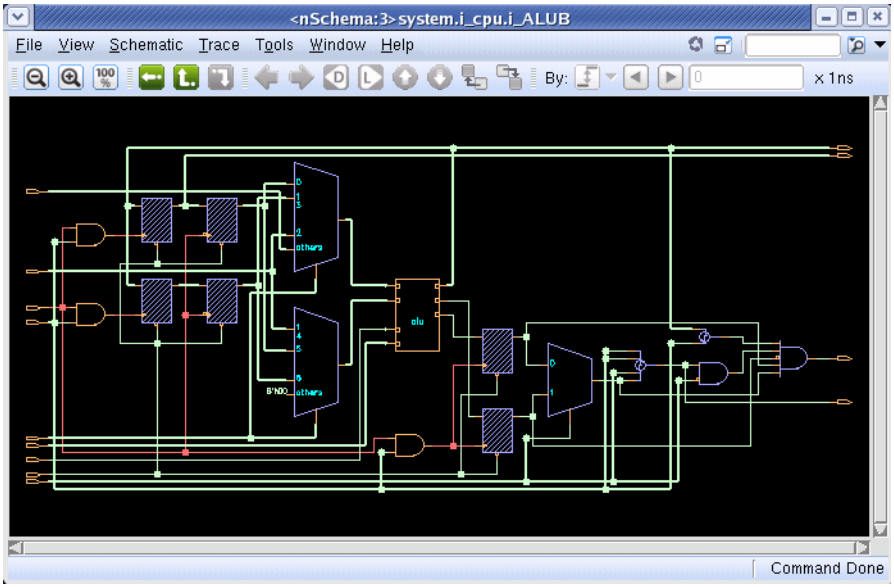


Figure: nSchema Stand-Alone Window

nSchema Toolbar Icons

Refer to the *Toolbar Icons and Fields* section in the *nSchema* chapter of the *Verdi and Siloti Command Reference* manual for information regarding available toolbar icons.

NOTE: The default toolbar can be modified through the **Tools -> Customize Menu/Toolbar** command.

nSchema Mouse Operations

The following table lists the *nSchema* mouse actions.

Mouse Action	Schematic Window
Left-click a signal/instance	De-selects the current selection and select the signal/ instance.
Left-click drag-and-drop	Select the object first and then drag-and-drop the object to the new location using the left mouse button on the indicated item.
Shift-left-click a signal/ instance	Adds the signal to a selection list for multiple signals/ instances selection.

Left-click anywhere without a signal/instance	De-selects all.
Drag-left	Zooms in an area.
Right-click	Opens a context-sensitive menu.
Double-click a signal	Highlights the connection (driving instance to loading instances with the connecting net) for the selected signal.
Double-click an instance	Pushes view into the schematic for the instance.
Drag & Drop an instance to a waveform frame/window	Displays the corresponding instance's I/O signal waveform.
Drag & Drop an RTL block to a waveform frame/window	Displays the corresponding RTL block's I/O signal waveform.
Drag & Drop a signal to a waveform frame/window	Displays the corresponding signal's waveform.
Drag & Drop an instance to a source frame/window	Finds and highlights the associated instance in source code pane/window.
Drag & Drop an RTL block to a source frame/window	Finds and highlights the corresponding source code of the RTL block.
Drag & Drop an instance / RTL block to a Temporal Flow View window	Highlights the corresponding instance's output signal if it exists. Adds the instance as a reference if it doesn't exist. The global cursor time is used to identify the signal.
Drag & Drop a signal to a Temporal Flow View window	Highlights the corresponding signal if it exists. Adds the signal and driving instance as a reference if it doesn't exist. The global cursor time is used to identify the signal.
Drag & Drop a signal to a source frame/window	Traces the signal's connectivity in the source code pane/window.
Drag & Drop any state from nSchema to nState	When you drag an FSM block from a schematic pane/window to an *nState* window, *nState* displays the state diagram of that FSM block.
Drag & Drop any state from nSchema to nWave	When you drag an FSM block from a schematic pane/window to an *nWave* pane/window, *nWave* adds all the I/O and state signals of that FSM block to the location of the cursor bar in the *nWave* pane/window.
Ctrl + Mouse Wheel	Zooms-in or zooms-out an area.
Ctrl + Drag-left	Pans the *nSchema* window.

nState User Interface

To open an *nState* frame, double-click the finite state machine (FSM) symbol (see left) in the *nSchema* pane/window. An *nState* pane can be released from the main window to become a stand-alone window by clicking the **Undock** toolbar icon. An example *nState* window/pane is shown below:

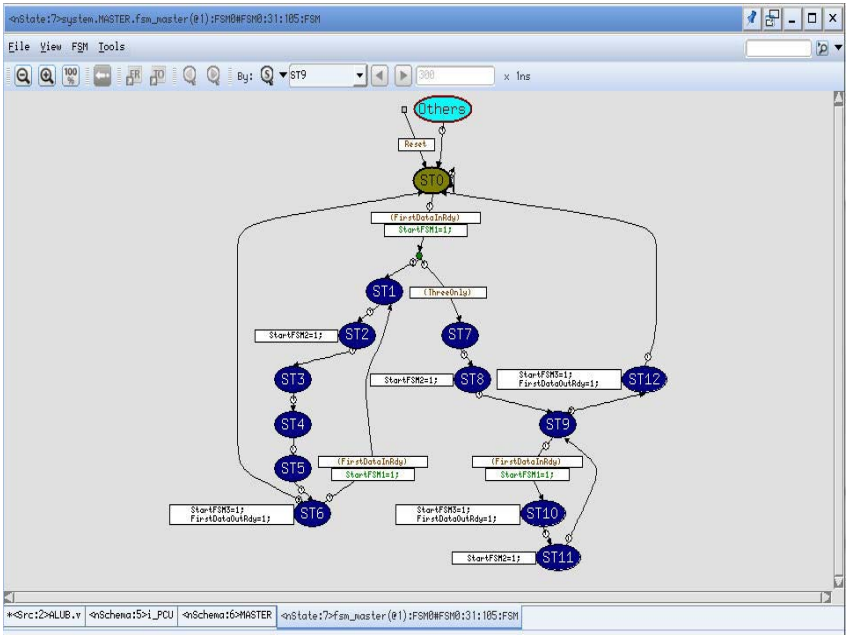


Figure: Example nState Frame

The *nState* window displays the generated bubble diagram for the corresponding state machine and provides another design view for debugging and understanding your finite state machine. You can debug your finite state machine using the menu commands or mouse actions in this window.

nState Toolbar Icons

Refer to the *Toolbar Icons and Fields* section in the *nState* chapter of the *Verdi and Siloti Command Reference* manual for information regarding available toolbar icons.

NOTE: The default toolbar can be modified through the **Tools -> Customize Menu/Toolbar** command.

nState Mouse Operations

The following table lists the *nState* mouse actions.

Mouse Action	nState Window
Right-click a transition in a *nState* window	A transition-context-sensitive menu opens for the commands: **Jump to From State**, **Jump to To State**, **Fit Select Set**, **Transition Condition,** and **Properties**.
Left-click drag-and-drop	Select the object first and then drag-and-drop the object to the new location using the left mouse button on the indicated item.
Right-click a state in a nState window	A state-context-sensitive menu opens for the commands: **State Action**, **Fit Select Se**t, and **Properties**.
Right-click the white space in a nState window	A finite-state-machine-context-sensitive menu opens for the commands: **Zoom All**, **Last View**, **Edit Search Sequence**, **Print**, and **Properties**.
Double-click a port in a nState window	If there are two ports, a properties dialog box opens to select the state. If there is only one port, go to the state properties directly.
Ctrl + Drag-left	Pans the *nState* window.
Drag & Drop any state or transition from nState to nSchema	When you drag any state or transition from inside an *nState* window to a *Schematic* window, *nSchema* displays the schematic with the corresponding FSM block whose state diagram is shown in that *nState* window.
Drag & Drop any state or transition from nState to nTrace	When you drag any state or transition from inside an *nState* window to the *Source code* pane, the corresponding source code is highlighted.
Drag & Drop any state or transition from nState to nState	When you drag a state or transition from one *nState* window to another *nState* window, the target *nState* window displays the same state diagram as in the source *nState* window; that is, the two *nState* windows are synchronized. The same state or transition is highlighted in both windows.
Drag & Drop any state from nSchema to nState	When you drag an FSM block from an *nSchema* window to an *nState* window, *nState* displays the state diagram of that FSM block.

Flow View User Interface

The *Flow View* pane can be invoked from the *nTrace* main window or *nWave* pane through the **Temporal Flow View -> New Temporal Flow View** command. A *Flow View* pane can be released from the main window to become a stand-alone window by clicking the **Undock** toolbar icon. An example of a *Temporal Flow View* pane/window is shown below.

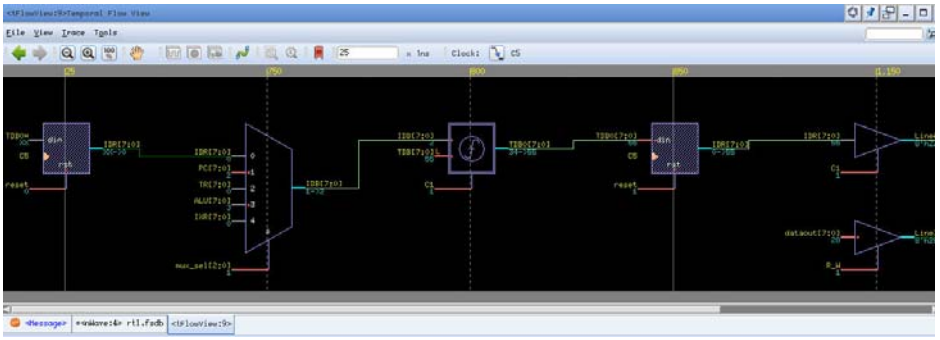


Figure: Example Temporal Flow View Window

The *Flow View* window displays a generated view of your design over time, starting from the selected reference signal and time. This provides another view in which to debug your design using the menu commands or mouse actions.

From a *Temporal Flow View* window, you can open the *Temporal Register Flow View* or the *Compact Temporal Flow View.* Refer to the *Verdi and Siloti Command Reference* manual for detailed information regarding these views.

Flow View Toolbar Icons

Refer to the *Toolbar Icons and Fields* section in the *Flow View* chapter of the *Verdi and Siloti Command Reference* manual for information regarding available toolbar icons.

NOTE: The default toolbar can be modified through the **Tools -> Customize Menu/Toolbar** command.

Flow View Mouse Operations

The following table lists the *Flow View* mouse actions:

Mouse Action	Temporal Flow View
Left-click a signal or instance or instance pin	De-selects the current selection and selects the signal/instance/port.
Left-click drag-and-drop	Select the object first and then drag-and-drop the object to the new location using the left mouse button on the indicated item.
Ctrl-left-click a signal/instance	Adds the signal/instance to the selection for multiple signals/instances selection.
Left-click anywhere without a signal/instance	Deselects all.
Drag-left in main display area (pan mode)	Pans left, right, up, or down
Drag-left in main display area (pointer mode)	Zooms in area.
Drag-left on time ruler	Zooms in area.
Right-click instance or instance pin	Opens a context-sensitive menu.
Double-click an instance pin	Traces the signal's drivers.
Drag & Drop an instance to a waveform window	Displays the corresponding instance's I/O signal waveform.
Drag & Drop an instance pin to a waveform window	Displays the corresponding signal's waveform.
Drag & Drop an instance to a source window	Finds and highlights the source code associated with the instance.
Drag & Drop an instance pin to a source window	Traces the signal's connectivity in the source code frame.
Drag & Drop an instance pin to an nSchema window	Changes the scope to the signal's hierarchy and highlights the corresponding signal.
Drag & Drop an instance to an nSchema window	Changes to the instance's hierarchy and highlights the corresponding instance.
Left-click an instance with nWave icon enabled	Adds the instance IO to nWave if they don't exist. Highlights the instance output if it exists.
Left-click an instance output port with Show Source Code icon enabled	Finds and highlights the source code associated with the output signal.

Left-click an instance with Show Source Code icon enabled	Finds and highlights the source code associated with the instance.
Ctrl + Mouse Wheel	Zooms-in or zooms-out an area.

Transaction/Message User Interface

The transaction/message FSDB file is loaded into *nWave* the same way as a general FSDB file. A stream name is shown in the *Signal* pane; begin time, end time, and attributes are shown in the *Value* pane; and the transaction/message is shown in the *Waveform* pane as rectangles enclosing all the attributes.

Detailed Transaction/Message View in nWave

The following figure summarizes the different aspects of transaction/message viewing in *nWave*.

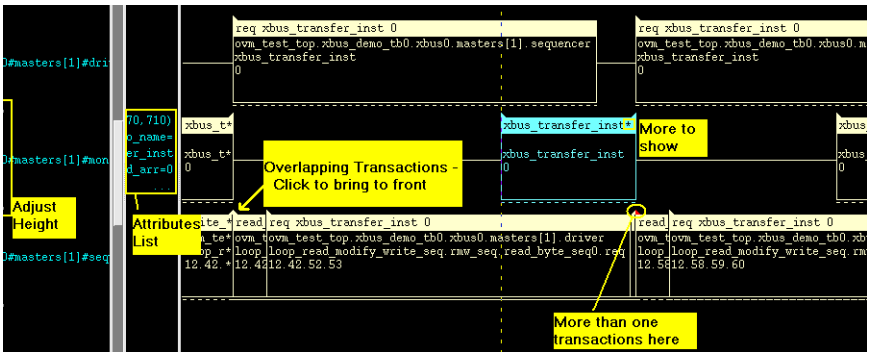


Figure: Detailed Transaction/Message View

Although there is a begin time and end time in a transaction/message, when you click a transaction/message, the cursor is located at the begin time. When you select a stream, you can click the **Search Backward/Search Forward** icons (left/right arrows) on the *nWave* toolbar to step through the transactions/messages. A dashed line under the transaction/message box indicates there are more attributes than are currently displayed. You can increase (decrease) the height of the stream in the *Signal* pane to show more (less) attributes.

Alternatively, you can move the cursor on top of the transaction/message attributes in the *Value* pane (middle column) to activate a yellow tip window showing all attributes as displayed in the following figure:

Figure: Transaction/Message Tip

Individual transactions/messages can be selected by clicking on the label in the *Waveform* pane; the background color of the selected transaction/message changes to light blue. Pressing the **Search Backward/Search Forward** toolbar icons will not change the selected transaction/message but will change waveform cursor time.

The selection is important for viewing covered or obscured transactions/messages when there is a time overlap for multiple transactions/messages. The top triangle is used to select the underlying transaction/message and bring it to the front. You can also select a stream and then click **Waveform -> Classic Transaction -> Expand/Shrink Overlapping** or **Waveform -> Classic Message -> Expand/Shrink Overlapping** commands to remove transaction/message overlap.

If there are transactions/messages related to the selected one, the related transaction/message is highlighted with a pink background color, similar to the following example.

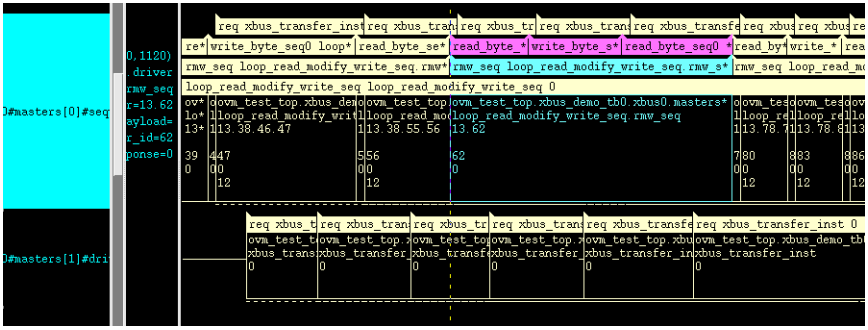


Figure: Transaction/Message Relationships

Transaction/Message Properties

Transactions/Messages contain a lot of data. You can view attributes and relationships of a selected transaction/message in a tabular format. To open the *Transaction Property* or *Message Property* form, select a transaction or a message, right-click to open the context menu, and choose the **Properties** command. The **Attributes** tab summarizes the transaction/message attributes, as shown in the following example:

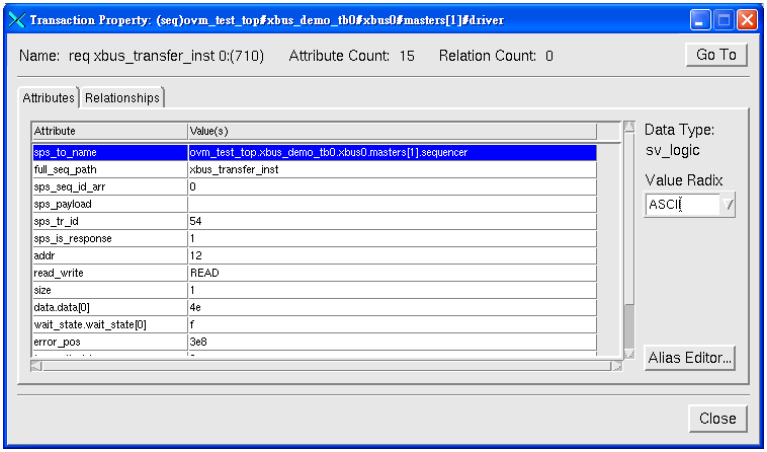


Figure: Transaction Property Dialog Window - Attributes

You can view the selected transaction relationships by selecting the **Relationship** tab in the *Transaction Property* form.

Transaction/Message Attributes

You can use string matching to search attributes. In *nWave*, click the **Waveform -> Set Search Attributes** command to open the *Search Attribute Value* form. Alternatively, you can left-click the **Search By:** icon on the toolbar and select the **Transaction Attribute Values** option.

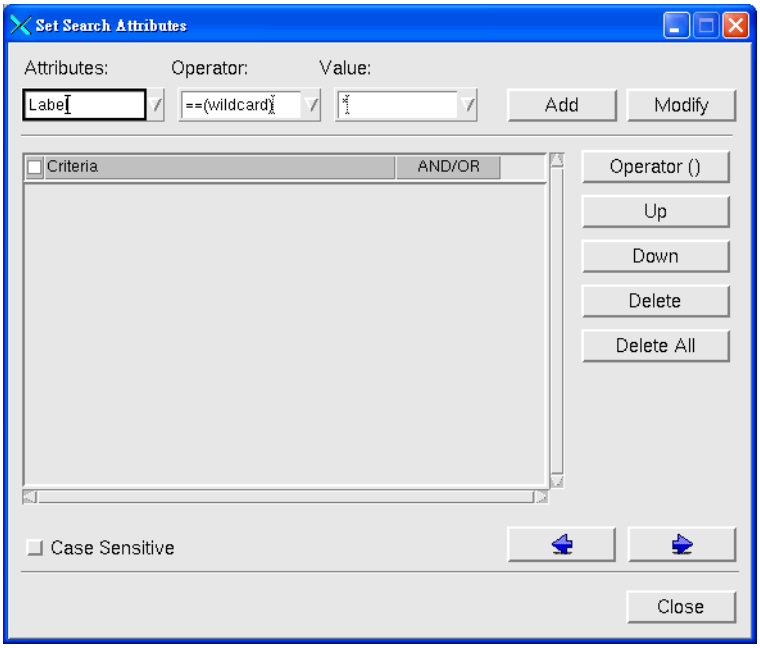


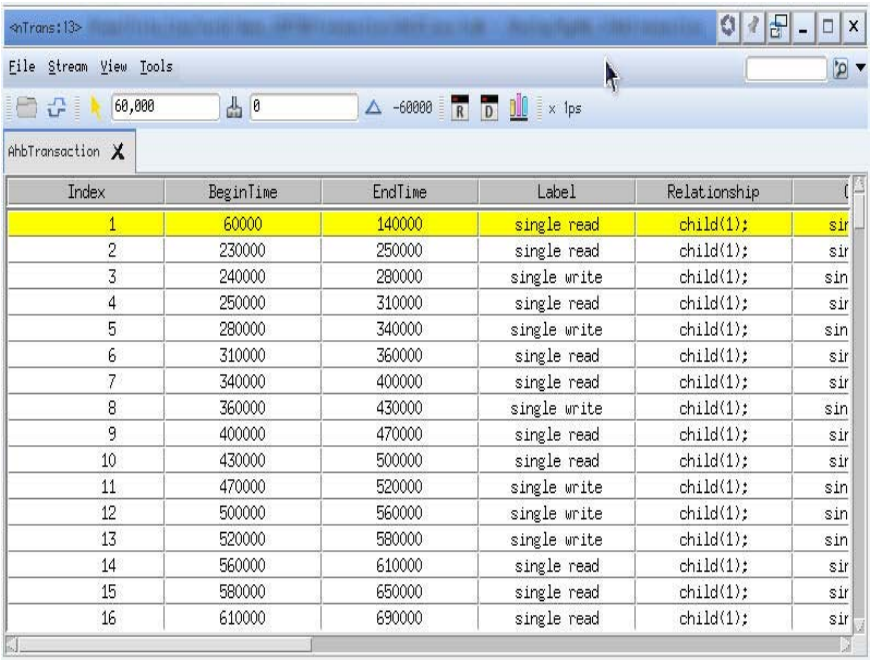
Figure: Search Attribute Value Form

You can specify the attribute name and value. After you've entered the search criteria and clicked **OK**, you can use the **Search Forward/Search Backward** icons on the *nWave* toolbar to step through the transactions/messages of the selected streams.

Analyzing Transactions/Messages

In addition to the waveform viewing capability for transactions/messages, you can open the *Transaction Analyzer* window by clicking the **Tools -> Classic Transaction -> Analysis Window** command (or the **Tools -> Classic Message -> Analysis Window** command) from *nWave*. After the window is open, you can load one or more streams individually or merge multiple streams together.

The window is similar to the following:



Index	BeginTime	EndTime	Label	Relationship	
1	60000	140000	single read	child(1);	sir
2	230000	250000	single read	child(1);	sir
3	240000	280000	single write	child(1);	sin
4	250000	310000	single read	child(1);	sir
5	280000	340000	single write	child(1);	sin
6	310000	360000	single read	child(1);	sir
7	340000	400000	single read	child(1);	sir
8	360000	430000	single write	child(1);	sin
9	400000	470000	single read	child(1);	sir
10	430000	500000	single read	child(1);	sir
11	470000	520000	single write	child(1);	sin
12	500000	560000	single write	child(1);	sin
13	520000	580000	single write	child(1);	sin
14	560000	610000	single read	child(1);	sir
15	580000	650000	single read	child(1);	sir
16	610000	690000	single read	child(1);	sir

Figure: Transaction Analyzer Window

For the current selected stream (or merged streams), you can use the **View -> Search** command to locate a string or pattern, or the **View -> Filter/Colorize** command to filter and display transactions/messages whose attributes match user-specified conditions. These commands allow you to navigate the streams quickly and focus on the transactions/messages of interest. After clicking the **Sync. Signal Selection Enabled** icon (see left) on both the *Transaction Analyzer* pane and the *nWave* pane, you can select a transaction/message in the spreadsheet view and the corresponding transaction/message gets selected in the waveform.

nCompare User Interface

The *nCompare* pane compares simulation results stored in FSDB dump files using flexible, user-specified comparison criteria. Optimized for extremely fast comparison of large data sets, the *nCompare* pane is fully integrated with the Verdi platform to intuitively display any differences between runs.

The *nCompare* pane can be invoked by invoking the **Tools -> nCompare** command from the *nWave* pane. After the pane is opened and the waveform comparison is completed, the *nCompare* pane is displayed as shown below.

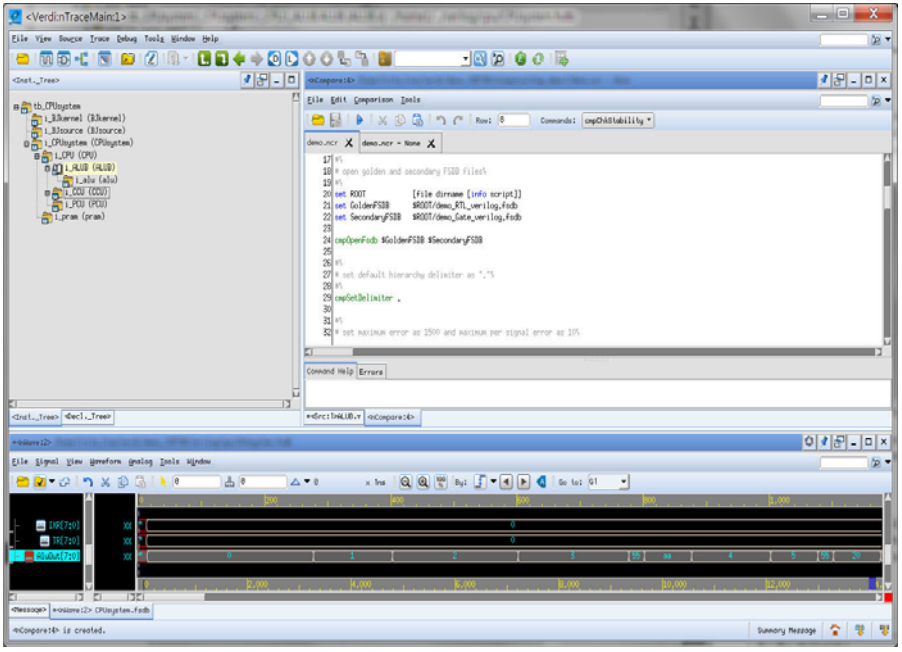


Figure: nCompare Pane

Comparing Different Simulation Runs

The *nCompare* pane is used to compare different simulation runs to find the mismatch simulation errors between pre-synthesis/post-synthesis, different clock speed of same design, different technology, or simulation files which are generated from different simulators.

Rule File

The rule file is described using Tcl language. The *nCompare* module uses Tcl language and nCompare-defined-Tcl-extended comparison commands to describe the comparison rules and specify comparison options.

A basic rule file should have at least the following three parts:

1. Specification of golden and secondary simulation files.
2. Specification of compared signal pairs.
3. Start time-based comparison.

The following is a simple rule file that would compare all signals in *1.fsdb* and *2.fsdb*:

```
cmpOpenFsdb 1.fsdb 2.fsdb
cmpSetSignalPair top -level 0
cmpCompare
```

Compare Waveforms and View Errors

After the rule file is created and the comparison is completed in the GUI or using the *nCompare* utility, the *nCompare* pane shows the mismatch errors. The errors can be sorted by design or time and easily traversed.

nCompare Mouse Operations

The default mouse action in the *nCompare* pane is summarized in the table below.

Mouse Action	Command Operations
Double-click a mismatch error node	This action launches the waveform tool, adds the mismatch signals into the waveform tool and changes the cursor time of the waveform tool to the mismatch time.

nECO User Interface

The *nECO* module provides the ability to perform gate-level engineering change orders (ECOs) in the flexible schematic views. The *nECO* module takes full advantage of the sophisticated capabilities in the Verdi platform to propagate changes throughout the design hierarchy and automatically create any new nets and ports that are required.

Refer to the *User Interface* chapter of the *nECO User's Guide and Tutorial* for details.

nAnalyzer User Interface

The *nAnalyzer* module provides the ability to analyze clock and reset trees (including crossing paths), to qualify Clock Tree Synthesis (CTS), to annotate standard delay format (SDF) files and CTS results, to load and display timing results from standard timing analysis tools, and to perform switching analysis on the design. These functions build on top of the functional debug aspects of the Verdi platform. The *nAnalyzer* module uses the same interface as *nSchema*.

Refer to the *nAnalyzer User's Guide and Tutorial* for details.

Before You Begin

Before you begin the tutorial, you (or your system manager) must have installed the Verdi and Siloti platforms as described in the accompanying *Installation and System Administration Guide*.

NOTE: The optional demo package (for example, Verdi-J-201412-demo.tar.gz where J corresponds to the version, 2014 corresponds to the year, and 12 corresponds to the month) must be installed.

Installation and Setup

You must also complete the following actions to set up the Verdi environment and the files required for this tutorial:

1. Add the Verdi application (binary) to the search path and specify the search path to the license file:

   Refer to the *Setting Up the Environment and Running the Software* section in the *Installation and System Administration Guide* for details.

2. Create a working directory:

   ```
   % mkdir <working_dir>
   ```

3. All of the tutorial data resides in the *$VERDI_HOME/demo* directory. Make a copy of these demo files in your working directory:

   ```
   % cp -r $VERDI_HOME/demo <working_dir>
   ```

Demo Details

The primary demo design used in this section is a simple microprogrammed CPU design delivered with the installation. The example represents a complete design spanning the behavioral, RTL, and gate levels.

Most tutorials use the Verilog design demo. However, be sure to check the instructions for each tutorial to ensure that you are running the correct demo that was included with your installation. Use the following commands to set the tutorial data:

- For Verilog Design:
  ```
  % cd <working_dir>/demo/verilog/cpu
  ```
- For VHDL Design:
  ```
  % cd <working_dir>/demo/vhdl/rtl
  ```
- For Mixed Design:
  ```
  % cd <working_dir>/demo/mixed/rtl
  ```
- For SystemVerilog Design:
  ```
  % cd <working_dir>/demo/systemverilog
  ```
- For Transactions:
  ```
  % cd <working_dir>/demo/transaction
  ```

Launching Techniques

This chapter summarizes the various methods for starting the Verdi platform, loading the design, and loading the simulation results stored in the Fast Signal Database (FSDB).

Dumping Elaboration Database

Overview

In previous releases, Verdi elaborated the design at runtime. As a result, loading a complex design took a lot of time and memory. Starting from the Verdi L-2016.06 release, a new elaboration process is introduced. Verdi elaborates the design and saves the elaboration database to the disk in batch mode. The elaboration time and memory usage is reduced significantly by using this new process.

Use Model

The following sections provide a detailed description for creating/generating the elabDB flow:

- Interactive Debug Flow
- Post-Simulation Debug Flow

Interactive Debug Flow

You can perform interactive simulation debugging by creating/generating the elabDB flow. The following sections describe how to elaborate your design and load the generated elaboration database into Verdi in interactive simulation debug mode.

- Generating Verdi Elaboration Database Using VCS
- Loading Verdi Elaboration Database into Verdi

Generating Verdi Elaboration Database Using VCS

You can generate the Verdi KDB using the VCS `-kdb` option either in the VCS two-step flow or three-step flow. In the VCS two-step flow, add the `-kdb` option to the command line to generate the KDB. In case of VCS three-step flow, add the `-kdb` option in all the **vlogan/vcs** command lines.

When you specify the `-kdb` option, VCS creates the Verdi KDB and dumps the design into the libraries specified in the **synopsys_sim.setup** file. To determine which databases are to be generated, specify one of the following arguments with the `-kdb` option:

- `-kdb=only`

  Generates only the Verdi KDB that is needed for both post-process and interactive simulation debug with Verdi.

  In VCS two-step flow, this option does not generate the VCS compile data/executable, and does not disturb the existing VCS compile data/executables.

  For example,

  ```
  % vcs -kdb=only <compile_options> <source files>
  ```

  In VCS three-step flow, this option dumps the minimum data required at analysis stage, and does not disturb the existing VCS compile data/executables.

  ```
  % vlogan –kdb=only <vlogan_options> <source files>
  ```

The following is the sample flow:

```
% vlogan test.v -sverilog
% vcs top –debug_access
% simv

// When the simulation fails, you can debug it in Verdi.
// You can only generate the KDB database for debugging with
Verdi, and do not need to remove anything here.

% vlogan -kdb=only test.v -sverilog // This step creates the
KDB database.
// There is no need of vcs compilation step as the simv was
already generated successful in the previous steps.
//You can invoke Verdi to do interactive simulation debug.

% simv -gui=verdi
//You can also invoke Verdi to do post-simulation debug.
% verdi -simflow –simBin <simv_path/simv> -ssf novas.fsdb
```

NOTE: If you want to compile a design that includes packages in the pre-compiled KDBs that were previously compiled with the `-kdb=only` option, you can only use the **vericom/vhdlcom** utility to include the pre-compiled KDB libraries.

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

Loading Verdi Elaboration Database into Verdi

When the elaboration database is generated as described in the **Generating Verdi Elaboration Database with Unified Compiler Front End** section and the `simv` simulator executable is generated, you can invoke Verdi in interactive simulation debug mode using the `-gui`/`-verdi`/`-gui=verdi` options.

Example1:

```
% simv <simv_options> -verdi [-verdi_opts “<verdi_options>”]
```

Example2:

```
% simv <simv_options> –gui=verdi [-verdi_opts
“<verdi_options>”]
```

The elaboration database file is imported into the invoked Verdi automatically in interactive simulation debug mode.

Post-Simulation Debug Flow

You can perform post-simulation debug by creating or generating the elabDB flow. The following sections describe how to elaborate the design and load the generated elaboration database into Verdi with or without the FSDB file.

- Generating Verdi Elaboration Database with Unified Compiler Front End
- Generating Elaboration Database Using VCS Elaboration Command
- Loading Verdi Elaboration Database into Verdi

Generating Verdi Elaboration Database with Unified Compiler Front End

Creating/generating the elabDB flow is supported in both VCS two-step and three-step flows with Unified Compiler Front End. The following method is used to generate the elaboration database:

- Generate the elaboration database during VCS elaboration (see the **Generating Elaboration Database Using VCS Elaboration Command** section).

Generating Elaboration Database Using VCS Elaboration Command

The `-kdb` option is provided on the VCS command line to generate the KDB and elaboration database of your design. For example,

VCS two-step flow

```
% vcs -kdb <vcs options> -verdi_opts "<verdi_options>"
```

The generated KDB and elaboration database are saved in the `work.lib++` and `kdb.elab++` directories. The `work.lib++` directory is saved in the same directory as `simv` and the `kdb.elab++` directory is saved under the `simv.daidir` directory.

The other usage for the VCS two-step flow is to generate the `work.lib++` and `kdb.elab++` directories by specifying `-kdb=only`. If you simply want to do static debug and do not want to run simulation, specify the `only` value and VCS only generates elaborated KDB and does not generate simv for simulation.

VCS three-step flow

```
% vlogan -kdb <vlogan options> <source files>
% vhdlan -kdb <vhdlan options> <source files>
% vcs -kdb <top_name>
```

The generated KDB are saved in the `work.lib++` directory, same as the analysis database of `vlogan` and `vhdlan`. The elaboration database *kdb.elab++* directory is saved in the `simv.daidir` directory.

Loading Verdi Elaboration Database into Verdi

When the elaboration database is generated, you can import it into Verdi using the `-elab <elabDB path without the .elab++ postfix >` option. For example,

Example 1: Load the `./kdb.elab++` elabDB

```
% verdi -elab kdb
```

Example 2: Load the `./myelab/mydesign.elab++` elabDB

```
% verdi -elab ./myelab/mydesign -ssf novas.fsdb
```

Reference Source Files on the Command Line

This method loads the design directly from the source files. It is not recommended for mixed language designs.

1. Use the following command to reference the source files on the command line:

   ```
   % verdi -f <source_file_name>
   ```

   Where, *source_file_name* is a file that lists all of the HDL source files.

2. Use the **File -> Open Waveform File** command to load the FSDB.

Compile Source Code into a Library

This method must be used if you have a mixed language design.

1. Use the utility program *vhdlcom* (supplied with the Verdi and Siloti installation) to compile your VHDL source code into a library and use *vericom* for Verilog code:

   ```
   % vericom -lib <libName> block1.v block2.v ...
   % vhdlcom -lib <libName> block1.vhd block2.vhd ...
   ```

2. Use the following command to load the compiled design:

   ```
   % verdi -lib <libName> -top <TopBlock>
   ```

   Where, *libName* is the compiled library and *TopBlock* is the highest-level block you wish to see.

Reference Design and FSDB on the Command Line

1. Use the following commands to reference both the source files and the FSDB on the command line:

   ```
   % verdi -f <source_file_name> -ssf <fsdb_file_name>
   ```

   Where, *source_file_name* is the source file name and *fsdb_file_name* is the name of the FSDB file.

2. Use the following commands to reference both the compiled library and the FSDB on the command line:

   ```
   % verdi -lib <libName> -top <TopBlock>
   -ssf <fsdb_file_name>
   ```

   Where, *libName* is the compiled library, *TopBlock* is the highest-level block you wish to see, and *fsdb_file_name* is the name of the FSDB file.

NOTE: If the specified FSDB file is an Essential Signal FSDB, you are presented with a *Question* dialog related to Data Expansion. If you plan to use the Siloti system, click **Yes**; otherwise, click **No**. Data Expansion can always be started or the options changed by invoking the **Tools -> Visibility -> Data Expansion -> Setup Data Expansion** command.

Perform Behavior Analysis on the Command Line

To perform Behavior Analysis on the command line, you must specify the `-ba` option.

1. Referencing the source files:

   ```
   % verdi -f <source_file_name> -ba -ssf <fsdb_file_name>
   ```

   Where, *source_file_name* is the source file name and *fsdb_file_name* is the name of the FSDB file.

2. Referencing the compiled library:

   ```
   % verdi -lib <libName> -top <TopBlock> -ba
   -ssf <fsdb_file_name>
   ```

   Where, *libName* is the compiled library, *TopBlock* is the highest-level block you wish to see, and *fsdb_file_name* is the name of the FSDB file.

NOTE: The `-ba` option executes Behavior Analysis using the Behavior Analysis settings from the *novas.rc* resource file unless you specifically include them on the command line. Refer to the *verdi* utility description in the *Verdi and Siloti Command Reference* for a list of Behavior Analysis options.

Replay a File

1. Use the following command to replay a file containing commands that load both the design and the FSDB (and perform a variety of other tasks):

   ```
   % verdi -play <command_file_name>
   ```

   Where, *command_file_name* is a file with a number of Tcl commands.

Start Verdi Without Specifying Any Source Files

1. Use the following command to start the Verdi platform:

   ```
   % verdi
   ```

   A blank *nTrace* main window is displayed.

2. From the main menu, choose the **File -> Import Design** command (or the **Import Design** icon on the toolbar) to open an *Import Design* form, similar to the example below:

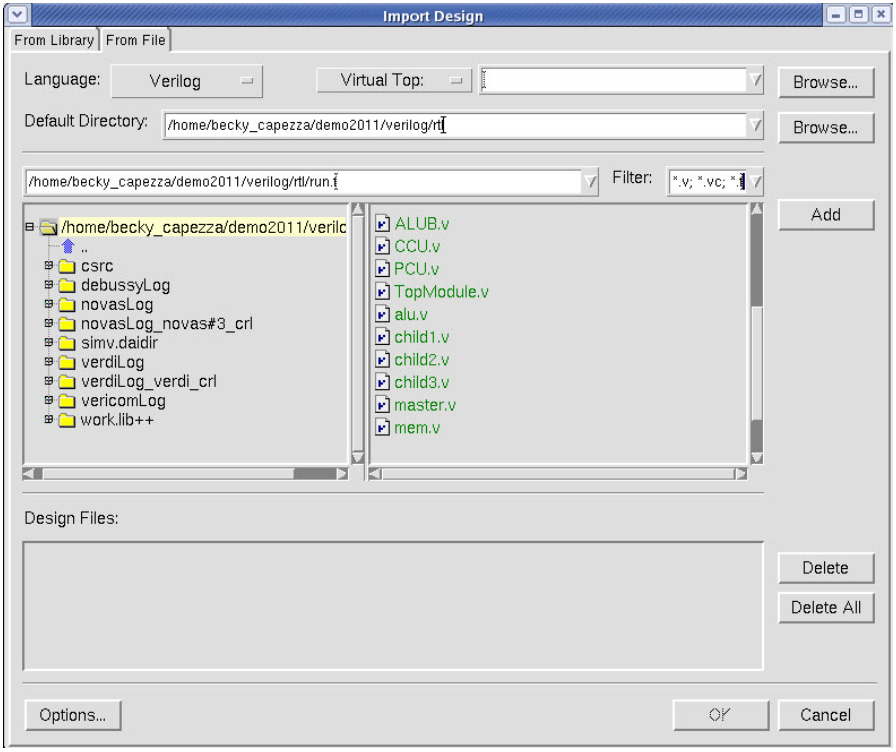


Figure: Example Import Design Window

3. If you are loading the design from the source files directly, do the following:
   a. Click the **From File** tab at the top of the window.
   b. Select the HDL language in the **Language** selection field.
   c. To open a folder, click the folder name. A list of sub-folders and/or files appears to the right.

d. Double-click the name of the files or click the **Add** button to the right of the window to add the file to the path name, which appears in the white space directly above the two windows in which you are working.

e. Select the design file(s) of interest. The recommendation is to use a run file where the individual design files are listed.

f. Click the **Add** button.

g. Click the **OK** button.

4. If you are loading the design from a compiled library, do the following:

   a. Click the **From Library** tab at the top of the window.

   b. To select a library, click the library name. A list of design units appears to the right.

   c. Select the top design unit.

   d. Click the **Add** button.

   e. Click the **OK** button.

5. Use the **File -> Open Waveform File** command to load the FSDB.

   You should see an *nTrace* main window with the design information displayed.

Loading when Design and FSDB Hierarchies do not Match

Use the following command if you have dumped the FSDB file for the entire design, but only want to load a portion of the design for debug:

```
% verdi -f <source_file_name> -ssf <fsdb_file_name>
-vtop <map_file_name>
```

Where, *source_file_name* is the source file name, *fsdb_file_name* is the name of the FSDB file, and *map_file_name* is the name of the map file.

The map file is used to match the design hierarchy to the hierarchy in the FSDB file so the simulation results are correctly annotated on the source code and schematic views. The Verdi platform automatically generates a virtual hierarchy in the design according to the definitions in the *map* file. The syntax is as follows:

```
module_name = hierarchical_instance_path
```

NOTE: The map file matches the case sensitivity of the associated language. For example, Verilog is case sensitive so if the module definition is all capitalized, the map file description needs to be as well (that is, cpu does not equal CPU). VHDL is not case sensitive (that is, cpu equals CPU).

If you incorrectly enter the module name in the map file, you may see an error similar to the following in the compiler.log (**File -> View Import Log**):

```
*Error* view cpu is not defined for inst i_cpu
“virtual_top_autov_15123.gen:, 7:
```

This error needs to be eliminated. Check the module definition in the source code and confirm that the map file matches it exactly.

User Interface Tutorial

Overview

The Verdi platform is a multi-window docking application with a flexible and easy-to-use graphical user interface (GUI).

The Verdi platform layout can be customized by dragging a frame away from its original position (undocking) and then dropping it in a new position (docking) to attach it to the left, right, above, below, or on top of another frame. Similarly, a toolbar category can be moved to the left, right, above, or below the relevant frame. The bind key of any command can be changed and the toolbar categories arranged fairly easily.

Before you begin this tutorial, follow the instructions in the *Before You Begin* chapter.

Refer to the *Launching Techniques* chapter for more information on starting the Verdi platform, and refer to the *User Interface* chapter for more details regarding the interface.

Start Verdi Platform

1. Change the directory to <*working_dir*>*/demo/verilog/rtl*.

   ```
   % cd <working_dir>/demo/verilog/rtl
   ```

2. Start the design using the following command:

   ```
   % verdi -f run.f
   ```

Using the Welcome Page

1. Choose the **Help -> Welcome** command if the *Welcome* page is not displayed.

Figure: Welcome Page

2. On the *Welcome* page, click the **What's New** icon and then click the **Application Notes** icon. You can see the application note files and the FAQ file here.
3. Click the **Home Page** icon to go back to the *Welcome* page and then click the **Work Modes** icon.

Figure: Work Modes Page

4. Select the **Testbench Debug Mode** option and then click the **Go to Work** button.

The window layout of the Verdi platform adds the *Constraint*, *Inheritance*, *FSDB_Msg* and the *Static* frames for testbench code browsing and message debugging purposes.

5. Click the **Welcome** icon  in the lower right corner of the main window to display the *Welcome* page.
6. Click the **Work Modes** icon to show to the *Work Modes* page again.
7. Select the **Hardware Debug Mode** option and then click the **Go to Work** button.

   The window layout of the Verdi platform is now optimized for **Hardware Debug Mode**.

NOTE: The work mode may also be specified on the Verdi command line with the `-workMode` option.

Saving and Restoring a Session

1. Choose the **File -> Save Session** command and save the current session to `"my.ses"`.
2. Go to the *Welcome* page and click the **History** icon to show the *History* page.
3. Select the **my.ses** option on the *History* page. A screen shot of the session is displayed to the right as shown in the following figure:

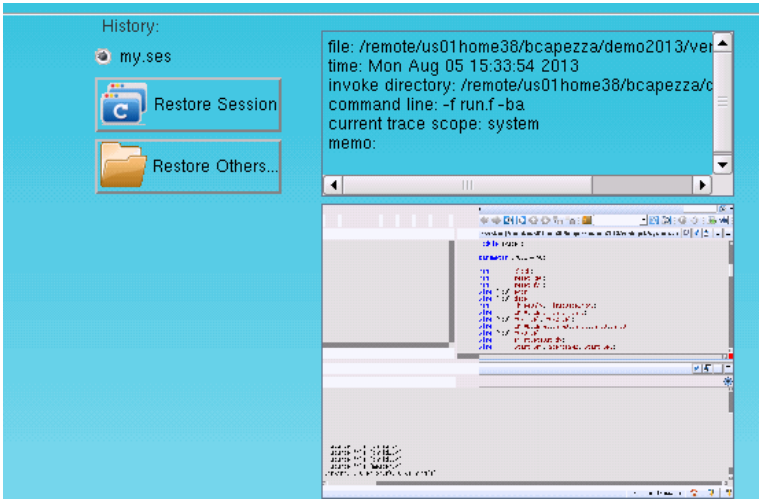


Figure: Session Preview

4. Select the `novas_autosave.ses` option.
   The image shows the time the session was saved.

NOTE: If you do not see this file, exit the Verdi session and start it again. The file is automatically created on exit.

5. Click the **Restore Session** button to restore the **novas_autosave.ses** session.
6. Go back to the *History* page and restore the **my.ses** session.

Changing the Default Frame Location

1. Click the **New Waveform** icon to open an *nWave* window.
   The *nWave* frame is added as a new tab on top of the *Message* frame.
2. Click the frame banner of the *nWave* frame and drag it to the left.
   The *nWave* frame is now floating (undocked).
3. Drag the *nWave* frame around and see that the dockable area (outlined with a dashed line) is changing along with the cursor position.
4. Drop the *nWave* frame to dock it.
5. Undock and dock the *nWave* frame (or another frame) to different positions several times to become familiar with the usage.

Maximizing the Display

1. Click the **Undock** icon on the *nWave* toolbar to make it a stand-alone window.
2. Click the **Dock** icon on the *nWave* stand-alone window.
   The *nWave* frame is docked to the main window again.
3. Double-click the *nWave* frame banner. This maximizes the *nWave* frame size and makes it easy to see the content of the frame more clearly.
4. Double-click the *nWave* frame banner again and the *nWave* frame goes back to its original size.

Modifying the Menu/Toolbar

1. Right-click the *nWave* frame banner to display the right-click command menu and select the **Menu** option to hide the menu bar.

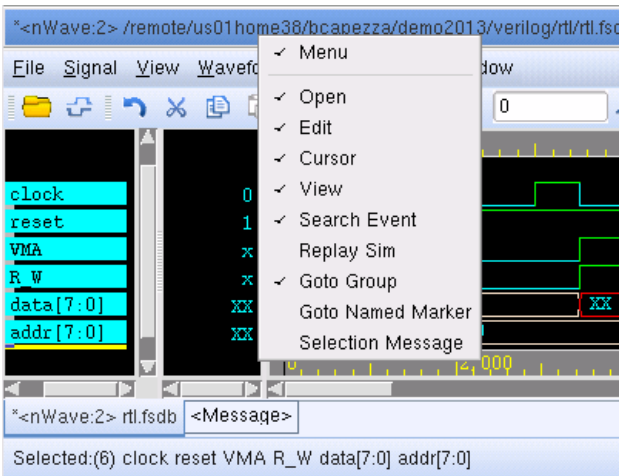


Figure: nWave Configuration Menu

2. After the **Menu** option is turned *off*, press the **Alt** key within the *nWave* frame's central area to show the menu bar. Pressing the **Alt** key again will hide the menu.
3. Turn *on* the **Menu** option in the right-click command menu.
4. On the right-click command menu, select the **Open** option to hide the icons associated with the **Open** toolbar category.

   The **Open** category disappears from the toolbar of the *nWave* frame.
5. On the right-click command menu, select the **Open** option again to restore the **Open** toolbar category.
6. Click the left handle (vertical bar) of the **Open** toolbar category and drag a little. The **Open** toolbar category is floating.
7. Drag it around and observe the dockable area (outlined with dashed line) is changing along with the cursor position.

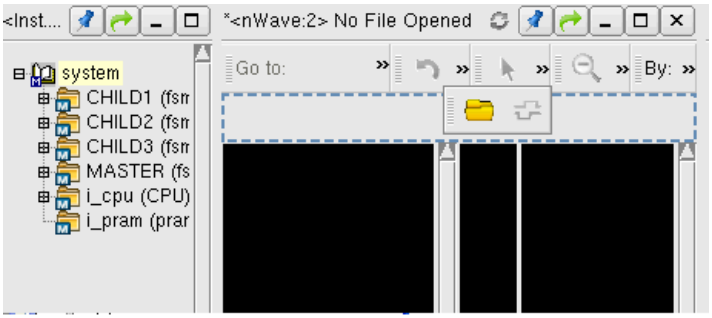


Figure: Relocating Toolbar Icons

8. Drop the **Open** toolbar category to dock it.
9. Move the **Open** toolbar category several times to become familiar with the usage.

Searching for a Command

1. On the top right corner of the main window, select the **Menu** option, type "*pref*" in the **Spotlight** text field and press the **Enter** key. This displays a list of commands matching the pattern.
2. Select one of the commands from the list, for example, **Preferences**, and then the **Preferences** command is invoked.

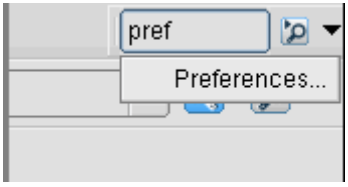


Figure: Spotlight Search Results

Customizing Bind Keys

1. Choose the **Tools -> Customize Menu/Toolbar** command from the main window.
2. Type `"trace"` in the text field and click the **Search** icon to locate the **Trace** menu.

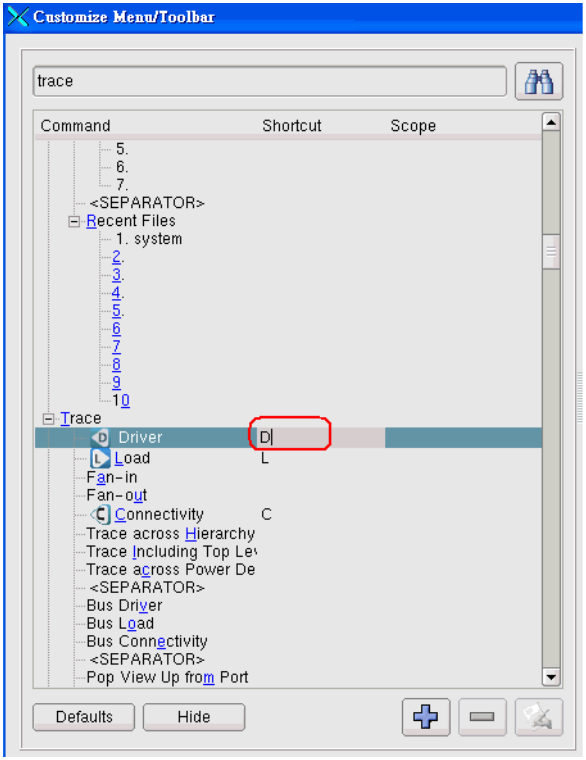


Figure: Customize Bind Key

3. Double-click the **Shortcut** cell of the **Driver** command. Press the **D** key on the keyboard to change the command's bind key to **D**.
4. Similarly, double-click the **Shortcut** cell of the **Load** command. Press the **L** key on the keyboard to change the command's bind key to **L**.
5. Click the **OK** button to complete the setting.
6. In the Source Code frame, select any signal and press the **D** key to execute the **Driver** command.
7. Press the **L** key and the **Load** command is executed. This is useful when you want to execute frequently used commands with bind keys you favor.

Customizing Toolbar Icons

1. Click the **Undock** icon on the *nWave* frame to make it become a stand-alone window.
2. Choose the **Tools -> Customize Menu/Toolbar** command from the *nWave* window.
3. Click the **Add Custom Toolbar** icon in the upper right section of the *Customize Menu/Toolbar* form to add a new toolbar category named `"new toolbar"`.

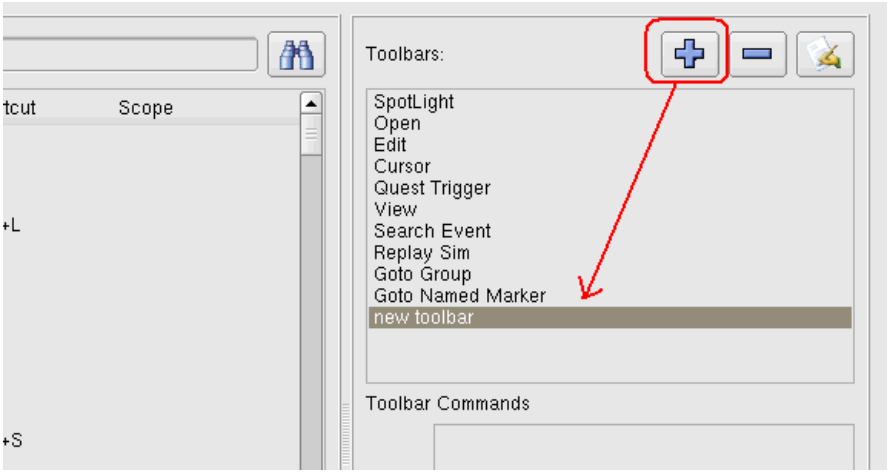


Figure: Add New Toolbar Category

4. Select the **Open** command under the **File** menu section in the left pane.
5. Click the **Add Selected Command to Toolbar** button to add the **Open** command to the newly created `"new toolbar"` toolbar category.

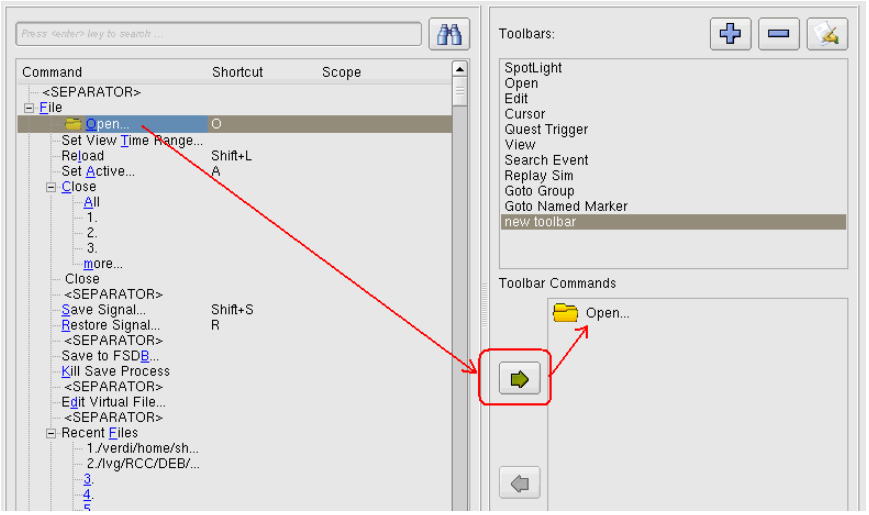


Figure: Add Commands to New Toolbar Icon Category

6. Similarly, add the **Zoom In**, **Zoom Out** and **Zoom All** commands under the **Zoom** menu section to the newly created `"new toolbar"` category.
7. Click the **OK** button to complete the setting. The new toolbar is added to the *nWave* toolbar area.

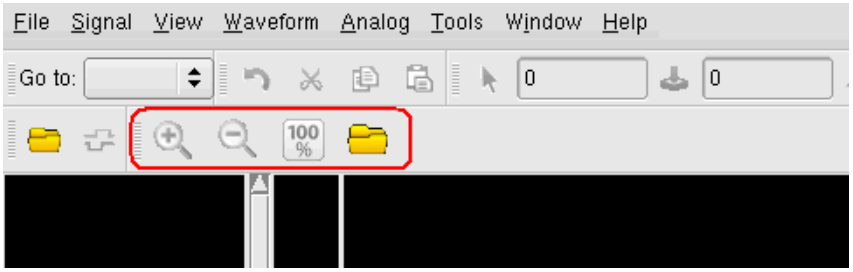


Figure: New Toolbar Icon Category

nTrace Tutorial

Overview

The *nTrace* main window is a source code viewer and analyzer that operates on the KDB to display the design hierarchy and source code (Verilog, VHDL, SystemVerilog, mixed) for selected design blocks. The Verdi platform quickly identifies signal connectivity information (drivers and loads) without any simulation overhead. With the FSDB, the simulation results can be back-annotated in the source code and then the Verdi platform can analyze and determine a signal's active driver at a particular simulation time.

Before you begin this tutorial, follow the instructions in the *Before You Begin* chapter.

Refer to the *Launching Techniques* chapter for more information on starting the Verdi platform and opening the *nTrace* main window, which is the default window. Also refer to the *User Interface* chapter for more details regarding the *nTrace* interface.

Traverse the Design Hierarchy in nTrace

You can traverse the design hierarchy to understand the design structure.

1. Change to the demo directory.

   ```
   % cd <working_dir>/demo/verilog/cpu/src
   ```

2. Start the design using the following command:

   ```
   % verdi -f run.f -workMode hardwareDebug &
   ```

NOTE: This tutorial uses a Verilog design example. The same capability is available for VHDL or mixed designs.

3. To expand a block hierarchy on the **Instance** tab in the design browser frame, click the plus symbol to the left of the *i_CPUsystem* block instance name to reveal its *i_CPU* and *i_pram* sub-blocks.

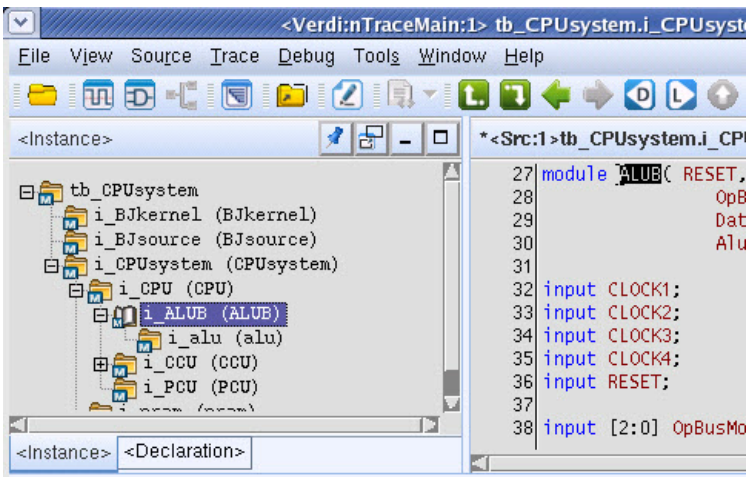


Figure: Expand the Hierarchy in nTrace

4. Click the plus symbol to the left of the *i_CPU* block instance name to reveal its *i_ALUB*, *i_CCU*, and *i_PCU* sub-blocks.
5. To collapse the hierarchy, click the minus symbol to the left of the name.

The plus/minus symbols to the left of the block instance names in the design browser frame are used to expand/collapse the display of the selected block's hierarchy.

Access a Block's Source Code

1. To access source code, double-click the *i_CPU* unit instance name in the design browser frame. The source code is displayed in the Source Code frame, as shown in the following figure:

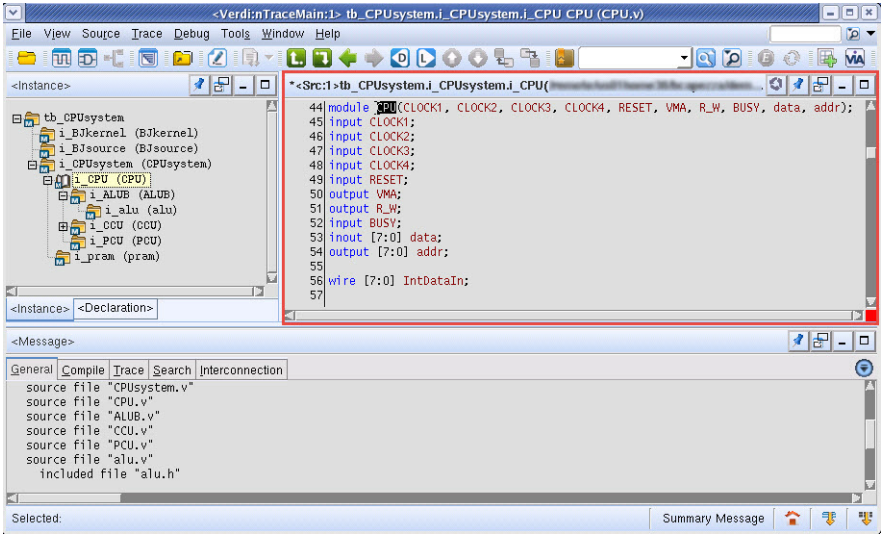


Figure: Source Code for the CPU Block

By default, the name of the block (*CPU*) is highlighted in the source code.

2. Double-click *CPU* to change the source code context to the calling block, which is *CPUsystem* (the corresponding *i_CPUsystem* block instance name is automatically highlighted in the design browser frame).

   By default, the instantiation of the previous block (*i_CPU*) is highlighted in the source code.

3. Double-click *i_CPU* to return the source code context to the *CPU* block.
4. You can also click the right mouse button to access a menu with the **Show Calling** and **Show Definition** commands to display the calling or definition of the block.

Find Scope

1. To locate a scope, choose the **Source -> Find Scope** command (or bind key "S").

   A *Find Scope* form displays, similar to the following example:

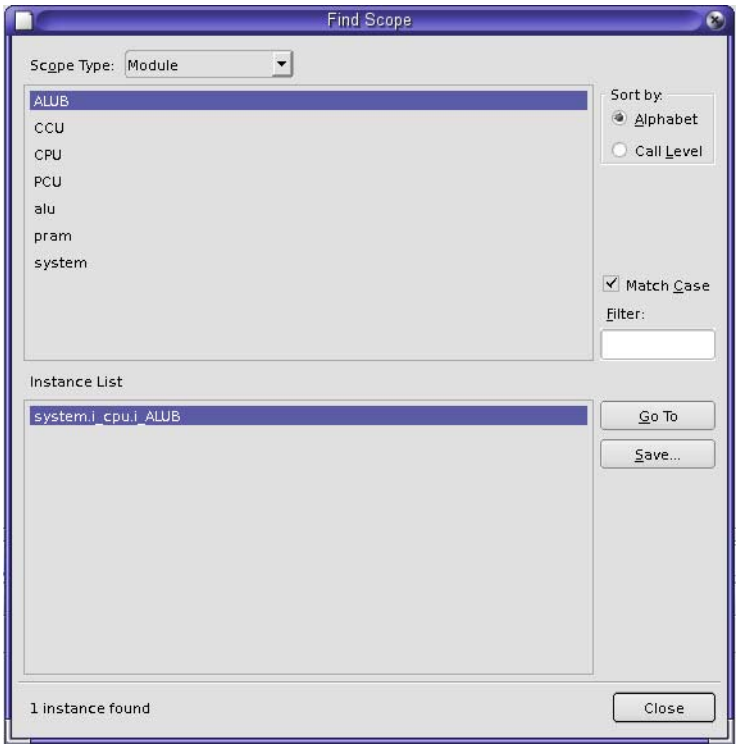


Figure: Find Scope Form

2. Enter **CU** in the **Filter** box and press **Enter** on the keyboard.

   The top frame updates to display the modules matching the search string.

3. Left-click to select *PCU*.

   The bottom frame lists all hierarchical paths for the module. In this case there is one path.

4. Click **Go To** to locate the associated module in the *Source Code* frame.

Trace Drivers and Loads

The **OneTrace -> Driver** and **OneTrace -> Load** commands (or their equivalent toolbar icons) trace all of the drivers and loads, respectively, that are associated with a selected signal. The **OneTrace ->Connectivity** command (only found in the menus) traces drivers and loads simultaneously. These commands can also be accessed by right-clicking a signal in the *Source Code* frame.

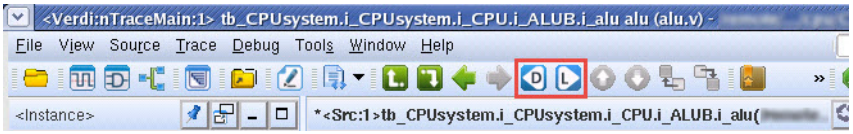


Figure: Trace Driver/Load Icons in nTrace

Find String

1. In the *nTrace* main window, double-click *i_CPU* in the design browser frame to display the associated source code.
2. To find a certain string to trace, choose the **Source -> Find String** command (or bind key "/").

   A *Find String* form displays, similar to the following example:

Figure: Find String Form

3. Enter *data* in the **Pattern** box.
4. Click **Find All** and then click **Close**.

   The results are displayed in the **Search** tab of the message frame.

5. Double-click the result *cpu.v(53):inout[7:0]data;* to highlight the associated line in the source code.

NOTE: The **Source -> Find Signal/Instance/Instport** command can also be used to locate a scope or a signal name anywhere in the hierarchy.

Trace Driver

As a result of **Find String**, you can view an 8-bit group of inout signals called data:

```
    :
inout [7:0] data;
    :
```

1. To begin the trace, double-click *data* or click the **Trace Driver** toolbar icon  or either of the following:
   - From the main menu, choose the **OneTrace -> Driver** command.
   - Or right-click the signal, and choose the **Trace Driver** command from that menu.

     The source code displays immediately changes to the pram block and highlights the signal data in the driving statement, as follows:

     ```
      :
     assign data = R_W ? dataout :8'hz;
      :
     ```

     Drivers are indicated in the *Source Code* frame with left-handed semi-circle next to the line number. The message frame also displays all of the drivers of the selected signal including any pass-throughs (the term *"pass-throughs"* refers to any intermediate nets on the path between the driver and the load as the path passes through different hierarchical levels in the design).

   The **Show Previous in Hierarchy** toolbar icon  is now enabled.

   If other drivers exist in the same hierarchy, the **Show Next** or **Show Previous** toolbar icons  may also become enabled.

NOTE: The icons that are enabled are all dependent on the results of a trace.

2. Click the **Show Previous in Hierarchy** icon to go to the *ExtData* driver in the *PCU* block.

   The **Show Previous in Hierarchy** toolbar icon  is now disabled, and the **Show Next in Hierarchy** toolbar icon is enabled.

3. Click the **Show Next in Hierarchy** icon to return to the *data* driver in the *pram* block.

Add Bookmarks

You can add a bookmark to any line number to mark it for future reference. To add a bookmark from **Trace Driver** results:

1. Click line 31 in *i_pram* where *data* is assigned.
2. Click the **Set/Unset Bookmark** icon  on the tool bar.

   The bookmark symbol appears, as shown in the following figure:

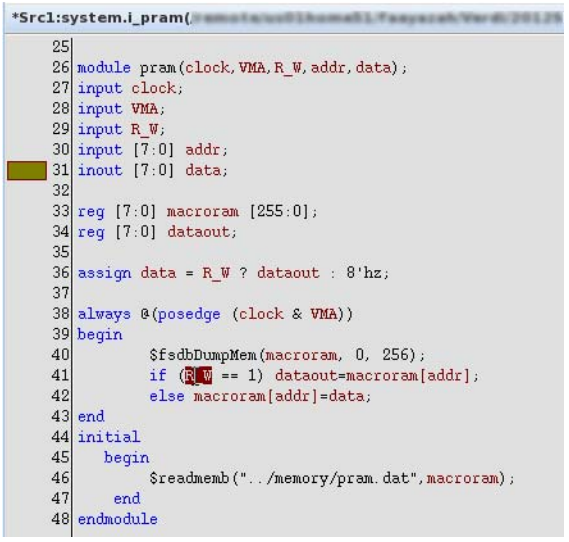


Figure: Example of a Bookmarked Source Code Line

Trace Load

1. To locate the first load on this net, highlight *data*, and use the **Trace Load** toolbar icon  .
   - You can also choose the **OneTrace -> Load** command from the main menu.
   - Or right-click the signal, and choose the **Trace Load** command from that menu.

   Loads are indicated in the *Source Code* frame with right-handed semi-circle next to the line number. The message frame also displays all of the loads of the selected signal including any pass-throughs (the term *"pass-throughs"*

refers to any intermediate nets on the path between the driver and the load as the path passes through different hierarchical levels in the design).

The source code display changes to the pram block and highlights the signal data in the loading statement as follows:

```
      :
macroram[addr]=data;
      :
```

Trace Connectivity

1. To find a bookmarked line, choose the **Source -> Manage Bookmarks** command, then select the line from the list that displays (**1. pram.v(41)**).
2. To trace the connectivity of a signal, highlight a signal, and choose the **OneTrace -> Connectivity** command from the main menu or the right mouse button menu.

   The results of the trace are displayed in **OneTrace** tab of the message frame in the *nTrace* main window, similar to the following example:

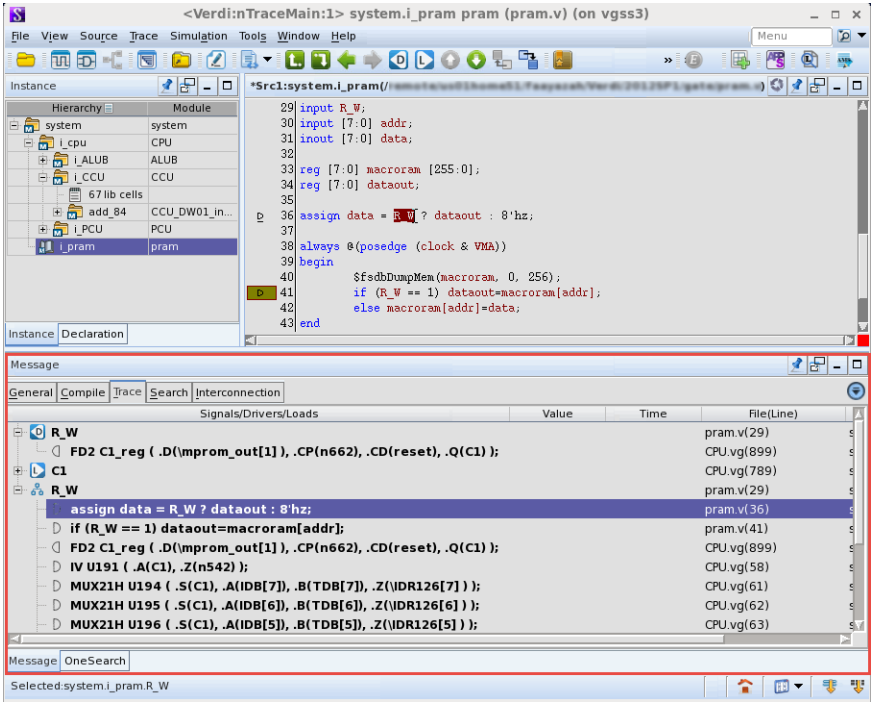


Figure: Example of nTrace Window With Trace Connectivity Results

3. The **Show Next** toolbar icon is now enabled. Click this icon to locate the next load (this is equivalent to using the **Trace -> Show Next** command).
4. Note that the **Show Previous** toolbar icon is now enabled. Click this icon to locate the previous load/driver (same as using the **Trace -> Show Previous** command).

Save Trace Result and Reset History

There are two methods for saving the trace results.

1. Use the **Save** command:
   a. On the **OneTrace** tab of the *Message* frame, click the right mouse button and choose the **Save** command.
   b. Enter the file name and click **OK**.
2. Use the **Save** icon:
   a. On the tab bar of the *Message* frame, click the **Show Toolbar** icon . A *Message* frame toolbar is displayed as shown in the following figure.

   b. Click the **Save** icon.
3. To reset the history of all signals, choose the **Trace -> Reset History** command from the main menu.

Edit Source Code

After you have located the source code which needs to be edited, you can modify the source code from the *nTrace* main window. After the code is modified you need to re-compile and load the design and also re-generate the simulation results file.

1. In the *nTrace* main window, choose the **Tools -> Preferences** command to open the *Preferences* form.
2. Expand the **Editor** folder.

3. Select the page for the editor, for example, **nEditor**, and turn on the **Set as Default Editor** option.

NOTE: If the editor you require is not listed, you can specify your own edit command on the **Other** page.

4. Click **OK**.
5. In the *nTrace* main window, double-click *i_ALUB* in the design browser frame to display the associated source code.
6. Click the **Edit Source File** icon  to open a second source viewer. Alternatively, choose the **Source -> Edit Source File** command in the *nTrace* main window.

Use Active Annotation

Active Annotation™ allows you to view your verification results in the context of the source code. Active Annotation allows you to view - in one place - the value resulting from a logic expression coupled with the values of the arguments feeding that expression.

NOTE: Active Annotation can also be used to display verification results in other views.

Before using Active Annotation, you must first load a set of simulation results in the form of a FSDB.

NOTE: Other formats can be loaded and are automatically converted.

1. Load the simulation results using the **File -> Open Waveform File** command in the *nTrace* main window which opens the *Load Simulation Results* form.
2. In the *Load Simulation Results* form, move up one directory from the current directory.
3. Select *CPUsystem.fsdb*.
4. Click **OK** to load the file.
5. In the design browser frame, double-click *i_ALUB*.
6. Choose the **Source -> GoTo -> Line** command, and enter *85*.
7. Click **OK**.

8. Choose the **Source -> Active Annotation** command (or “x” key after putting the cursor in the *Source Code* frame) to activate Active Annotation. The values associated with each signal (as the time 0) are displayed under the signals and a new sub-toolbar appears, as shown in the following figure:

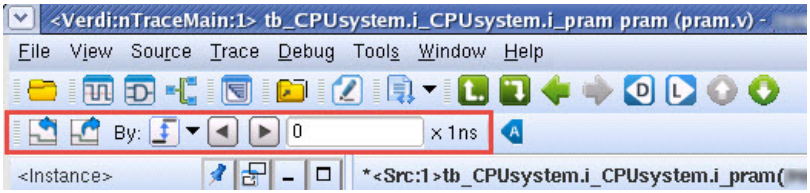


Figure: Sub-toolbar on nTrace Window Main Toolbar

NOTE: The *XX* Active Annotation symbol shown under the signals reflect the un-initialized condition of these signals at time 0.

9. Select the *RESET* signal, and click the **Search Forward** and **Search Backward** icons on the new Active Annotation sub-toolbar. Note that the display updates to reflect the transitions from value to value at the time in which it occurs.
10. Search for rising edge changes on the *RESET* signal by changing the search **By** selection to **Rising Edge** By: and continue to click the **Search Forward** and **Search Backward** icons.
11. On the toolbar, choose the **Source -> Go To -> Line** command.
12. In the *Go To Line* form, enter *82* and click **OK**.
13. On the toolbar, enter 777 in the **Cursor Time** box.
14. Press the <**Enter**> key on the keyboard.

    The *nTrace* main window updates the display similar to the following figure:

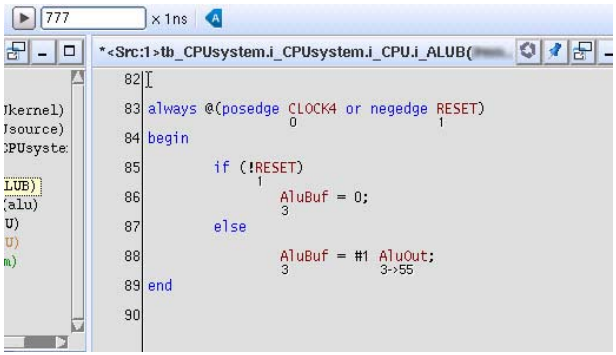


Figure: Active Annotation

All the signals in the design are assigned to non-*X* values.

Trace the Active Driver

1. Using the results from the previous section, go to line *44*, and select *AluOut*.
2. Right-click, and choose the **Active Trace** command from the menu (or the bind key, Ctrl-t).

   The source code displays changes to the active driving unit, and the signal in the driving statement is highlighted, as shown in the following figure:

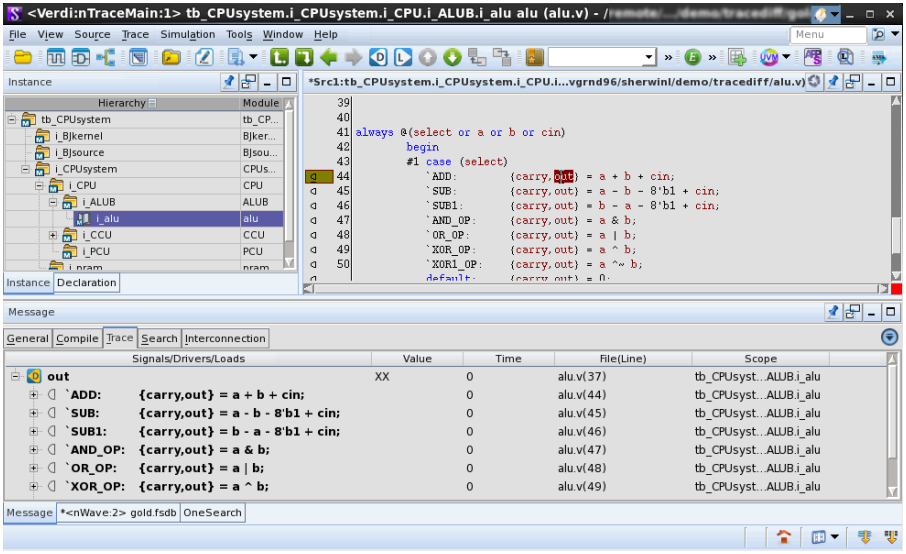


Figure: Active Trace

   AluOut changed names to *out* as it crossed hierarchy boundary. See the message frame in the figure above for example.

   The time field in the toolbar may also change to reflect when this assignment to the signal is made. This information is also presented in an *Information* dialog window.

3. Check the equation, and note that 55 appears from signal *a*.
4. Select *a*, and choose **Active Trace** again.

   You can continue to Active Trace until you locate the source of a value.

Use Verdi Executable to Import Design from UFE

The Unified Compiler Flow (UFE) uses VCS with the `-kdb` option and the generated `simv.daidar` includes the KDB information.

Verdi supports the following use models to import design from `simv.daidir` related files:

- "verdi -dbdir *simv.daidir*"
- "verdi -simflow -simBin *simv*"
- "verdi -simBin *simv*"
- "verdi -ssf *novas.fsdb*"

The priority to import design from UFE is as following (from high to low):

1. Use the new `-dbdir` option to specify the `simv.daidir` directory

   Similar to: `dve -dbdir simv.daidir`

2. Use the `-simflow -simBin` options to specify simv

   Load `simv.daidir` from the same directory as simv. An error is reported if `simv.daidir` is not available.

NOTE: The import options `-top, -elab, -lib, -f,` and `-ridb` cannot be specified for this usage.

3. Use the `-simBin` option to specify simv

   Load `simv.daidir` from the same directory as simv and invoke Verdi if `simv.daidir` is not available.

4. Use the `-ssf` option to specify FSDB file and the design is automatically loaded.

   Load KDB automatically from FSDB. The FSDB file needs to be generated from “vcs -kdb” with VCS K-2015.09-SP2 or later.

NOTE: The FSDB or all FSDB files in virtual FSDB should be generated by the same simv that is generated by VCS with the `-kdb` option.

The import options `-top, -elab, -lib, -f,` and `-ridb` cannot be specified for this usage.

nSchema Tutorial

Overview

nSchema is a schematic viewer and analyzer that generates interactive debug-specific logic diagrams displaying the structure of the selected portions of a design. RTL diagrams show the interconnection of finite state machines, storage elements, and multiplexers. Gate-level diagrams show the interconnection of semiconductor vendor cells and special flattened diagrams cut through the design hierarchy to isolate connected design elements. *nSchema* dynamically generate partial schematics to focus on the circuits of interest in a large design.

Before you begin this tutorial, follow the instructions in the *Before You Begin* chapter. Refer to the *User Interface* chapter for general information on the *nSchema* window.

The *nSchema* window is used to display auto-generated schematics and logical diagrams, the frame can be undocked to be a standalone *nSchema* window, as shown in the following example:

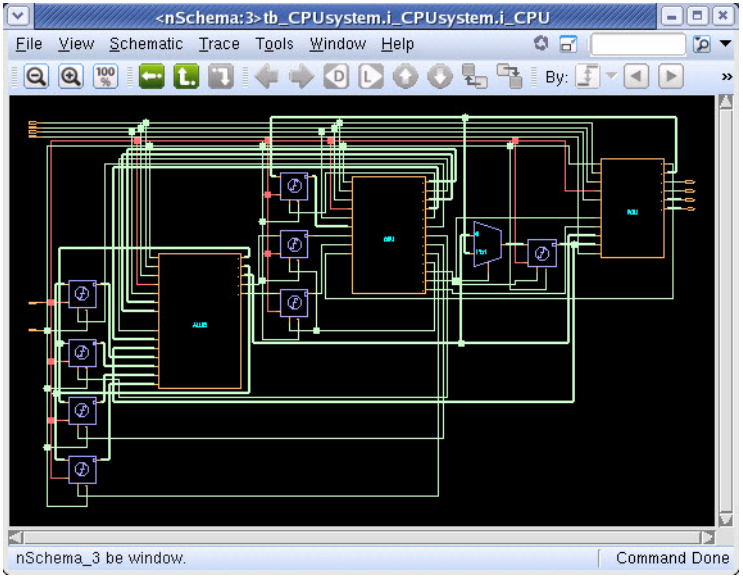


Figure: Example of nSchema Window

nSchema generates the schematic for both RTL and gate-level designs. For RTL designs, the Verdi platform extracts certain types of synthesizeable function blocks from the HDL code, such as registers, latches, multiplexers, pure combinatorial or sequential circuits, and so on. With this capability, you can get a clear picture of the design intent, especially for a design with which you are unfamiliar. For gate-level designs, the Verdi platform uses standard symbols, such as nand, nor, inverter, and so on, to make the schematic more readable and understandable. To perform certain functions, such as signal tracing or intuitive searching, you can drag-and-drop items between windows to cross-link the tools.

Start nSchema

1. Change the directory to <*working_dir*>*/demo/verilog/cpu/src*, and execute the following command to import the sample CPU design:

   ```
   % verdi -f run.f -workMode hardwareDebug &
   ```

   You can continue from the previous Verdi session if the window is still open.

2. In the *nTrace* main window, highlight the folder *tb_CPUsystem* in the **Instance** tab of the design browser frame.
3. Click the **New Schematic** icon  on the toolbar in the *nTrace* main window (or choose the **Tools -> New Schematic from Source -> New Schematic** command), the *nSchema* window is displayed. Click the **Undock** icon in the upper right of the *nSchema* window, a separate window showing the schematic of the current module (*tb_CPUsystem)* displays, as follows:

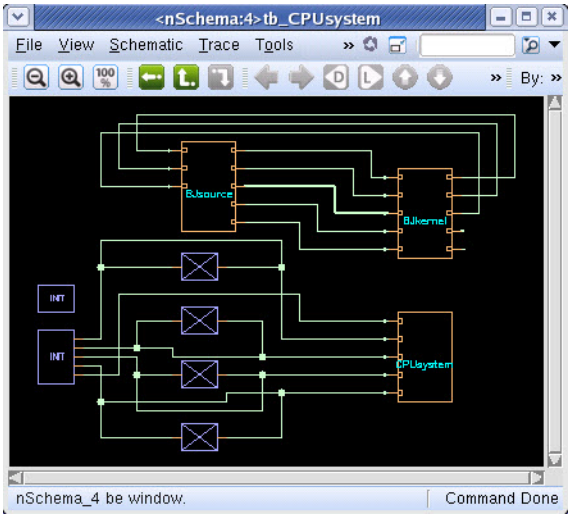


Figure: nSchema Window Displaying tb_CPUsystem Schematic

Click the **New Schematic** icon to create a new schematic frame in full hierarchical view. Each new schematic frame initially shows the schematic view of the HDL source module currently displayed in the source code frame.

4. Drag-and-drop the instance, *i_CPUsystem*, in the design browser frame to the separate schematic window to display the instance's schematic.

   The results are displayed similar to the schematics displayed in the following figure:

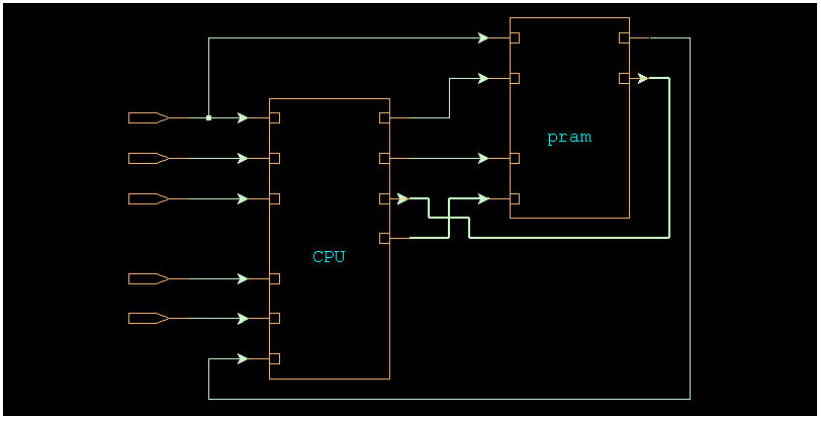


Figure: Displaying CPU Block as Schematic

Manipulate the Schematic View

Double-click the *i_CPUsystem, i_CPU* and *i_CCU* module names in the design browser frame to access the source code of the module *CCU*.

You can change the view of the schematic using the following zoom commands:

- **Zoom In**  - View more details of the schematic by moving the view 50% from the center point in both the horizontal and vertical directions. Invoke this command in one of following three ways:
  - Toolbar icon
  - Bind key "Z"
  - Menu **View -> Zoom -> Zoom In** command
- **Zoom Out**  - View more contents of the schematic by expanding the view 2X from the center point, both horizontally and vertically. Invoke this command in one of the following three ways:
  - Toolbar icon
  - Bind key "z"
  - Menu **View -> Zoom -> Zoom Out** command
- **Zoom All**  - View the entire contents of the schematic. Invoke this command in one of the following three ways:
  - Toolbar icon
  - Bind key "f"
  - Menu **View -> Zoom -> Zoom All** command
- **Zoom Area** - View more details in a specific area of the schematic by dragging-left to form a rectangle over the area.

NOTE: You can change if the right mouse button or the left mouse button performs the zoom on the **General** page under the **General** folder of the *Preferences* form (invoked with the **Tools -> Preferences** command). This example assumes that the left mouse button is set to zoom.

You can move the viewing area of the schematic in different directions:

- **Scrolling** - Click or drag the scroll bar of the schematic window horizontally or vertically.
- **Panning** - Move the viewing area up, down, left, or right using the arrow keys on your keyboard or the menu commands: **View -> Pan -> Pan Up**, **View -> Pan -> Pan Down**, **View -> Pan -> Pan Left**, and **View -> Pan -> Pan Right**.

In addition, you can use the **View -> Last View** command or the bind key "l".

5. In the *nSchema* window, choose the **Tools -> New Schematic -> Current Scope** command from the main menu to create a new *nSchema* window with the same schematic view.
6. To close schematic windows that are no longer required use the window manager's control button (the 'X' icon in the upper right) or the menu **File -> Close Window** command.

Change the Schematic View Among Instances

1. Using the middle mouse button, drag the instance *i_ALUB* from the design browser frame and drop it into the *nSchema* window to show the corresponding schematic.

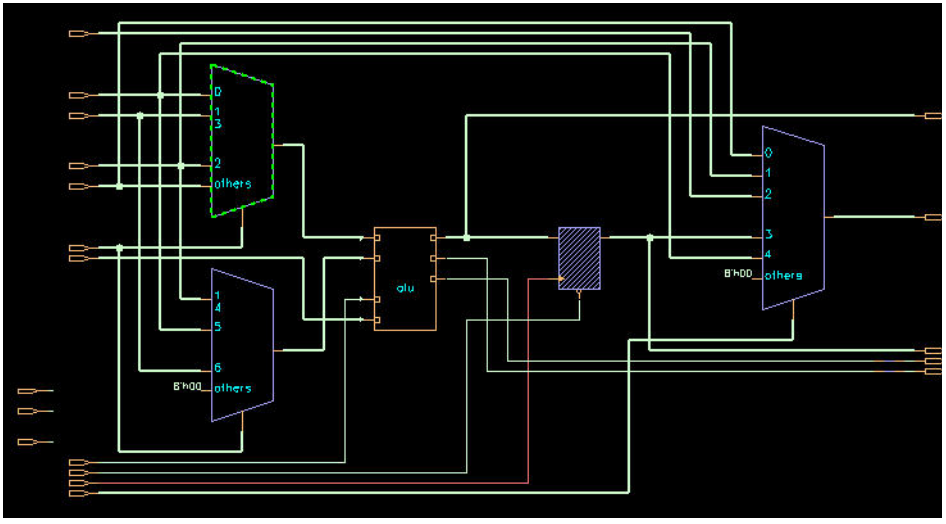


Figure: Displaying i_ALUB Schematic

2. Move your cursor over various symbols or nets in the schematic view, and notice the name is identified in *nSchema*'s lower bar. The name information is shown in the lower bar of the main window if the *nSchema* windows is not undocked as a standalone window.
3. Right-click the schematic window, select **Pop View Up** in the shortcut menu and the schematic view changes from the child module *ALUB* to its parent module *CPU* with *ALUB* block highlighted as the selected object.
4. Choose the **View -> Push View In** command or its corresponding toolbar icon  to update the schematic view to the selected module *ALUB*.
5. Choose the **View -> Pop View Up** command or its corresponding toolbar icon  to change the schematic view to module *CPU*.

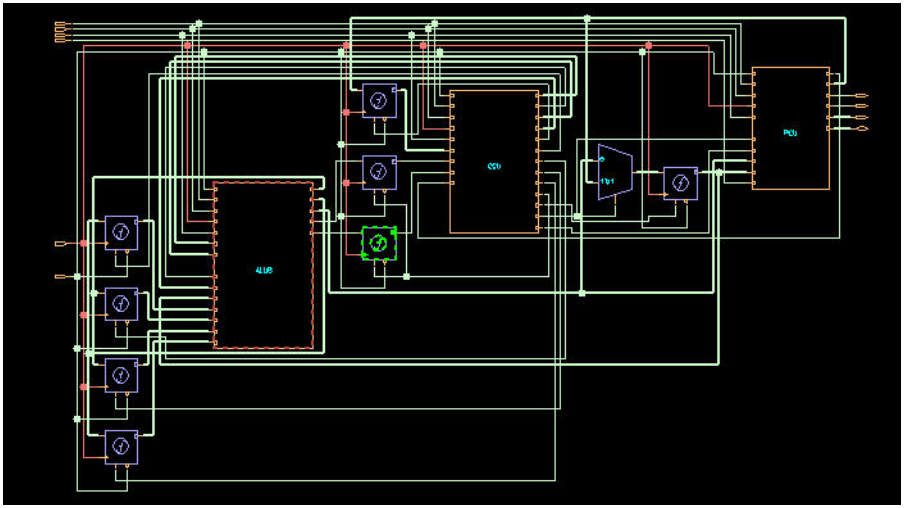


Figure: Displaying CPU Module With ALUB Block Highlighted

6. Choose the **View -> Last View** command or the bind key **L** or its corresponding toolbar icon  to roll back the schematic view to module *ALUB*.

Enable Viewing Objects

You can enable or disable viewing for different objects (for example, nets, instances, ports, and so on) in the schematic window.

1. In *nSchema*, choose the **Tools -> Preferences** command to open the *Preferences* form.
2. Select the **Color/Font** page under the **Schematics** folder.
   a. Change the **Type** field from the default **Background** selection to **Selected Set**.
   b. Change the **Color** to red and the **Line Style** to dashed.

   Notice the changes that affect the open schematic.
3. Select the **View** page under the **Display Options** folder and turn on the **Local Net Name**, **Instance Name**, and **IO Port Name** options.
4. Click **OK** to close this form and apply the changes.
5. In the *nSchema* window, open the **View** menu and select the **Net Name**, **Instance Name**, and **IO Port Name** commands individually. The net, instance and port names are removed from this schematic.

The options on the *Preferences* form affects all schematics - a global setting. The **View** menu only affects the current schematic - a local setting.

Find an Instance or a Signal in a Schematic

1. Display the *i_ALUB* module in the *nSchema* window and then drag-left around the multiplexor in the upper left corner to zoom in.
2. Choose the **Schematic -> Find in Current Scope** command (or “a” bind key) to display the *Find* form as shown in the following figure:

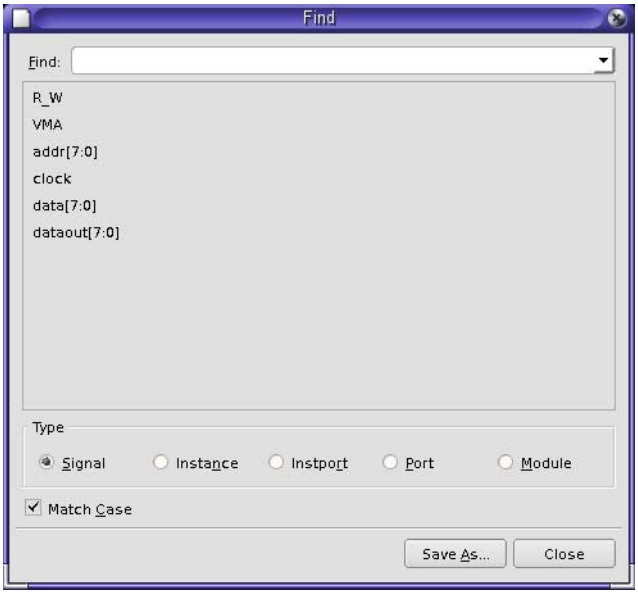


Figure: Find Form

Select the **Instance** option or the **Signal** option to list all the instances or signals under the current module in the alphabetical order.

NOTE: If the **Schematic -> Auto Fit Found Object(s)** toggle command is enabled or the **Auto Fit Selection** option is enabled on the **Schematics -> Select** page of the *Preferences* form (invoked with the **Tools -> Preferences** command) and a target instance/signal is selected, its corresponding object is immediately selected in the schematic window and the schematic view scales properly to make the target object viewable.

---

3. Select the **Signal** option, and uncheck the **Match Case** option.
4. In the **Find** text field, enter *alu** and press <**Enter**>.
5. Select *AluBuf[7:0]*.

The signal is highlighted in the schematic view, as shown in the following figure:

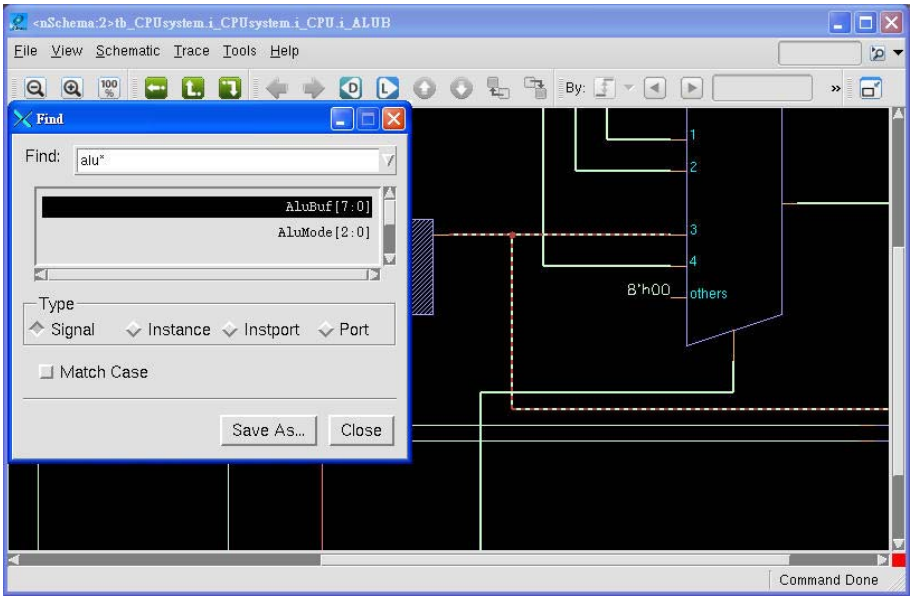


Figure: Find Form With Highlighted Signal

6. Click **Close** to close the form.
7. In *nSchema*, double-click the multiplex or instance attached to the *AluBuf* signal. A *View Source Code* window opens displaying the associated source code lines.
8. With the same multiplex or instance selected, use the middle mouse button to drag the instance to the *Source Code* frame to display the source code in context.
9. In the **Instance** tab of the design browser frame, double-click *i_CCU* to locate the source code.
10. Locate the signal declaration for *IXR_load* in the *Source Code* frame.
11. From the *Source Code* frame, drag-middle and drop the signal in any open *nSchema* window.

    The schematic is updated and the signal is highlighted.

Change the Color of the Selected Signal

1. To identify certain signals in a schematic clearly, choose the **Schematic -> Change Color** command.

   A *Change Selection Color* form displays the selected signal name and its color, as shown below:

Figure: Change Selection Color Form

2. Click a color in the color map to change the *FirstDataInRdy* signal's color instantly. Click **Default** if you want to reset a signal color.
3. Click **Close** to close the form.
4. To return everything to the default colors, in the *nSchema* window, choose the **Schematic -> All Objects to Default Color** command.
5. Close all open schematic windows.

Trace Signals

The following sections are based on the schematic of *i_CPUsystem*. Before you begin, double-click *i_CPUsystem* in the design browser frame, and click the **New Schematic** icon to display the schematic. Then click the **Undock** icon in upper right to open a standalone *nSchema* window.

Find the Drivers of a Signal

The **Trace Driver** function shows all drivers of a selected signal on the schematic.

1. For example, choose the **Schematic -> Find in Current Scope** command to locate and select *data[7:0]* in the schematic window.
2. Choose the **Trace -> Driver** command (or click the icon) on the selected bus, *data[7:0]*. The result is shown in the following figure:

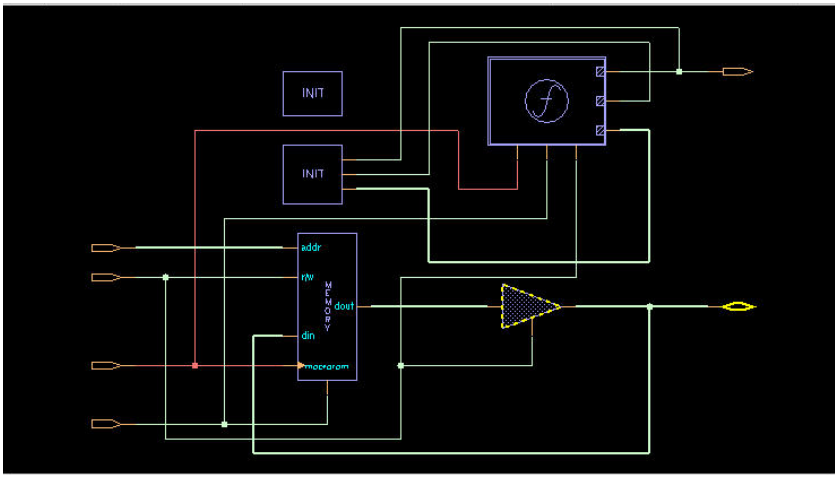


Figure: Example Results of Trace Driver on data[7:0]

The figure above shows a driver found in module *pram*. Since no other drivers exist in that module, the **Show Previous** and **Show Next** icons are disabled. You can access the **Show Next in Hierarchy** icon because drivers exist in other modules.

3. To show the schematic of *PCU* and the traced drivers in that module, click **Show Next in Hierarchy**. The **Show Previous in Hierarchy** becomes enabled.
4. To return to the schematic view of module *pram*, click **Show Previous in Hierarchy**.

Find the Load of a Signal

The **Trace Load** function shows all loads of a selected signal or the schematic.

1. Use the pop view up icon to go back to the *CPUsystem* schematic.
2. Highlight *data[7:0]*.
3. Choose the **Trace -> Load** command or the toolbar icon  .

Find the Connectivity of a Signal and Generate a New Schematic from Trace Results

To narrow the debugging scope, you can generate a partial schematic containing only the trace results.

1. Use the pop view up icon to go back to the *i_CPUsystem* schematic.
2. In the *i_CPUsystem* schematic, choose the **Trace -> Connectivity** command on the selected bus, *data[7:0]*. The schematic updates with the highlights of the trace results.
3. Choose the **Tools -> New Schematic -> From Trace Result** command to create a new schematic frame with only the trace results. Click the **Undock** icon in upper right to make the frame a standalone *nSchema* window.

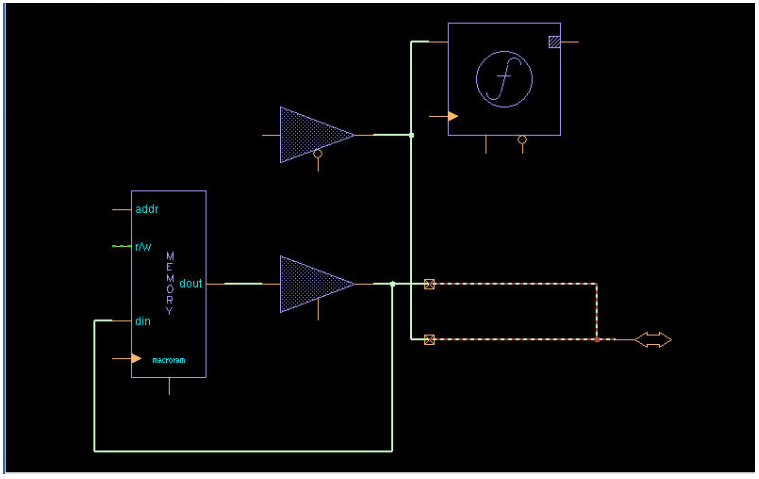


Figure: Displaying Trace Results for data[7:0] as a Schematic

4. Close all open schematic windows.

Show RTL Block Diagram in a More Meaningful Way

The Verdi platform can recognize some specific hardware elements and display them using meaningful RTL block-diagram symbols. See *Appendix C: Enhanced RTL Extraction* for a complete list of symbols. When you want to see the boolean equivalent views, perform the following steps:

1. Open the *i_CCU* block in *nSchema*:

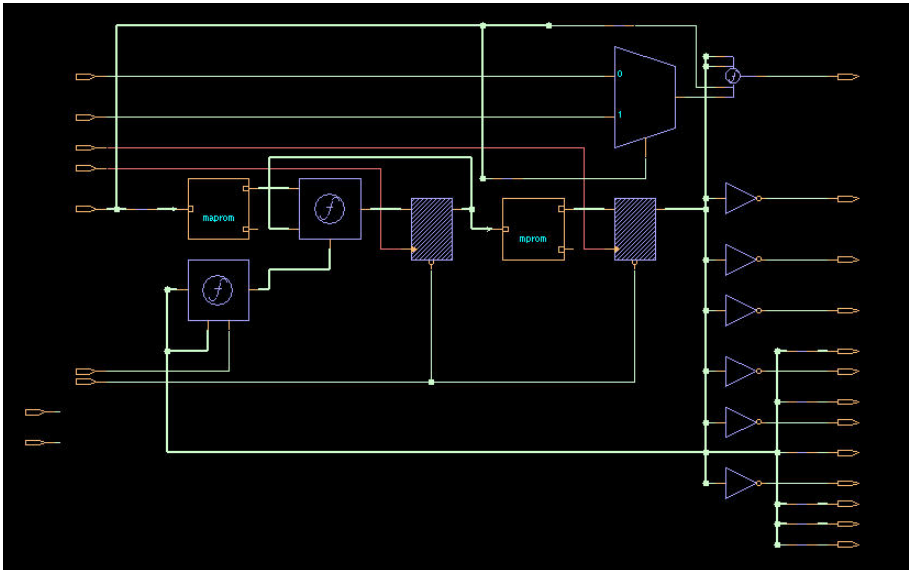


Figure: Displaying RTL Block as a Schematic

   Note that the function symbol is in the upper right side.

2. Double-click the function symbol to see the associated source code (or drag to the source code frame and drop).
3. To enable the detailed RTL view, choose the **Tools -> Preferences** command to open the *Preferences* form.
4. On the *Preferences* form, select the **RTL** page under the **Schematics** folder and then turn *on* the **Enable Detailed RTL** option.
5. Click **OK**.

The results are displayed similar to the schematics displayed in the following figure:

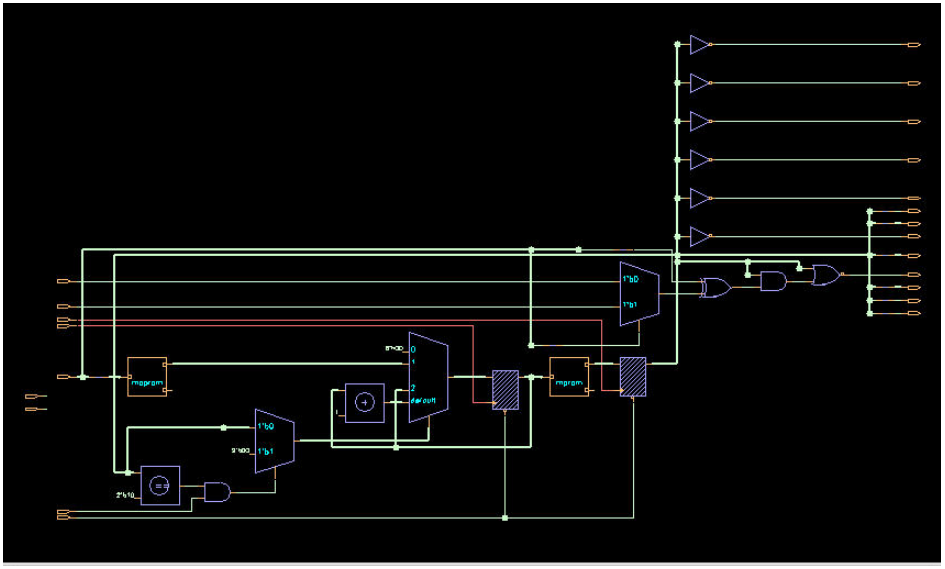


Figure: Displaying RTL Block in Detail

The function symbol is gone and has been replaced with an *xor*, *and*, and *nor* gate.

NOTE: The preferences affect **all** windows. Use the **View -> Detail RTL** command to only change **one** window.

Generate Partial Schematics

Hierarchical

Often, the top level block diagram is too cluttered and you want to be able to focus on a couple of blocks or signals.

1. In the design browser frame, select *i_CPU*, and open a new schematic frame.
2. In the *nSchema* frame, select the *ALUB* block.
3. Press the <Shift> key and, select the *PCU* block. (The same thing can be accomplished with multiple nets.)
4. Choose the **Tools -> New Schematic -> Browser Window** command.

   A new schematic frame with the selected block and connections is displayed. Click the **Undock** icon in upper right to create a standalone *nSchema* window, as shown below:

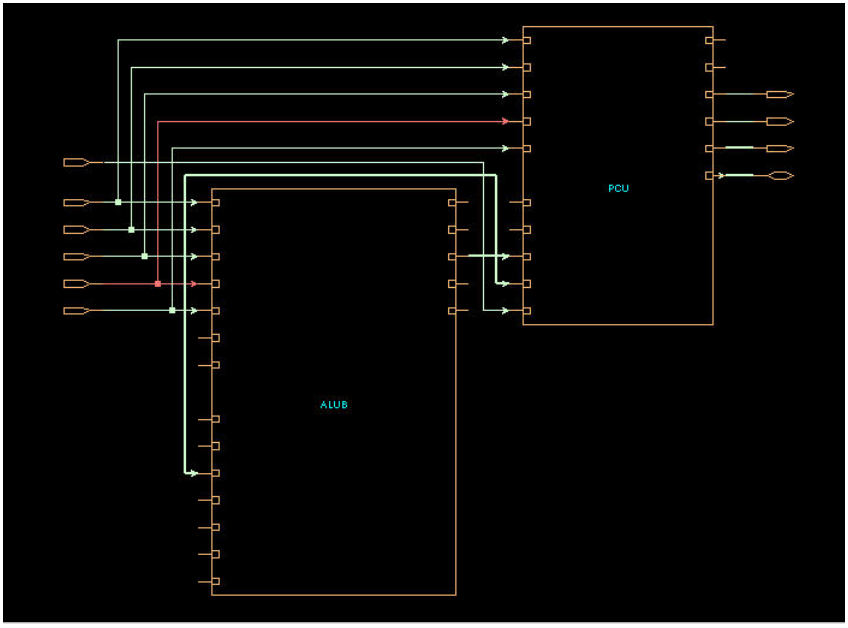


Figure: nSchema Partial Hierarchical View

The *CCU* block is not included in the above window.

5. Double-click the *ALUB* block and notice that there is a reduced logic display and therefore a partial hierarchical view.

6. Go back to the previous hierarchy using the **Pop View Up** icon.
7. Choose the **View -> InstPort Name** command to annotate the instance port names on the schematic.
8. Double-click the *AluOut[7:0]* port on the *ALUB* block (upper right) to expand the connecting logic. (Any port can be expanded).
9. Click the **Undo** toolbar icon  to get back to the previous view.
10. In the *nTrace* main window, choose the **File -> Exit** command to close the Verdi session.

Flattened Window

The following topics work equally well for RTL designs, a gate-level design is more interesting.

1. Before you start, close the current Verdi session, and change the directory to *<working_dir>/demo/verilog/gate.*
2. Set the environment variables:

```
% setenv NOVAS_LIBPATHS $VERDI_HOME/share/symlib/32
% setenv NOVAS_LIBS lsi10k_u
```

3. Invoke the Verdi platform:

```
% verdi -f run.f -workMode hardwareDebug
```

4. In the **Instance** tab of the design hierarchy frame, expand *system*, *i_cpu*, and then double-click *i_ALUB* to display the source code.
5. In the **Find String** box on the toolbar, enter *U250.*
6. Click the **Find Next** icon  to find the instance on line 639.
7. Choose the **Tools -> New Schematic from Source -> Flattened Window** command.

   The associated NAND gate is displayed in *nSchema*. Click the **Undock** icon is the upper right to change the *nSchema* window to a standalone window.
8. Double-click the output port to expand the loading logic.

   The symbol of a box with a cross in it indicates a crossed hierarchy 

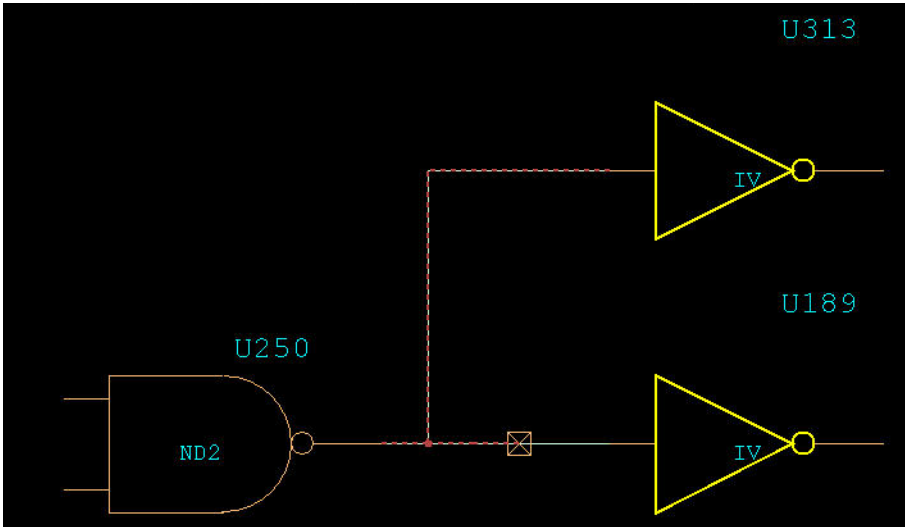


Figure: nSchema Expanded Logic

9. Choose the **View-> Instance Name** command.
10. Select the instance *U313*, and click the **Remove** icon to delete the gate from the view.
11. Click the **Undo** icon to add it back to the schematic.

Fan-in and Fan-out

The Fan-in and Fan-out functions automatically generate the fan-in or fan-out cones for the selected instance or net.

1. Continue from the gate design.
2. Select *U250* in the *Flattened Window* schematic window.
3. In the *nSchema* window, choose the **Tools -> New Schematic -> Fan-in** command. Click the **Undock** icon in upper right to create the *View Trace Fan-In Cone Result* window from the *nSchema* window.

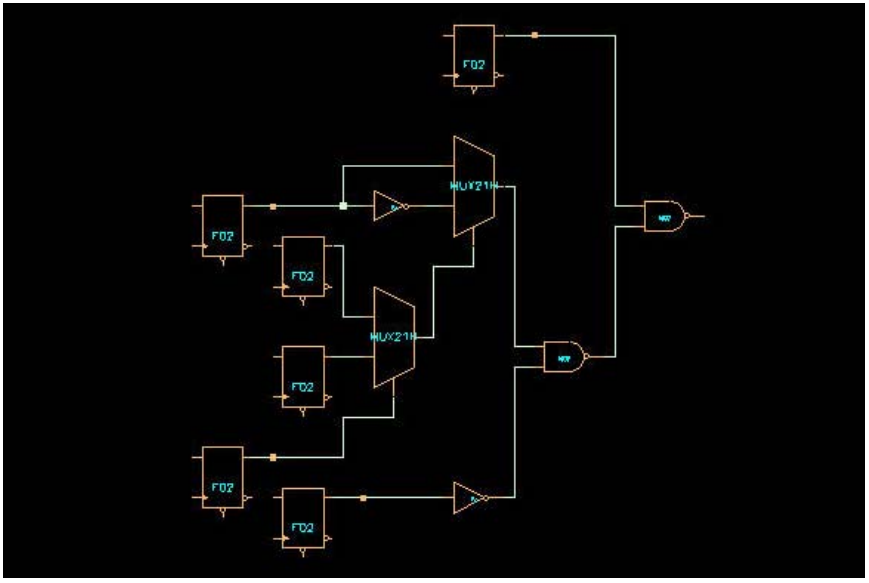


Figure: nSchema Displaying Fan-in Cone

Several hierarchies are represented and tracing automatically stops on storage elements.

4. In the *Source Code* frame, select *U251*, line 640, of *i_ALUB*.
5. Use the middle mouse button to drag *U251* to the *View Trace Fan-In Cone Result* schematic view, and drop it.

   Note that *U251* is not connected to any of the existing logic.
6. Change the design browser frame to *i_CCU*, and open the source code.
7. Find *U248* on line 832.
8. Use the middle mouse button to drag *U248* to the *View Trace Fan-In Cone Result* schematic view, and drop it.

   Note that it automatically connects to an existing storage element.

   In partial flattened schematics, you can easily add logic by double-clicking to expand ports or dragging and dropping instances or nets from other windows.

Trace Between Two Points

Tracing between two points isolates connecting logic between two storage elements and display the results.

1. Close the existing *nSchema* windows.

2. In the *nTrace* main window, choose the **Source -> Find String** command.
3. In the *Find String* form, enter *IDR_reg* in the pattern box.
4. Turn on the **In All Files** option.
5. Click the **Find All** button. The results are listed in the **Search** tab of the message frame.
6. Double-click the text line associated with *IDR_reg[7]*.
7. In the *nTrace* main window, click the **New Schematic** icon to open the *nSchema* frame. Click the **Undock** icon on upper right to change the *nSchema* window to a standalone *nSchema* window.
8. Drag the instance name *\IDR_reg[7]* from the *Source Code* frame and drop to the *nSchema* window. The instance is highlighted.
9. Zoom in around the highlighted instance by dragging the left mouse button over the area.
10. Choose the **Trace -> Two Points** command. The *Trace Two Points* form is displayed as follows:

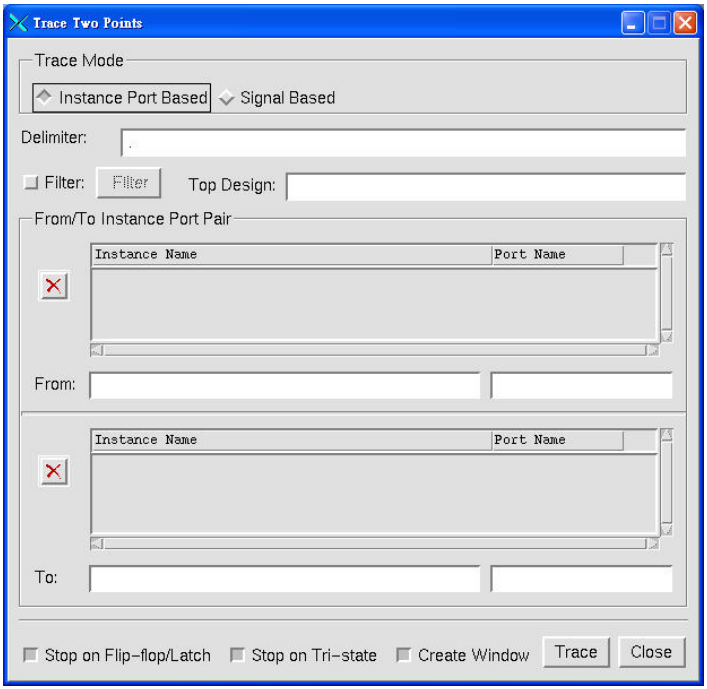


Figure: Trace Two Points Form

11. Use the middle mouse button to select the *Q* output port of *\IDR_reg[7]*, and then drag and drop it in the **From** box of the form.
12. Move the form out of the way while you find the next point.

13. In the *nTrace* main window, search for *IXR_reg* using **Find String**.
14. Double-click the text associated with *\IXR_reg[7]* in the **Search** tab of the message frame.
15. In the *Source Code* frame, drag and drop *\IXR_reg[7]* to the **To** box of the *Trace Two Points* form, and select *D* in the *Port name* column.
16. In the *Trace Two Points* form, confirm the **Create Window** option is enabled (depressed), and click **Trace**.

    The results are displayed in a new *nSchema* window, click **Undock** icon to create the *View Trace Result* schematic window:

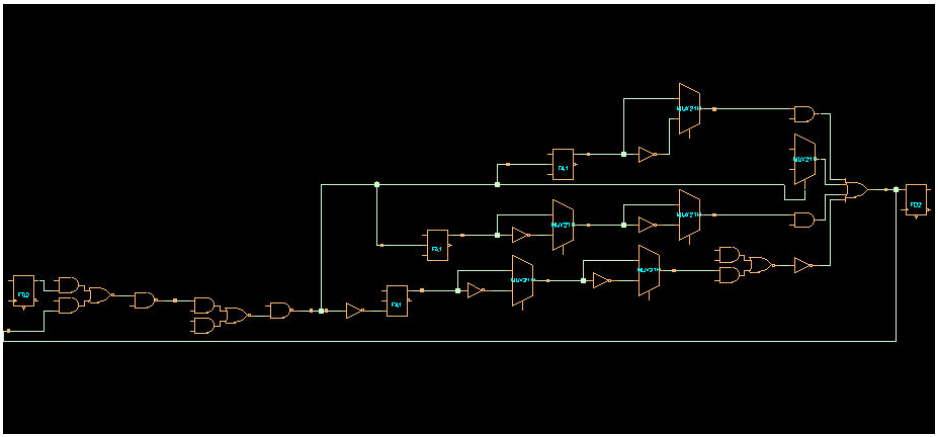


Figure: Results of Trace Two Points

Use Active Annotation to Show Signal Values

The **Schematic -> Active Annotation** command allows you to display signal values from simulation directly on the schematic. The display style can appear in either text value or line coloring (for values 0, 1, z, and x only).

1. In the *nTrace* main window, choose the **File -> Load Simulation Results** command to load the *gate.fsdb* results file.
2. In the *nTrace* main window, select the *i_PCU* unit in the **Instance** tab of the design browser frame, and open a new schematic.
3. Zoom in around the upper right corner.
4. In *nSchema*, choose the **Schematic -> Active Annotation** command or use the 'x' hot key to display the waveform values on the schematic.
5. Select the *ALU[7:0]* signal and use the **Search Forward** icon  on the toolbar to step through value changes. You can also **Search Backward**.
6. Add line coloring annotation by turning on the **Schematic -> Annotate in Color** toggle command.

   The definitions of the annotation text and colors are as follows:

   - 1: logic high (green)
   - 0: logic low (yellow)
   - x: unknown (red)

Change the Color or the Line Style for Annotations

1. To change the color or the line style for annotation, choose the **Tools -> Preferences** command in the *nTrace* main window to display the *Preferences* form.
2. Select the **Color/Font** page under the **Schematics** folder.
3. Click the **Type** option menu.
4. Set the color and line style (including the annotation line coloring for value 1, value 0, value x, and value z) for the list of objects displayed.
5. In the *nTrace* main window, choose the **File -> Exit** command to close the Verdi session.

nWave Tutorial

nWave is a state-of-the-art graphical waveform viewer and analyzer that is fully integrated with the source code, schematic, and flow views of the Verdi platform. A waveform search engine combined with backward and forward navigation allows you to search for signal transitions, bus values, discrepancies, or user-defined events easily. *nWave* also offers flexible signal group management, user-customizable glitch detection, a built-in logic analyzer, logical operations, events, display of delays back-annotated from SDF files, mixed analog/digital (A/D) display capabilities (including overlap, vertical zoom, delta x and y, arithmetic operations, analog- to-digital signal conversion, and others), and transaction/message display.

Before you begin this tutorial, follow the instructions in the *Before You Begin* chapter. Refer to the *User Interface* chapter for more details regarding the *nWave* interface.

Start nWave and Open a Simulation Result File

Before you start, change the directory to <*working_dir*>*/demo/verdi_mixed*, and use the following commands to import the sample CPU design:

```
% vericom -autoalias -f run_verilog.f
% vhdlcom -f run_vhdl.f
% verdi -top tb_CPUsystem -workMode hardwareDebug &
```

1. After the Verdi platform is started, click the **New Waveform** icon  on the toolbar to start *nWave* (or choose **Tools -> New Waveform** command). Click the **Undock** icon in upper right corner to change the *nWave* window to a stand-alone *nWave* window.
2. Choose the **File -> Open** command or click the toolbar icon  .

   The *Open Dump File* form displays, as shown in the following figure:

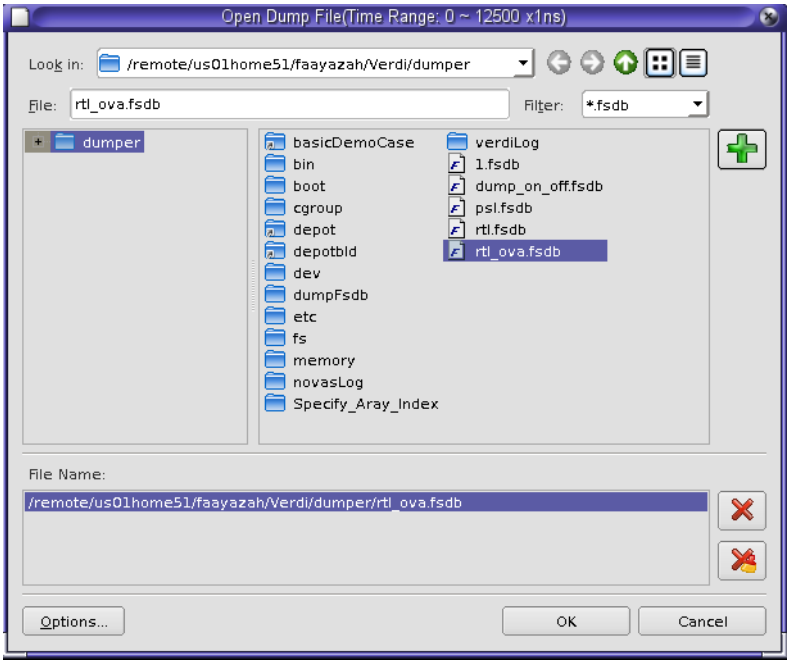


Figure: Open Dump File Form

3. Go to the directory where the FSDB file resides. In this example, it is the current directory.
4. Select the *rtl_ova.fsdb* file in the **File** list.
5. Click **Add File** or double-click *rtl_ova.fsdb* to add the simulation results file to the **File Name** section.
6. Click **OK** or double-click *rtl_ova.fsdb* to open the simulation results file.

NOTE: On the *Open Dump File* form, confirm that the **Open File by Time Range** option is not enabled. If it is enabled, a *Put in Time Range* form is displayed. If it is not enabled, the FSDB file loads into *nWave*.

You can add multiple FSDB files before clicking **OK**.

The signal pane displays one default group named G1. The signal cursor is initially located under group G1. (The signal cursor is the default location where signals are inserted.)

If you open a VCD file, the Verdi platform automatically converts it to an FSDB file and appends *.fsdb* to the file name.

Add Signals

There are two primary methods to add signals as follows:

- Drag and drop signals from other Verdi frames or windows.
- Use the *Get Signals* form to search and add signals.

Add Signals from Other Windows

Perform the following steps to add signals to the *nWave* window:

1. In the design browser frame, expand *i_cpusystem* to display *i_cpu*.
2. Use the middle mouse button to drag and drop the *i_cpu* to the signal pane of *nWave* and display the I/O signals.

NOTE: The horizontal yellow line in the signal pane moves so that it is under the signal added to the display. This line is the *signal cursor*, which marks the insertion point for signal commands for example, **Add**, **Move**, **Overlay**, and **Create Bus**. Right-click in the signal pane to access related commands.

3. In the design browser frame, expand *i_cpu* and double-click *i_ALUB* to display the source code.
4. In the *Source Code* frame, scroll to line 70 and select *AluBuf* from the signal declaration and drag to G2 to drop.

NOTE: To select individual, non-contiguous signals, press and hold the <Ctrl> key and click the signals you want. To select a large range of signals, drag the mouse over the selection or select the first line in the range and hold the <Shift> key while selecting the last line in the range.

You can also drag and drop signals or instances from other Verdi windows such as *nSchema* or the *Temporal Flow View*.

Use Get Signals to Add Signals

Perform the following steps to add signals using the *Get Signals* form. There are four panes in this form.

1. To display signals of interest, choose the **Signal -> Get Signals** command or click the toolbar icon  .
2. In the *Get Signals* form, click *CHILD1* in the design hierarchy box.

The signal list displays all the signals in *CHILD1*, as shown in the following figure:

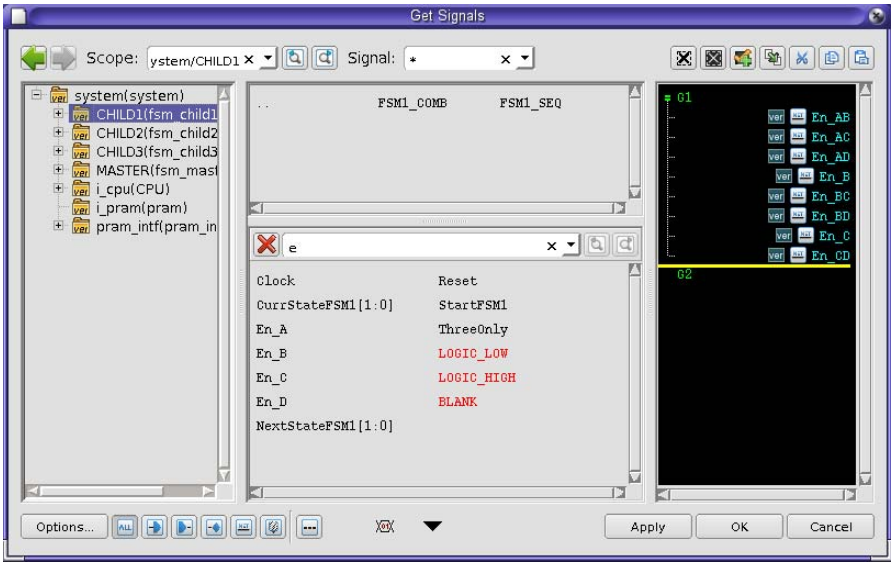


Figure: Get Signals Window

3. Click the middle button on group G3 in the mirror signal pane. This selection moves the signal cursor bar from group G2 to group G3, indicating that the insertion point for adding new selected signals is changed to group G3.
4. Click the **Select/Deselect All Signals** icon to select all signals.
5. Click the **Add Selected Signals** icon to add the signals to the mirror signal pane.

NOTE: You can rearrange the signal sequence in the mirror signal pane using the middle button to drag the signals to a new location.

---

6. Click **OK** to display the waveforms of the selected signals. The results are similar to the following:

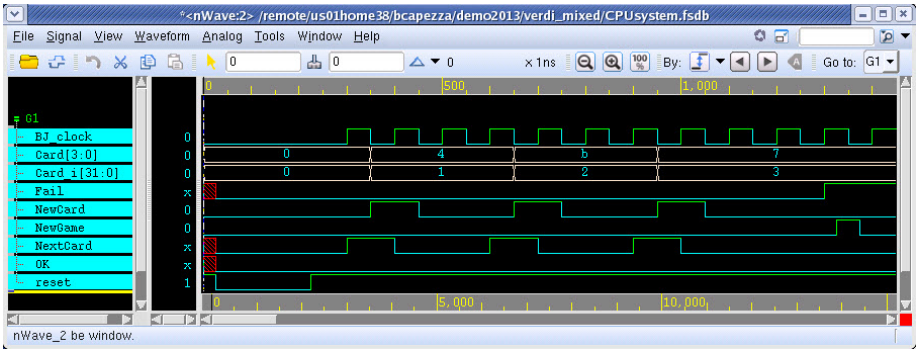


Figure: Signal Display in nWave

Search for Signals to Add

You can also search for signals to add in the *Get Signals* form.

1. Type 'g' in the waveform pane to invoke the **Signal -> Get Signals** command using the hot key. The *Get Signals* form appears.
2. Expand *i_cpusystem* and *i_cpu* and then click *i_CCU*.
3. Set the signal cursor position under group G4 in the mirror signal pane.
4. In the *Get Signals* form, click **Options**.
5. On the **Search** tab of the *Options* form, turn on the **Search Signals with Case Matching** option.
6. Click **Close**.
7. Type *clo** in the **Find Signal** field.
8. Press <**Enter**>.

   Note that no signals match.
9. Now type *CLO** in the **Find Signal** field.
10. Press <**Enter**>.

    Signals with names starting with *CLO* are listed.
11. Drag-left in the middle pane to select *CLOCK1, CLOCK2, CLOCK3,* and *CLOCK4.*
12. Click **Apply**.
13. In the *Get Signals* form, click **Options**.
14. On the **Search** tab of the *Options* form, turn on the **Search Signals in Sub-Scopes** option and click the **Close** button.
15. Click *i_cpusystem*.
16. Type *sel** in the **Find Signal** field.

17. Press <**Enter**>.

    Signals with names starting with *sel* in all scopes are listed.

18. Double-click *sel[2:0]* in the middle frame to add it to the right-hand window.
19. Click **OK**.

    The selected signals are displayed in *nWave* window.

Creating a Parent Group With Sub-Groups

When adding an array of interface with the same parent, it creates a parent group to contain these sub-groups, as illustrated in the following figure:

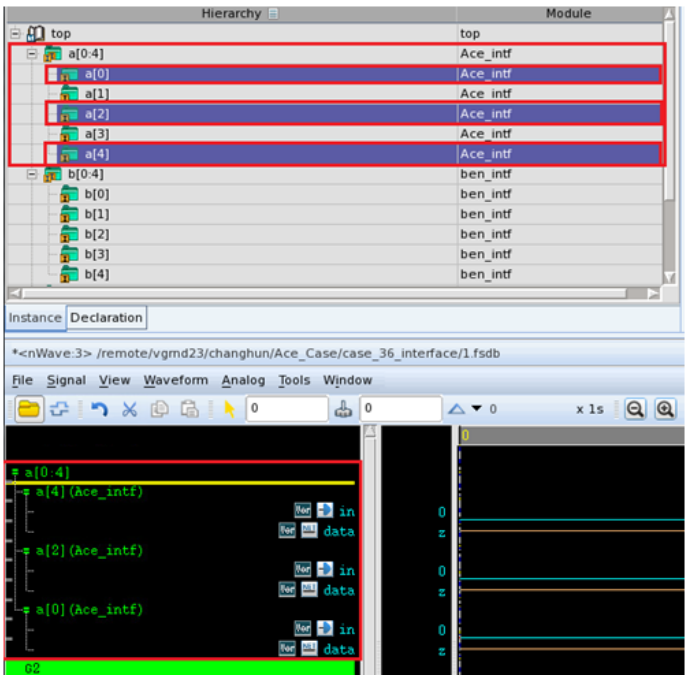


Figure: Array of interface added With the Same Parent Name

If the array of interfaces is added without the same parent name, then it creates a parent group first to contain these sub-groups. The location for adding it depends on the cursor position of the horizontal yellow line in the signal pane as illustrated in the following figure:

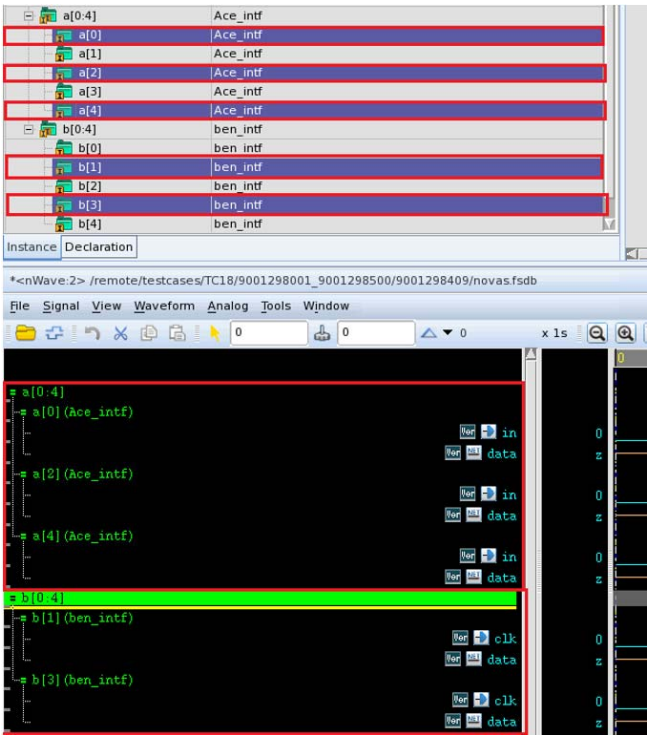


Figure: Array of interface added Without the Same Parent Name

If you right-click to add array of interfaces to the waveform and the signals are not available in the group, then it creates a parent group to contain these sub-groups. However, when a group is not empty, and **Add to New Group** option is turned *on*, it creates a parent group to contain these sub-groups. If the **Add to New Group** option is turned off and you right-click to add array of interface, it creates parent group to contain these sub-groups.

When adding an interface or a scope to signal pane, it adds the complete interface including the nested interface to the waves and organizes them hierarchically as illustrated in the following figure:

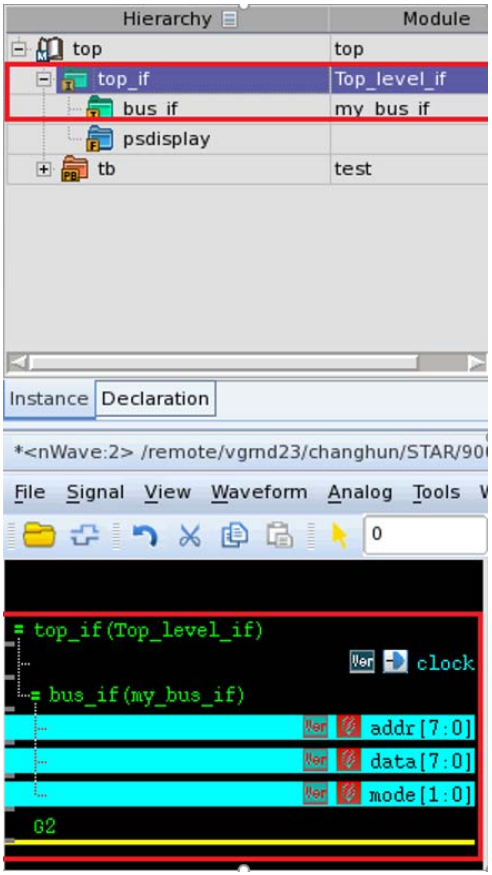


Figure: Added Nested Interface

Manipulate the Waveform View

nWave displays a cursor and a marker in the waveform pane. Use the cursor/marker to measure time differences, perform a fast zoom, or examine signal values. The cursor appears as a dashed yellow line, and the marker is a dashed white line. You can also add grid lines to make it easier to line up multiple signal transitions.

Set the Cursor/Marker Positions

1. Left-click the signal *R_W* under group G1 where it transitions from 0 to 1 at time 276 in the *nWave* window and note the following:
   - A vertical *cursor* line appears in the waveform pane.
   - The simulation time (276 ns) associated with the cursor's current location is displayed in the toolbar.
   - The value pane is automatically updated to reflect the values on the signals at the current cursor time.
   - The Active Annotation values (if enabled) in the source code frame are automatically updated to reflect the values on the signals at the current cursor time.

NOTE: You can also use the **Waveform -> Go To -> Time** command to set the cursor position to the specified time.

2. Click the middle button on the next transition (0 to 1) of the signal *addr[7:0]* under group G1 and note the following:
   - A vertical *marker* appears in the waveform pane.
   - The simulation time (425 ns) associated with the marker's current location is displayed in the toolbar.
   - The delta (time difference) between the cursor and marker is displayed in the toolbar.

NOTE: By default, the cursor and marker snaps to the closest transition on the selected signal. (You can turn on the **Waveform -> Snap Cursor to Transitions** toggle command to allow you to set the cursor/marker to any location.)

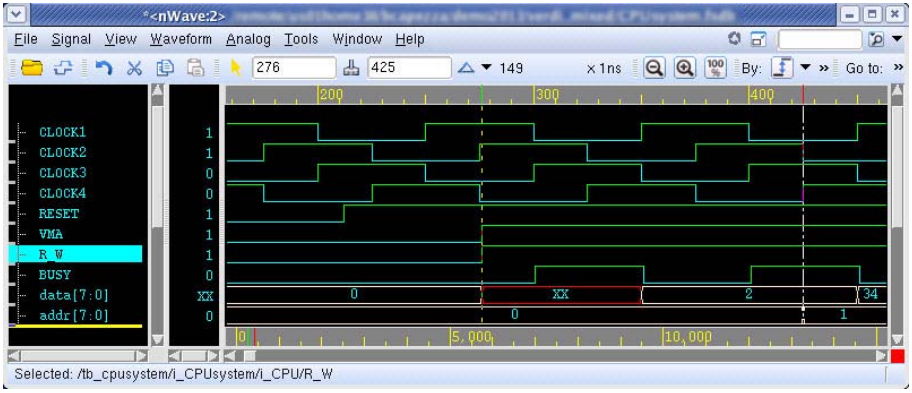


Figure: Example of Time Delta Between Cursor and Marker in nWave Window

3. In the *nWave* window, choose the **View -> Zoom -> Zoom All** command (or the toolbar icon or the “f” key) to display the entire simulation results for the signals currently on display.

Zoom Cursor With Three Clicks

You can set the cursor and marker positions using left-click and middle-click in a signal's waveform, then click the **Delta Time** icon on the toolbar to zoom in and view the waveform between the cursor and the marker.

Fast Zoom on the Full Scale Ruler

A full-scale ruler appears at the bottom of the waveform pane. The ruler shows the full simulation time span of the opened dump file. To view a time range quickly, left-click and middle-click the specified time along this ruler, then click the **Delta Time** icon on the toolbar to view the waveform between the two points. You can also drag-left along the full-scale ruler for **Fast Zoom**.

Pan the Waveform

To pan different time ranges or signals, use the waveform pane's horizontal or vertical scroll bars and the up, down, left, and right arrow keys.

Use Bind Key Commands

The following table lists the bind keys that you can use to view the waveforms quickly and easily:

Bind Key	Action
Up Arrow Key	Pan Up
Down Arrow Key	Pan Down
Left Arrow Key	Pan Left
Right Arrow Key	Pan Right
z	Zoom Out
Z	Zoom In
f	Zoom All
l	Last View

Turn On/Off Signal Grids

1. In the *nWave* window, zoom in around the 0 to 1000 time range.
2. Select *CLOCK1*.
3. Choose the **View -> Grid Options** command to open the *Grid Options* form. Turn on the **Grid on** option and select the **Rising Edge** in the option field.
4. Turn on the **Grid Count with Start Number** option and input **1** in the associated text field. The *Grid Options* form is similar to the following:

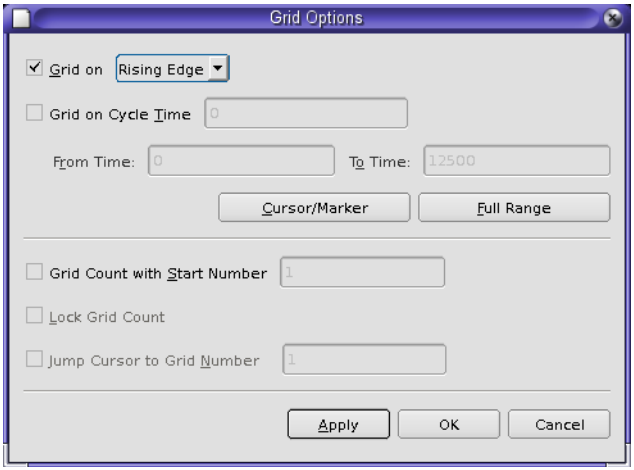


Figure: Grid Option form

5. Click **Apply** to show the numbering along with each grid line starting from the current cursor time.
6. Disable the **Grid on** option and click **OK** to remove grids.

Add Marker Labels

1. In the waveform pane, place the cursor on the 3 to a transition of signal *alubuf[7:0]* at time 825.
2. Choose the **Waveform -> Marker** command (or use the **shift-m** bind key) to open the *Marker* form.
3. In the *Marker* form, specify *ALUFail* in the **Name** field.
4. Click **Get Cursor Time** .
5. Click **Add**.

   The *Marker* form is similar to the following:

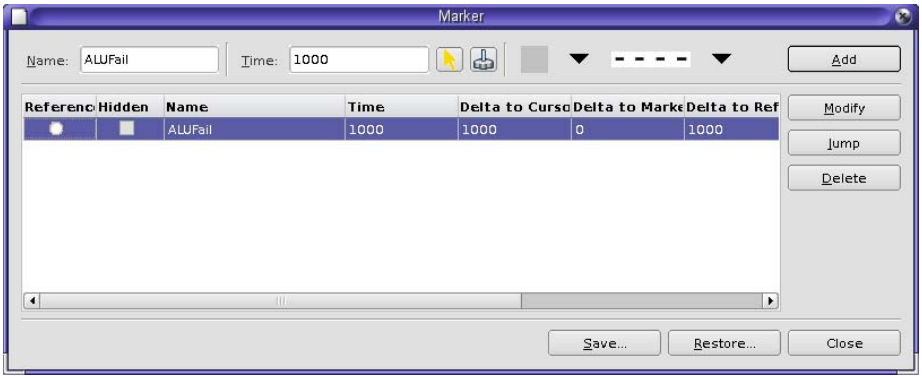


Figure: Marker Form

6. Click **Close**.

   The marker label is added to the top of the waveform view at the appropriate time and the **Goto Marker** field is added to the tool bar.

Figure: Marker Label Additions

Change the Display Sequence of Signals

To rearrange signals, you can drag-left to select and drag-middle to move or remove signals from the waveform pane with the **Cut** icon. The signals are copied to the clipboard. You can then use the **Paste** command to copy signals from the clipboard to the signal cursor position.

1. To rearrange the display sequence of signals, drag-left in the signal pane to select four signals under group G1. For example, *RESET, VMA, R_W,* and *BUSY*.
2. Use the middle mouse button to drag them to right after the signal *addr[7:0]*.
3. Click the **Cut** icon  to remove the *RESET, VMA, R_W,* and *BUSY* signals. Then middle-click to set the signal cursor position under group G3.
4. Click the **Paste** icon to insert the four signals at the current signal cursor position.

5. Click the **Undo** icon . The four newly inserted signals are deleted.
6. Click **Undo** again and the signals are inserted back under group G3. You can undo for one level only.

Search for Signal Value Transitions

You can search a signal by **Any Changes**, **Rising Edge, Falling Edge, Analog Values, Bus Values, Mismatches,** or **Search Constraint**.

1. Select the signal *CLOCK1*.
2. Click the **Search Forward** icon on the toolbar. The cursor jumps to the next signal transition.
3. Click the **Search Backward** icon to move the cursor to the previous transition.
4. Click the **Search By AnyChanges** icon By: to change the search criteria to **Rising Edge**.
5. Search again and notice the difference.
6. Select the signal *data[7:0]*.
7. Click the **Search By AnyChanges** icon to change the criteria to **Bus Values**.
8. In the *Search Value* form, enter *20* in the **Signal Value** field.

NOTE: You can also search for value to value transitions (including mnemonics) by entering 'value1 -> value2' in the **Signal Value** field.

---

9. Click **OK**.

   To change the bus value, choose the **Waveform -> Set Search Value** command or change the value on the tool bar to define the search value.

10. Click the **Search Forward** or **Search Backward** icons to find the value of *20* at time 4650 ns.
11. With *data* still selected and a bus value of *20*, choose the **Waveform -> Set Search Constraint** command.

    The *Set Search Constraint* form displays.

12. Change the **Value of <Search By> is stable for** menu to <= and enter 100 in the **x 1ns** box, as shown in the following figure:

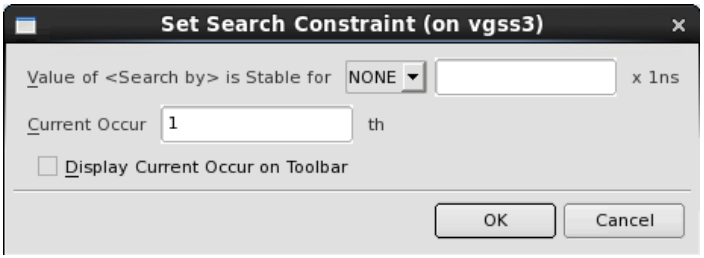


Figure: Set Search Constraint Box Filled

13. Click **OK**.
14. Use **Search Forward** and **Search Backward** icons to find any occurrences of the value *20* on *data*, stable for less than 100 ns.

    There is one at time 1576 ns.
15. Change **Search Constraint** back to **NONE**, and **Search By** to any transition.

Add Comments

You can insert comments to indicate items of interest.

1. Continue from the previous example and place the cursor under *data[7:0]*.
2. Choose the **Signal -> Comment -> Insert** command to add a comment field to the waveform.

   The cursor should still be at time *1576* where data value *20* is less than *100*. You want to identify this location with a comment.
3. Choose the **Signal -> Comment -> Add Attached Square Box** command.
4. Left-click near the value of 20 between the red comment lines to add the comment box.

   After the comment box is added, you can reposition it with the left mouse button.
5. Place the cursor over the **Comment Box** text and select the text by double-clicking.
6. Press **Delete** on the keyboard to remove the text.
7. Type '`This is where the width is too small`.' to enter the new comment text.
8. Re-size the comment box as needed.

Compress Time Ranges

You can compress time ranges to make viewing different times easier.

1. In *nWave*, choose the **View -> Compress Time Range** command to open the *Compress Time Range* form.
2. In the *Compress Time Range* form, enter 1000 in the **From Time** field and 13000 in the **To Time** Field.
3. Click **Insert**.
4. In the waveform, click the **Zoom All** toolbar icon. The *nWave* window is similar to the following figure:

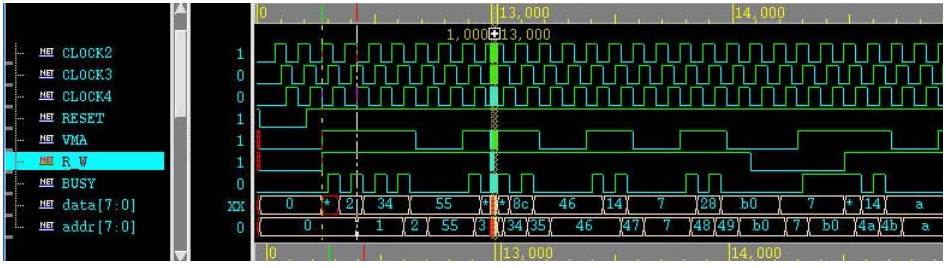


Figure: Compress Time Range

The yellow vertical bar indicates where time is compressed. Only the waveform view is affected.

5. In the *Compress Time Range* form, click **Delete All** and **Close** to remove the compressed time range.

Split the Waveform View

You can split the waveform window to keep a standard set of signals in the top part of the window while you scroll through the remaining signals.

1. In *nWave*, choose the **Window -> Horizontal Split** command to split the window.

   You can select the separating bar and drag to change the split size.
2. In the upper split, scroll to display *addr[7:0]* and *data[7:0]*.
3. In the lower split, scroll to display the signals in group *G4*. The *nWave* window is similar to the following figure:

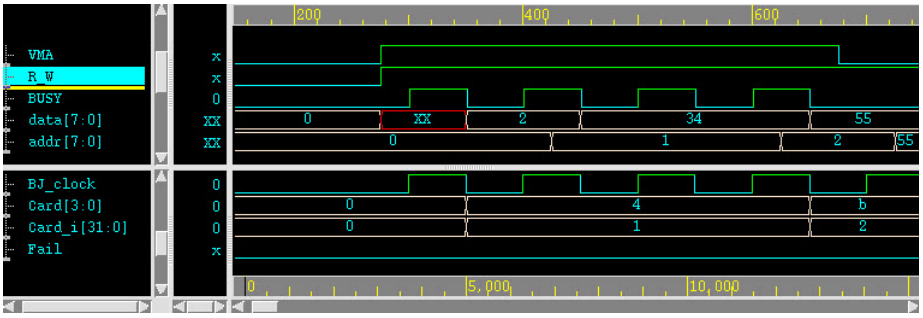


Figure: nWave Split Window

4. Choose the **Window -> Stop Split** command to return to a single window.

Change Signal/Group Attributes

Search for a Group

1. Scroll to the bottom of the waveform list.
2. Right-click the signal pane. A **Signal** right mouse button command menu appears.
3. Select the group on the **Go To** sub-menu to jump among groups.
4. Go to group G1.

Change the Group Name

1. Right-click the group G1. A **Signal** right mouse button command menu displays.
2. Choose the **Rename** command. G1 turns orange.
3. Drag-left to highlight G1.
4. Type *CPU* and press <**Enter**>. The group name G1 changes to CPU.

NOTE: You may need to use delete or backspace to remove the existing group name.

5. Change group G2 to *ALUB*, and change group G3 to *MISC*.
6. Double-click the *ALUB* group to collapse.

NOTE: You can also create hierarchical groups by choosing the **View -> Group Manager** command.

Modify the Display Format in the Value Window

1. Zoom the waveform to the time range between 6500 and 7500.
2. In the signal pane, search for the bus *addr[7:0]*. The bus values displayed on the waveform are in hexadecimal format.
3. In the value pane, right-click the value of *addr[7:0]*. A data format menu appears.
4. Choose the **Radix -> Binary** command. The value displayed for *addr[7:0]* changes to binary format.
5. Change back to hexadecimal format.

Display Hierarchical Signal Names

1. Choose the **View -> Hierarchical Name** command to display the signal names with their full hierarchical paths.
2. To make the signal pane bigger, drag the boundary between the signal pane and the value pane to widen the signal name pane. This change allows you to view the full hierarchical names.

   The *nWave* window is similar to the following figure:

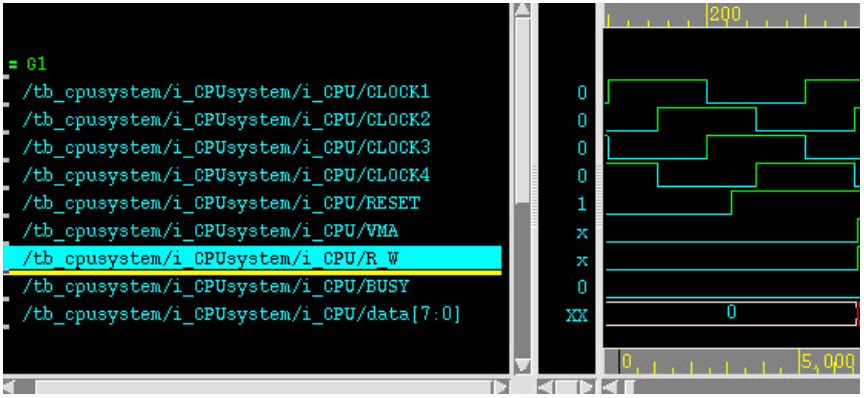


Figure: Full Hierarchical Names

3. Choose the **View -> Hierarchical Name** command again to restore the signals to their base names.

Add Alias to Display Bus Values

You can display logic states in a more meaningful way than by only viewing the plain logic values using the alias mechanism in *nWave*. You can associate mnemonics with logic states using the **Waveform -> Signal Value Radix -> Add Alias from File** command. Using a symbolic alias can make your debugging process easier.

1. In the signal pane click the signal *data[7:0]*.
2. Choose the **Waveform -> Signal Value Radix -> Add Alias from File** command.

   The *Open Alias File* form displays, as shown in the following figure:

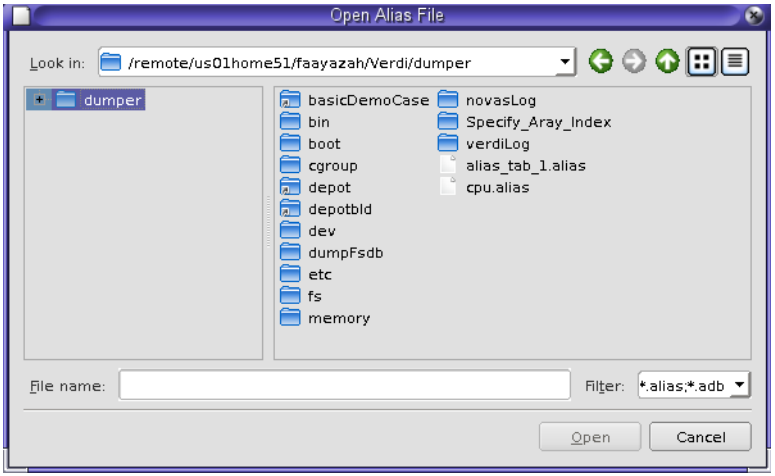


Figure: Example Open Alias File Form

3. Select the *CPU.alias* file under the *<working_dir>/demo/verilog_mixed* directory and click **OK**.

   The values displayed on the waveform for signal *data[7:0]* change to alias strings, which are more readable and meaningful. Use a text editor to create your own alias file by following the format in the *CPU.alias* file.

NOTE: You can add color to the alias values by choosing the **Waveform -> Signal Value Radix -> Edit Alias** command while the aliased signal is selected. This opens the *Alias Editor* form where you can specify a background color for the alias values.

---

4. Create a copy of the signal *data[7:0]*.
   a. In the signal pane, place the cursor above *data[7:0]*.
   b. Select *data[7:0]*.
   c. Click the **Copy** icon on the toolbar and then click the **Paste** icon.
5. Place the cursor over the value in the value column for the *data[7:0]* signal copy and right-click to view the right mouse button menu.
6. Choose the **Remove Local Alias** command.

   The values displayed on the waveform for *data[7:0]* signal copy change back to hexadecimal values. Now you can see the numeric value and the mnemonic value simultaneously.

Change the Spacing and Signal Height

You can change the spacing equally among all displayed signals. You can also change the signal height for each signal individually.

1. Disable the **Signal -> Select Group Mode** toggle command (Group/Signal). This allows you to easily select all signals in a group instead of the group name.
2. Click group *MISC* to select all the signals under that group.
3. Choose the **Waveform -> Height** command and type *20* in the **Signal Height** field in the *Signal Height* form, shown in the following figure:

Figure: Signal Height Form

4. Click **Apply** to change the signal height to 20 pixels.
5. Click **Default** to go back to default height.
6. Choose the **Waveform -> Spacing** command to change the signal spacing.

NOTE: The height and spacing can be changed globally through the **Waveform -> Default Value** folder -> **Display Signal** page on the *Preferences* form (invoked with the **Tools -> Preferences** command).

---

7. Turn on the **Signal -> Select Group Mode** toggle command (Group/Signal).

   This allows you to select group names.

Change Signal Color/Pattern

To change a signal's color or line width/style, perform the steps following steps:

1. Choose the **Waveform -> Color/Pattern** command. The color palette is displayed, as shown in the following figure:

Figure: Example Color/Pattern Palette

2. Select the signals you want to change and the preferred color from the color palette. The color for the selected signals changes accordingly.
3. You can also change the signal's line width and line style by setting the corresponding option menu.
4. Click **Default** on the color/pattern palette to change the selected signals to their default color/pattern.

Create New Signals/Buses from Existing Signals

Sometimes it may be necessary to manipulate signals for better understanding or to test a theory. There are two methods to perform this function:

- Logical Operations
- Bus Creation

Anything created with these commands can be saved to the signal file.

Logical Operations

If you want to view what the waveform looks like if *BUSY* was combined with inverted *VMA* signal.

1. In the CPU group, select *BUSY* and *VMA* (press and hold the <Ctrl> key to select multiple signals).
2. Choose the **Signal -> Logical Operation** command.

   The *Logical Operation* form displays, as shown in the following figure:

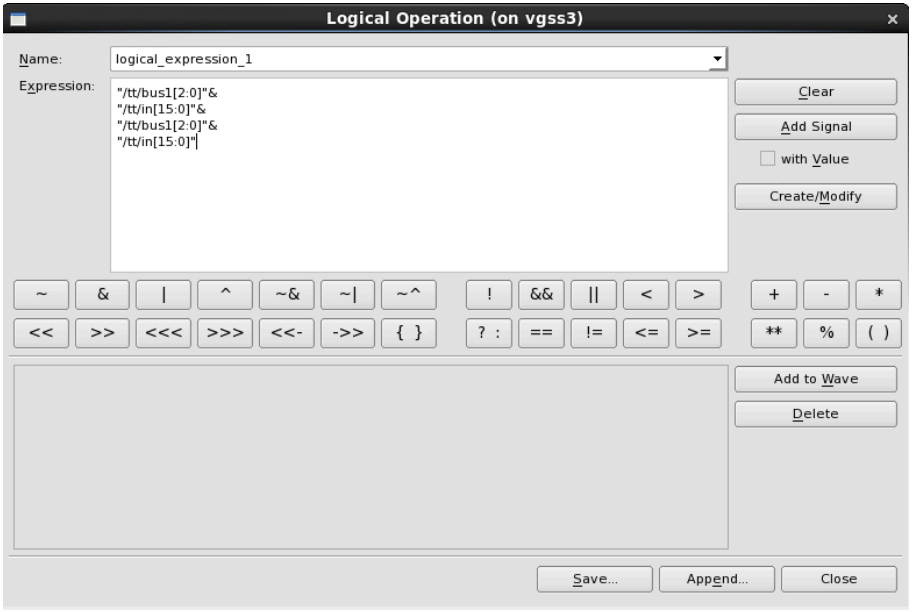


Figure: Logical Operation Form

3. Enter *newsig* in the **Name** field.
4. In the **Expression** field, highlight the "&" symbol and delete using the **Delete** key on the keyboard.
5. With the cursor between the signals, left-click the *|:B-or*, *R-or* button under the **Expression** field to insert the new operator.
6. Put the cursor in front of the string for VMA and insert the *~:B-negation* operator.
7. Click **Create/Modify** to create the new signal. The *nWave* window is similar to the following figure:

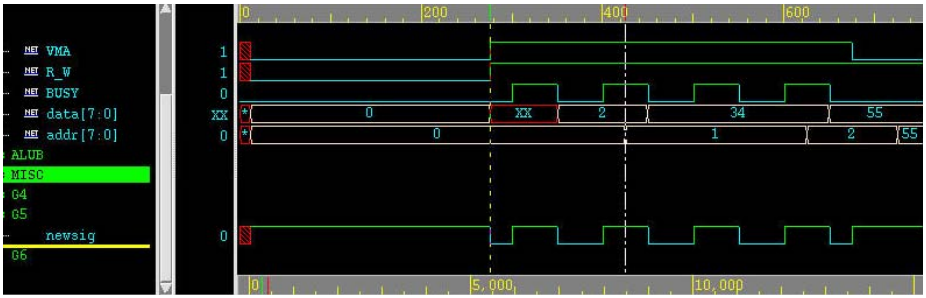


Figure: Logical Expression

The new signal is added to the waveform at the current cursor location.

8. Click **Close**.

Bus Creation

This exercise describes how to create a bus from the displayed signals.

1. Put the cursor in group G4.
2. Select signals *CLOCK1*, *CLOCK2*, *CLOCK3*, and *CLOCK4* in group G4.
3. Choose the **Signal -> Bus Operations -> Create Bus** command, or use the right mouse button and choose the **Bus Operations -> Create Bus** command.

   The *Create Bus* form displays, as shown in the following figure:

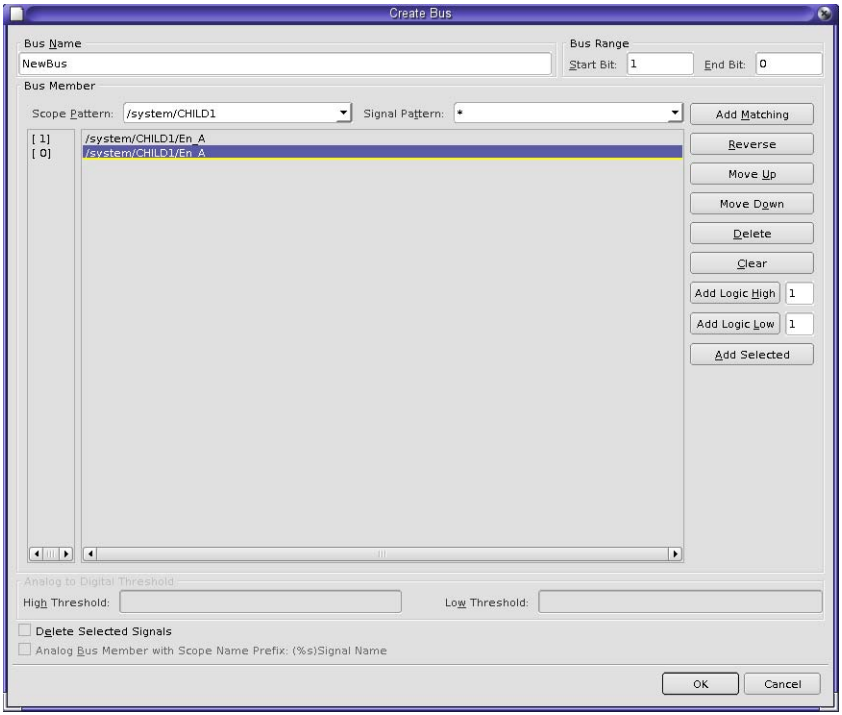


Figure: Create Bus Form

4. Enter *CBUS* in the **Bus Name** field.

   CBUS is the new name for the created bus.

5. Click the **Reverse** button.

   CLOCK 4 becomes the MSB.

6. Click **OK**.

   The created bus *CBUS[3:0]* is added to the waveform at the signal cursor position.

Expand or Collapse the Bus

To expand a bus to its individual members, double-click the bus in the signal pane. You can also choose the **Signal -> Expand Bus** command.

1. Double-click *CBUS*. Signals *CLOCK1*, *CLOCK2*, *CLOCK3*, and *CLOCK4* are displayed after *CBUS[3:0]*.
2. Double-click *CBUS* again to collapse the bus.

Save and Restore Signals

Save the Displayed Signals

nWave saves all the displayed signals and their signal attributes such as color, height, and other information to a save signal file. You can easily restore the same waveforms later by restoring this file.

1. Choose the **File -> Save Signal** command.

   The *Save Signal* form displays.

2. Click **Options**, and decide which attributes to save.
3. Enter *demo.rc* as the file name and click **OK**. By default, the file name is saved as *signal.rc*.

Restore Previously Saved Signals

1. In the *nTrace* main window, choose the **File -> Exit** command to close the Verdi session.
2. Start the Verdi platform and *nWave* again.
3. Choose the **File -> Restore Signal** command and select *demo.rc* in the file name list.
4. Click **OK**.

   nWave displays the signals you saved in the last session.

Create a Second Waveform Window and Restore Other Signals

1. Choose the **Tools -> New Waveform** command to create a name *nWave* window. Click the **Undock** icon to make the new *nWave* window as a standalone *nWave* window.
2. Open the *<working_dir>/demo/verdi_mixed/CPUsystem.fsdb* file that contains the results from a different RTL simulation.
3. Restore the file *demo.rc*.

   Note that the signal arrangement used for the first *nWave* window is now shown in the second *nWave* window.

   If no file is currently open, the **File -> Restore Signal** command opens the simulation result file specified in the *demo.rc* file and restores the signals.

If a file is open, the **File -> Restore Signal** command ignores the file specified in *demo.rc* and restores the signals in the open file.

4. After you have two waveforms open, you can choose the **Window -> Change to Primary** command to change the primary waveform window. This command selects which FSDB file will be used for active annotation.

   The primary *nWave* window is identified with a red square in the lower right corner.

NOTE: If you load multiple FSDB files in the same *nWave* window, you can specify the active file by choosing the **File -> Set Active** command. This opens the *Active File* form, where you can select the desired file and turn on the **Apply to Active Annotation** option.

Calculate Toggle Coverage

Sometimes, it is not easy to determine if the test patterns toggle all the signals in the design. The **Toggle Coverage Report** command analyzes all signals in the design for transitions using a post-simulation FSDB file.

1. After the FSDB file is loaded, choose the **Tools -> Toggle Coverage Report** command in the *nWave* window to open the *Toggle Coverage* form as shown in the following figure.

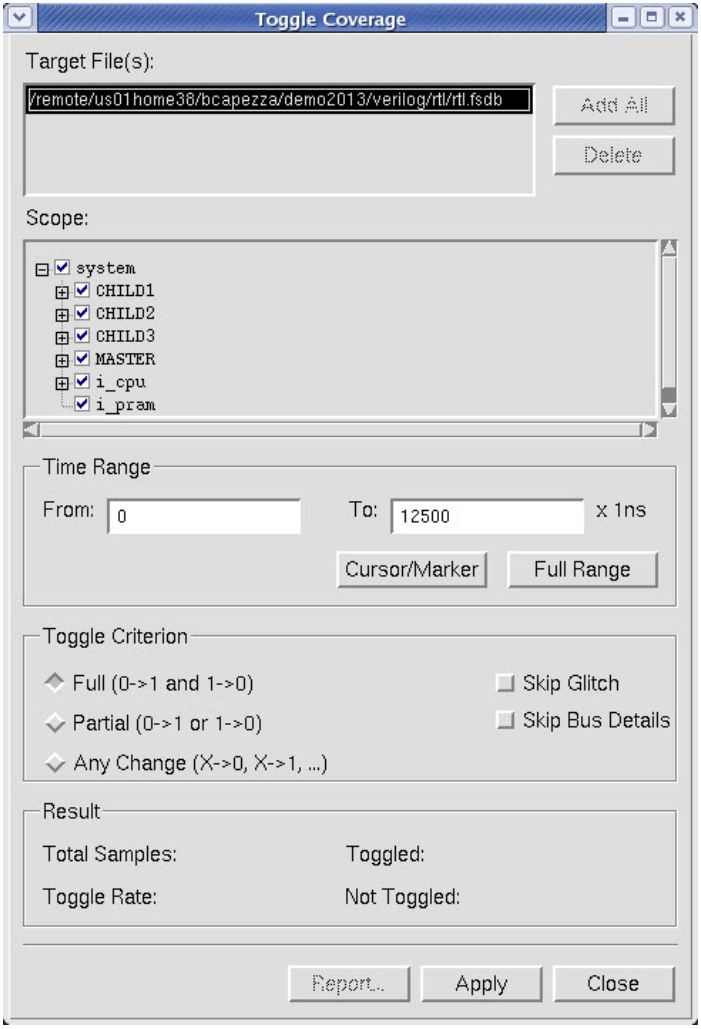


Figure: Toggle Coverage Form

The *Toggle Coverage* form provides several options to direct the coverage analysis.

2. In the **Scope** section, uncheck the CHILD1, CHILD2, CHILD3, and MASTER scopes.
3. Click **Full Range** to add the entire FSDB time range.
4. Confirm **Full** is selected in the **Toggle Criterion** section. That is, the signals that go from 0 to 1 and then 1 to 0 or signals that go from 1 to 0 and then 0 to 1 are identified as one toggle.
5. Turn *on* the **Skip Glitch** option so glitches are not included in the toggle counts.
6. Click **Apply** to start the toggle coverage analysis. During the analysis process an *Information* dialog window is opened.
7. Click **OK** on the *Information* dialog window.

   When the toggle coverage analysis is complete, the results are displayed in the **Result** section of *Toggle Coverage* form.

Figure: Result Section of Toggle Coverage Form

The description of each item in the **Result** section is listed below:

- **Total Samples**: Total number of analyzed signals.
- **Toggled**: Signals that are toggled.
- **Not Toggled**: Signals that are not toggled.
- **Toggle Rate**: The percentage of toggled signals divided by the analyzed signals (that is, **Toggled** / **Total Samples**).

8. In the *Toggle Coverage* form, click **Report** to open the *Toggle Coverage Report* form. The default displays the **Not Toggled** signal list.

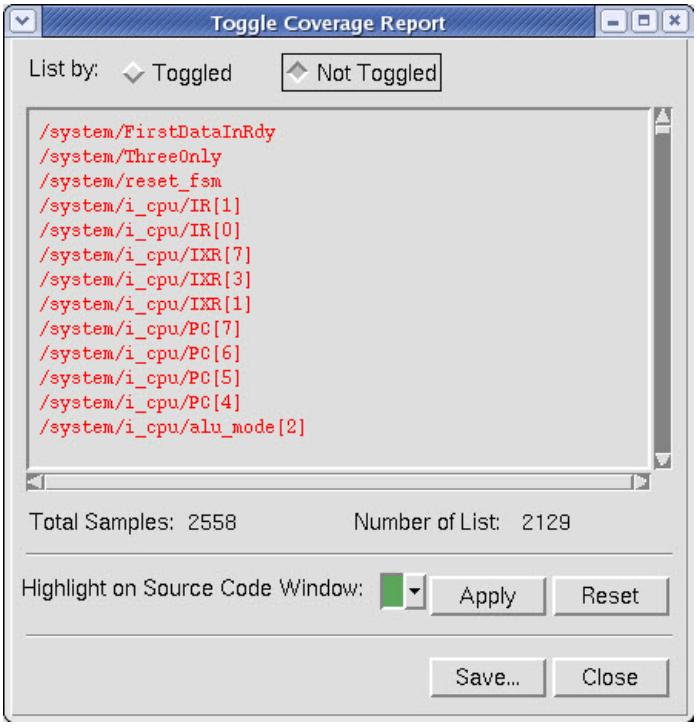


Figure: Not Toggled Results

The un-toggled signals are listed in red. The total number of signals analyzed and the current list are summarized in the bottom of the form.

9. Select the **Toggled** option to display the toggled signal list. The toggled signals are listed in green.

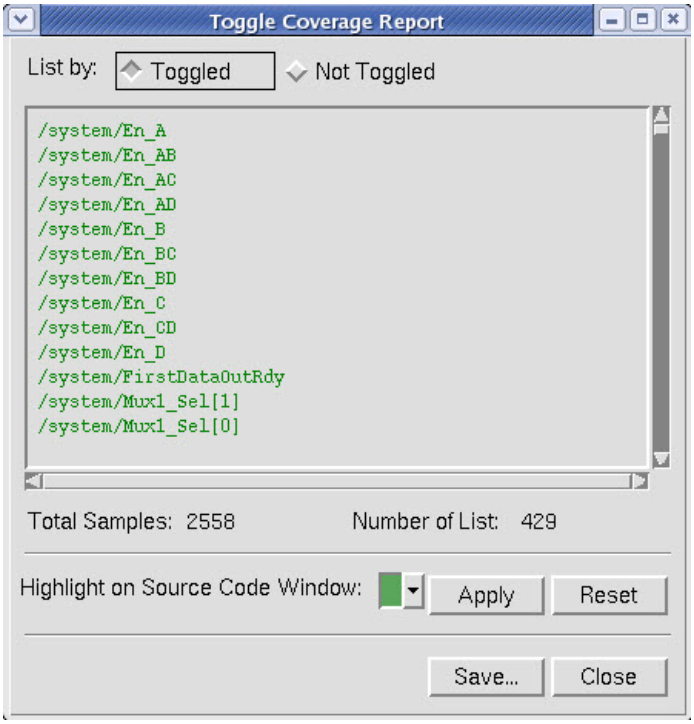


Figure: Toggled Results

10. In the *Toggle Coverage Report* form, click **Save** to open the *Save Report To* form and specify a file name for the saved report. The information in the saved text file includes all the settings and information related to the current toggle coverage analysis.

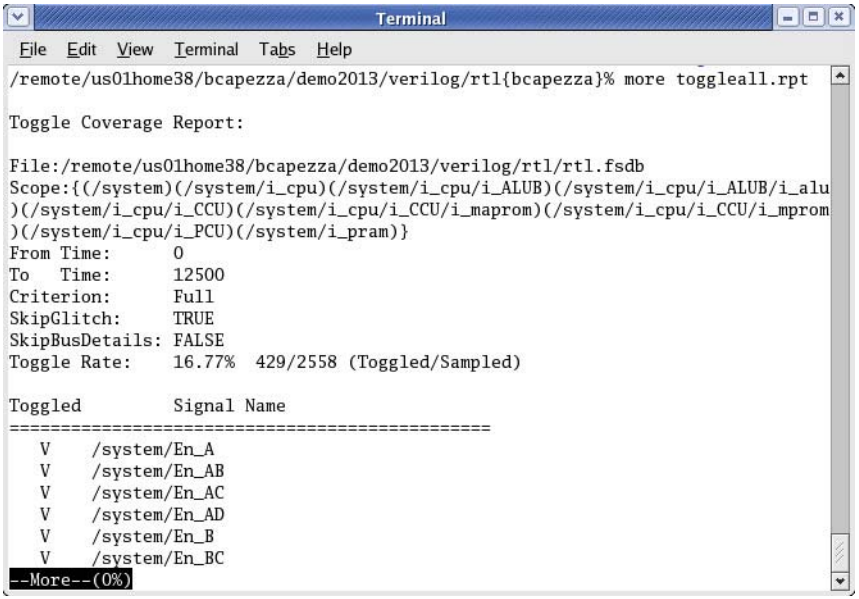


Figure: Toggle Coverage Text Summary

Define Events and Complex Events

nWave allows you to create events to help figure out complex conditions. Through previous events, you can then capture the designated conditions clearly.

Create a Single Event

Simple events can be created through a fixed combination of signals. For example, to capture a synchronous read and write cycle, you can specify the correct signals and values. Simple events can then be used to form a complex event (for example, limit the duration, the occurred sequence, or the number of times the event must be issued).

1. Close the previous Verdi session.
2. Change directories to the <*working_dir*>*/demo/verilog/rtl* directory and then invoke the Verdi platform as follows.

   ```
   % cd <working_dir>/demo/verilog/rtl
   % verdi -f run.f -ssf rtl.fsdb -workMode hardwareDebug &
   ```

3. In the *nWave* window, choose the **Signal -> Get Signals** command to open the *Get Signals* form.
4. In the *Get Signals* form, select the R_W, clock, and reset signals under the system/i_cpu scope and click the **OK** button.
5. Choose the **Waveform -> Go To -> Time** command to open the *Search Time* form.
6. Enter *400* in the **Time Value** field and click the **OK** button.

   The cursor changes to time 400 in the waveform pane.
7. With the three signals selected, choose the **Signal -> Event** command to open the *Event Window*.
8. In the *Event Window*, click **Insert** to create *event 0*.

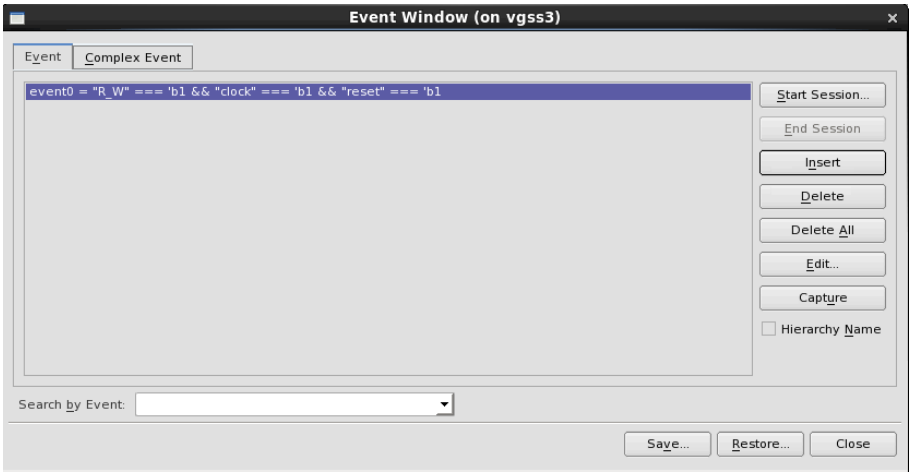


Figure: nWave Event Window

An event is inserted using the selected signals current values as the event conditions.

9. In the *Event Window*, click **Edit**.
10. In the *Edit Event* form, change the name to *read_cycle* in the **Event Name** field. Do not change the expression which should be:

    ```
    “clock”=== `b1 && “reset” === `b1 && “R_W” === `b1
    ```

11. Click **OK** and the *read_cycle* event is added into *Event Window*.
12. Click **Capture** to view the related waveform. Leave the *Event Window* open.

Save and Reload Events

1. In the *Event Window*, click **Save** and save the event to a file named read_cycle.rc
2. Exit the Verdi session.
3. Start the Verdi platform again with:

    ```
    % verdi -f run.f –ssf rtl.fsdb -workMode hardwareDebug &
    ```

4. In the *nWave* window, choose the **Signal -> Event** command.
5. In the *Event Window*, click the **Restore** button and load the read_cycle.rc file. The *read_cycle* event is listed in **Event** tab.
6. Select the *read_cycle* event and click **Capture**, you find that the *read_cycle* event is added to the *nWave* window.

Create a Complex Event

1. In the *Event Window*, click the **Complex Event** tab to add complex events.
2. On the **Complex Event** tab, click **Edit**. The *Edit Complex Event* form opens.
3. In the *Edit Complex Event* form stay at level 0 and perform the following:
   a. In the **Complex Event Name** field, enter *level_trigger1*.
   b. Change the condition to **IF read_cycle LASTS** (click **OCCURS** to change the action) **25ns THEN trigger**.

   After the changes, the form is similar the following figure:

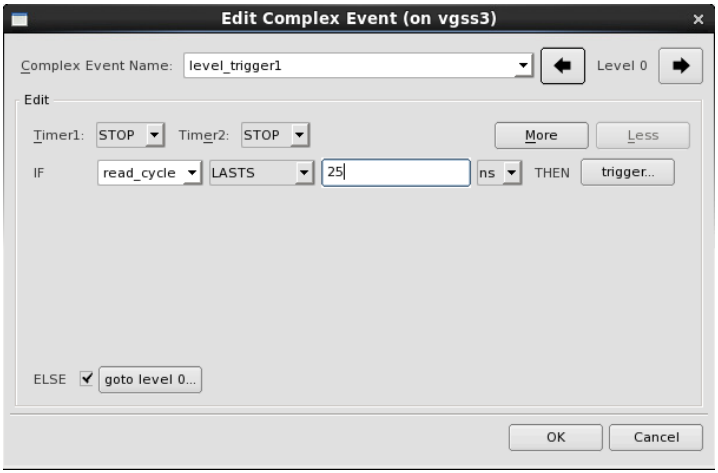


Figure: nWave Edit Complex Event Form

   c. Click **OK** to create this complex event and close the *Edit Complex Event* form.
4. In the *Event Window*, click **Capture** to capture the event. The event is added to the *nWave* window with signal name *level_trigger1* as shown in the following figure:

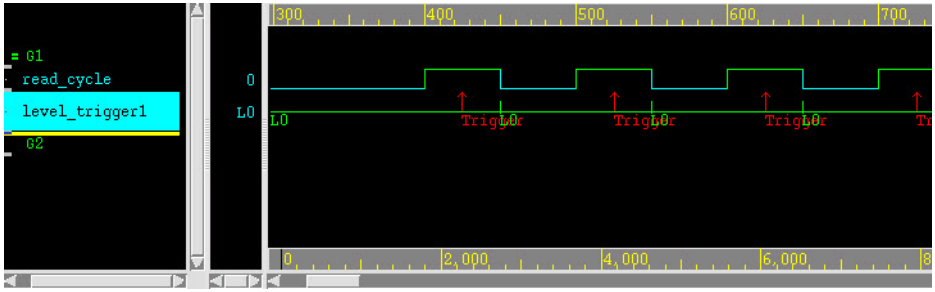


Figure: level_trigger1 Captured in nWave

Create a Complex Event with a Timer

Two built-in timers are provided that can be used to compose the complex event. Timers are used to trace the period of a complex event. They can be controlled with different options: START, STOP, PAUSE, and CONT. The timer is globally set. If no value is set for the timer in the sub-conditions, then the value is inherited from the global setting. The sub-condition can have its own setting with a priority higher than the global ones.

Example 1

1. In the *Event Window*, **Complex Event** tab, click **Edit** to open the *Edit Complex Event* form.
2. In the *Edit Complex Event* form, stay at level 0 and perform the following:
   a. In the **Complex Event Name** field, enter *level_trigger2*.
   b. Change the condition to **IF read_cycle LASTS 25 ns THEN goto level 1** (click **trigger** to change the action).
   c. Leave the remaining fields unchanged.
   d. Increase the level number from **Level 0** to **Level 1** by clicking the right arrow.
3. In the *Edit Complex Event* window, stay at level 1 and perform the following:
   a. Change **Timer1** to **START**
   b. Change the condition to **IF timer1 == 5 ns THEN trigger**
   c. Leave the remaining fields unchanged.
4. Click **OK** to create this complex event and close the *Edit Complex Event* form.
5. In the *Event Window*, turn *on* the **Also Capture Timer and Counter** option and click the **Capture** button to capture the event.

The event is added to the *nWave* window with signal name *level_trigger2*. When the state jumps from level0 to level1, timer1 is started, and then after 5 ns, the trigger occurs as shown as in the following figure:

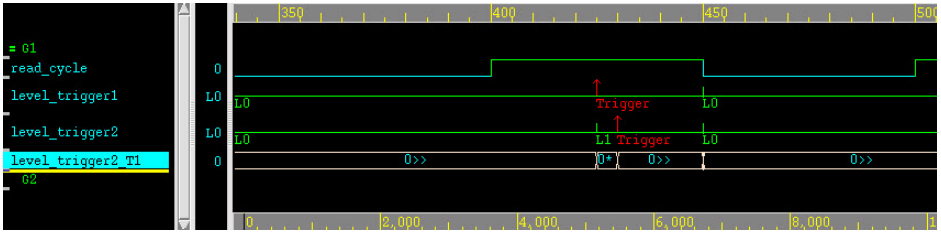


Figure: level_trigger2 Captured in nWave

Example 2

1. In the *Event Window*, **Complex Event** tab, click **Edit** to open the *Edit Complex Event* form.
2. In the *Edit Complex Event* form, stay at level 0 and perform the following:
   a. In the **Complex Event Name** field, enter *level_trigger3.*
   b. Change **Timer1** to **START**.
   c. Change the condition to **IF read_cycle LASTS 25 ns THEN goto level 1** (click **trigger** to change the action).
   d. Leave the remaining fields unchanged.
   e. Increase the level number from **Level 0** to **Level 1** by clicking the right arrow.
3. In the *Edit Complex Event* window, stay at level 1 and perform the following:
   a. Change **Timer1** to **STOP**.
   b. Change the condition to **IF timer1 > 5 ns THEN trigger**.
   c. Leave the remaining fields unchanged.
4. Click **OK** to create this complex event and close the *Edit Complex Event* form.
5. In the *Event Window*, turn *on* the **Also Capture Timer and Counter** option and click **Capture** to capture the event.

   The event is added to the *nWave* window with signal name *level_trigger3*. When the state jumped to level1, timer1 is stopped, but the “IF” condition is checked at the same time. At this time point, timer1 is still larger than 5, so the trigger occurs as shown in the following figure:

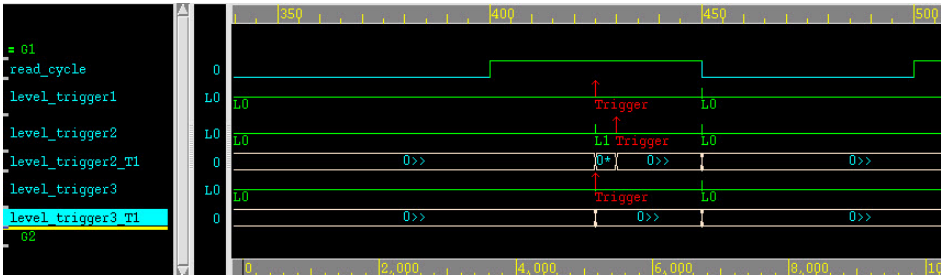


Figure: level_trigger3 Captured in nWave

Example 3

1. In the *Event Window*, **Complex Event** tab, click **Edit** to open the *Edit Complex Event* form.
2. In the *Edit Complex Event* form, stay at level 0 and perform the following:
   a. In the **Complex Event Name** field, enter *level_trigger4*.
   b. Change **Timer1** to **STOP**.
   c. Change the condition to **IF read_cycle LASTS 25 ns THEN goto level 1** (click the trigger button to change the action).
   d. Leave the remaining fields unchanged.
   e. Increase the level number from **Level 0** to **Level 1** by clicking the right arrow.
3. In the *Edit Complex Event* window, stay at level 1 and perform the following:
   a. Change **Timer1** to **STOP**.
   b. Change the condition to **IF timer1 > 5 ns THEN trigger**.
   c. Leave the remaining fields unchanged.
4. Click **OK** to create this complex event and close the *Edit Complex Event* form.
5. In the *Event Window*, turn *on* the **Also Capture Timer and Counter** option and click **Capture** to capture the event.

   The event is added to the *nWave* window with signal name *level_trigger4*. The timer is not started. When the state jumped to level1, it checks the “ELSE” condition and stays in level1 as shown in the following figure:

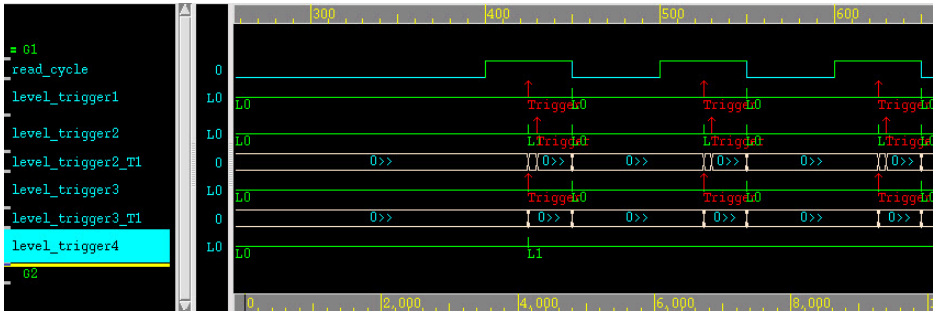


Figure: level_trigger4 Captured in nWave

6. Exit the Verdi session.

Create Complex Event with a Counter

Two built-in counters are provided that are used for counting if some condition occurs or not. Counters can be controlled with different options: INC, DEC, or RESET. The counters are also set globally and they do not stop counting when the level changes unless you reset them.

1. Start the Verdi platform again with:

   ```
   % verdi -f run.f –ssf rtl.fsdb -workMode hardwareDebug
   ```

2. In the *nWave* window, choose the **Signal -> Event** command.
3. In the *Event Window*, click **Restore** and load the read_cycle.rc file. The *read_cycle* event is listed in **Event** tab.
4. Select the *read_cycle* event and click **Capture**, you find the *read_cycle* event is added to the *nWave* window.

Example 1

1. In the *Event Window*, **Complex Event** tab, click **Edit** to open the *Edit Complex Event* form.
2. In the *Edit Complex Event* window stay at level 0 and perform the following:
   a. In the **Complex Event Name** field, enter *counter_trigger1*.
   b. Change the condition to **IF read_cycle OCCURS 2 Times THEN goto level 0**. When you click the **trigger** button to change the value to “goto level 0”, the *Actions* form opens.
   c. In the *Actions* form, turn on **Counter1** and choose **INC**, then click **Apply**.

d. Click **More**.

e. Change the ELSE IF condition to **ELSE IF counter1 == 2 Times THEN trigger**.

f. Click **trigger** to open the *Actions* form.

g. In the *Actions* form, turn on **Counter1** and choose **RESET**, then click **Apply**.

h. Leave the remaining fields unchanged.

3. Click **OK** to create this complex event and close the *Edit Complex Event* form.

4. In the *Event Window*, turn on the **Also Capture Timer and Counter** option, and then click **Capture** to capture the event.

   The event is added to the *nWave* window with signal name *counter_trigger1*. The system is operating in real time, not sequential, so the transition of *read_cycle* is checked twice when the state jumped to level0 as shown in the following figure:

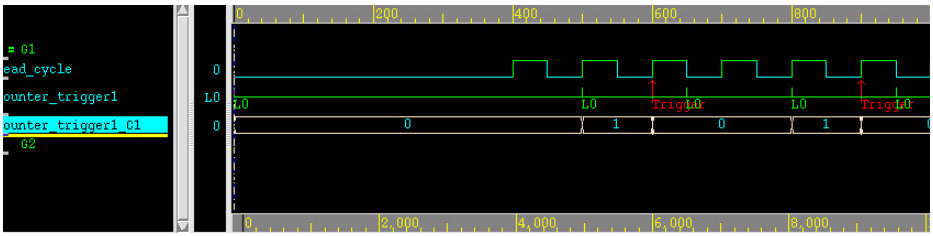


Figure: counter_trigger1 Captured in nWave

Example 2

1. In the *Event Window*, **Complex Event** tab, click **Edit** to open the *Edit Complex Event* form.

2. In the *Edit Complex Event* window stay at level 0 and perform the following:

   a. Click **New** to clear the previous event description.

   b. In the **Complex Event Name** field, enter *counter_trigger2*.

   c. Change the condition to **IF read_cycle OCCURS 2 Times THEN goto level 0**.

   d. Click **More**.

   e. Change the ELSE IF condition to **ELSE IF counter1 == 2 Times THEN trigger**.

   f. Leave the remaining fields unchanged.

3. Click **OK** to create this complex event and close the *Edit Complex Event* form.
4. In the *Event Window*, click **Capture** to capture the event.

   The event is added to the *nWave* window with signal name *counter_trigger2*. As counter1 is not increased in the condition; it does not reach 2, so no trigger occurs as shown in the following figure:

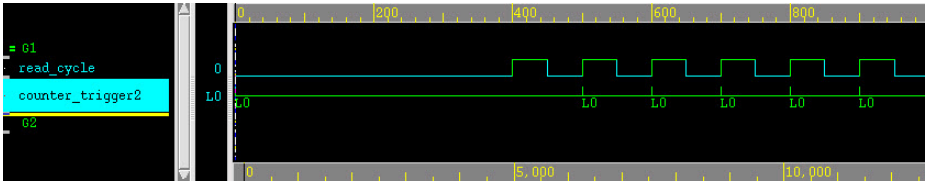


Figure: counter_trigger2 Captured in nWave

nState Tutorial

Overview

nState is a Finite State Machine (FSM) viewer and analyzer that generates bubble diagrams for visualization of state machines that are automatically recognized by the Verdi platform when compiling the Verilog/VHDL source code modules. States and transitions are annotated with logic conditions and animated with simulation results. *nState* analyzes the simulation results to determine state and transition coverage.

Before you begin this tutorial, follow the instructions in the *Before You Begin* chapter.

The Verdi platform automatically recognizes any FSMs and indicates them in *nSchema* with a symbol containing three linked circles, as shown in the following figure:

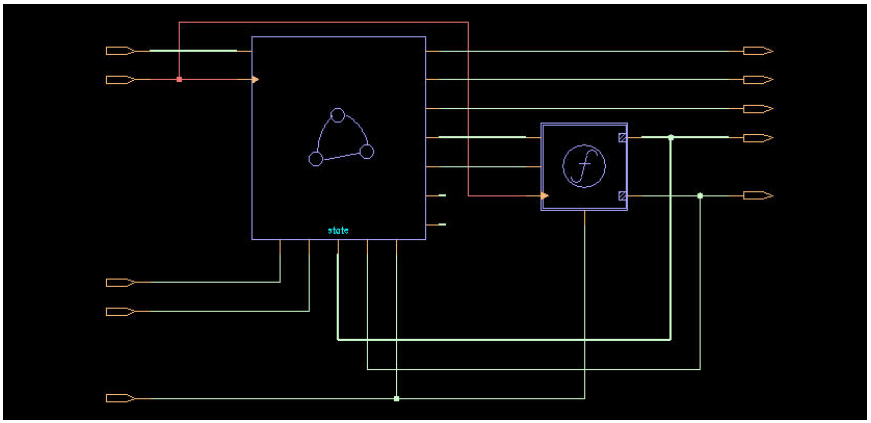


Figure: Example of FSM Symbol in the nSchema Window

Start nState

1. Change the directory to <*working_dir*>/demo/verilog/cpu/src, and execute the following command to import the FSM design:

   ```
   % verdi -f run.f -workMode hardwareDebug &
   ```

2. Open a schematic window by double-clicking to select *i_BJkernel* in the **Instance** tab of the design browser frame.
3. Click the **New Schematic** icon  on the toolbar. The *nSchema* window is displayed, as shown in the *Example of FSM Symbol in the nSchema Window* figure above.

NOTE: If the *nSchema* window does not look like the above figure, confirm the **Enable Detail RTL** option is turned on. This option is available on the **RTL** page under the **Schematics** folder of the *Preferences* form (invoked with the **Tools -> Preferences** command).

---

4. Double-click the FSM block  in the *nSchema* window to view the state diagram in an *nState* frame. An *nState* frame opens as a new tab in the same area as the *nSchema* window:

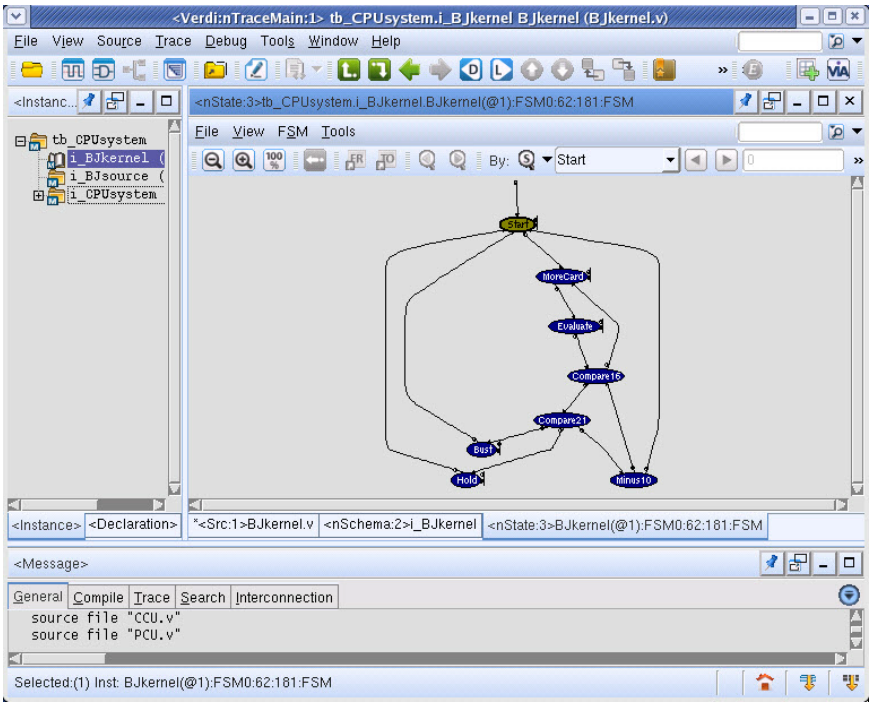


Figure: nState Frame Displaying FSM

Manipulate the State Diagram View

You can change the view of the state diagram using the following zoom commands:

- **Zoom In** - You can view more details of the state diagram by moving the view 50% from the center point in both the horizontal and vertical directions. Invoke this command using one of the following methods:
  - Toolbar icon
  - Bind key **Z**
  - **View -> Zoom -> Zoom In** command
- **Zoom Out** - You can view more contents of the state diagram by expanding the view 2X from the center point, both horizontally and vertically. Invoke this command using one of the following methods:
  - Toolbar icon
  - Bind key **z**
  - **View -> Zoom -> Zoom Out** command
- **Zoom All** - You can view the entire contents of the state diagram. Invoke this command using one of the following methods:
  - Toolbar icon
  - Bind key **f**
  - **View -> Zoom -> Zoom All** command
- **Zoom Area** - You can view a specific area of the state diagram by dragging-left to form a rectangle over the zoomed area.

You can move the viewing area of the state diagram in different directions using the following methods:

- **Scrolling** - Click or drag the scroll bar of the *nState* window either horizontally or vertically.
- **Panning** - Move the view up, down, left, or right using the arrow keys or menus: **View -> Pan -> Pan Up**, **View -> Pan -> Pan Down**, **View -> Pan - > Pan Left**, and **View -> Pan -> Pan Right**.

In addition, you can use the **View -> Last View** command or the bind key "l" (lowercase L) to return to the last view.

Enable Viewing Objects

You can enable or disable viewing for different objects (for example, state actions, transition conditions, and so on) in the *nState* frame.

1. In *nState* frame, choose **Tools -> Preferences** command to open the *Preferences* form.
2. Select the **View Options** page under the **FSM** folder and turn on the **State Action**, **Transition Action**, and **Transition Condition** options.
3. Click **OK** to close this form and apply the changes.
4. Choose **Tools -> Duplicate Window** command to open another *nState* frame and see the additional information. The *nState* frame is displayed as follows:

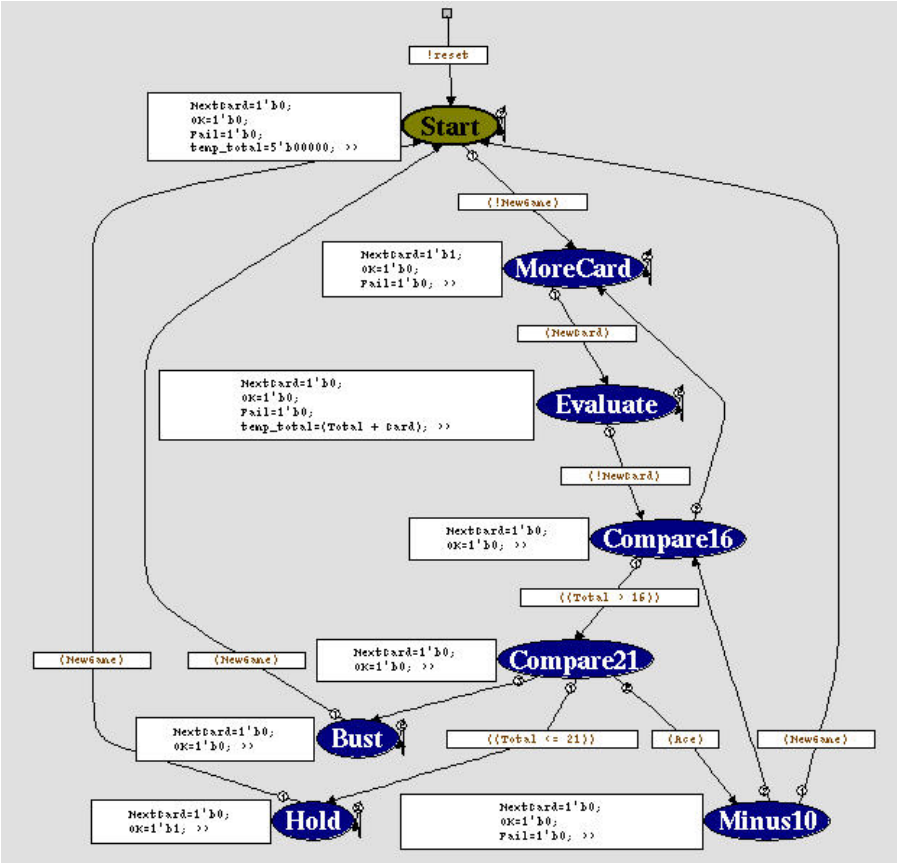


Figure: nState Frame With Viewing Objects

5. In the original *nState* frame, open the **View** menu and select the **State Action**, **Transition Action**, and **Transition Condition** commands individually. The state action and transition details are added to this view.
6. Choose **Tools -> Preferences** command to open the *Preferences* form again, select the **View Options** page under the **FSM** folder. Turn off the **State Action**, **Transition Action**, and **Transition Condition** options, and click **OK** to close the form.

The options on the *Preferences* form affect all new *nState* frames - a global setting. The **View** menu only affects the current *nState* frame - a local setting.

Find the Start and End States of a Transition

You can view the state that a transition is coming from and going to by clicking the toolbar commands.

1. Click any transition in the *nState* frame.

   For example, the transition leaves from the *Evaluate* state with a transition condition of (*!NewCard*). Notice that the starting point of a transition arrow represents the starting point of the transition.

   After you have selected a transition, the **Jump to From State** and **Jump to To State** icons on the toolbar are enabled.
2. Click the **Jump to From State** icon  to see where the selected transition is from.

   The *Evaluate* state is highlighted with red color, that is, the selected transition is starting from the *Evaluate* state.
3. Click the **Jump to To State** icon  .

   The *Compare16* state is highlighted with red color, which indicates that the selected transition ends in the *Compare16* state.
4. Click any of the transitions to view what state the transition is coming from and going to.
5. Select any transition and use the middle mouse button to drag the state and drop it in the source code frame. The transition is automatically located and loaded into the source code frame. All the statements associated with the transition are highlighted.
6. Repeat the drag and drop action for any state for example, the *Compare16* state. The statements associated with the state are automatically highlighted.

Create a Partial Finite State Machine Frame

When your state machine is very large with many states, you may focus on a portion only.

1. Select the *MoreCard* state.
2. Press and hold the <Shift> key, and select states *Evaluate* and *Compare16*.
3. Choose **Tools -> Partial FSM** command to open a new *nState* frame showing only the selected states.

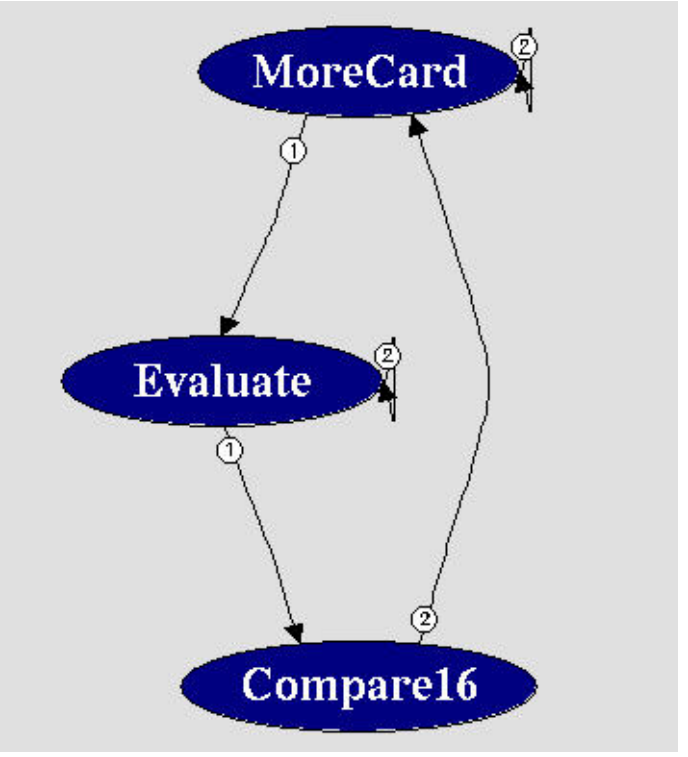


Figure: nState Frame Displaying Partial FSM

4. Click the **Undock** icon in upper right to make the partial FSM frame a standalone window. From the full *nState* frame, select the *Compare21* state and drag it to the partial frame.

   The state connection is added.

5. Close the Partial FSM frame.

State Animation

One of the challenges of state machine debug is seeing how the state machine changes based on the current stimulus. You can use *nState*, *nWave,* and **State Animation** to understand the flow.

Continue the following steps with an opened *nState* frame.

1. In the *nTrace* main window, choose **File -> Load Simulation Results** command to open the *Load Simulation Results* form and load the simulation results.
2. Select the *<working_dir>/demo/verilog/cpu/CPUsystem.fsdb,* and click **OK** to load the simulation result file.
3. Click the **New Waveform** icon in the *nTrace* main window, the *nWave* frame displays in the bottom.

   The simulation file loaded in Step 1 is used as the loaded simulation result file in *nWave*.

NOTE: For information on getting signals into the *nWave* window, refer to the *nWave Tutorial* chapter.

4. In *nSchema*, select the FSM symbol and use the middle mouse button to drag and drop to *nWave*. The FSM signals are added, as shown in the following figure:

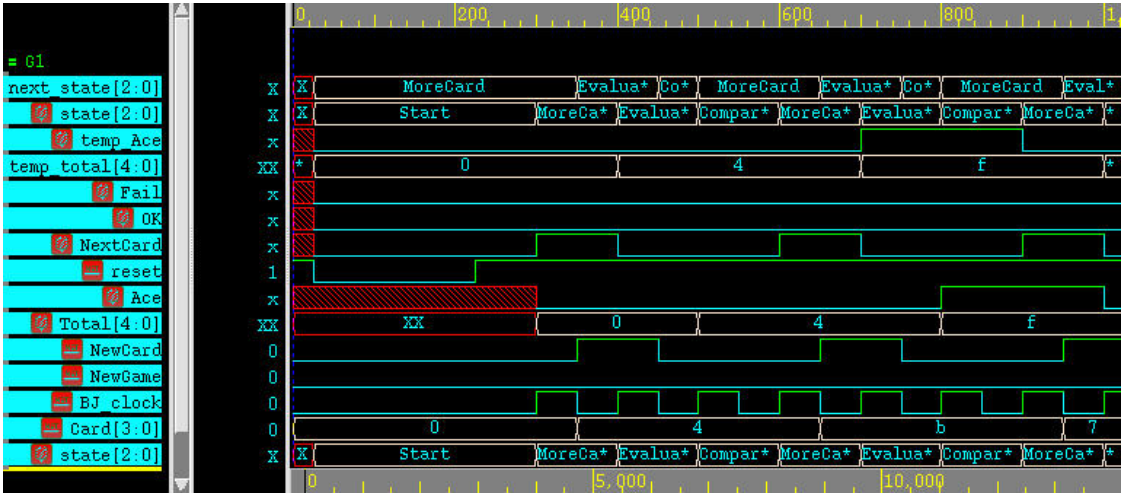


Figure: nWave Frame Displaying FSM Signals

NOTE: The state variable signal displays state aliases.

5. After the simulation results are loaded, turn on the **FSM -> State Animation** toggle command in the *nState* frame to enable state animation.

   The *nState* frame shows the value changes by highlighting the state(s) and transition, while the *nWave* window shows the value changes in the waveform format. Compare the state signal *State[2:0]* in the waveform pane of the *nWave* window with the state signal in the *nState* window. The following figure shows *nState* and *nWave* at cursor time 700.

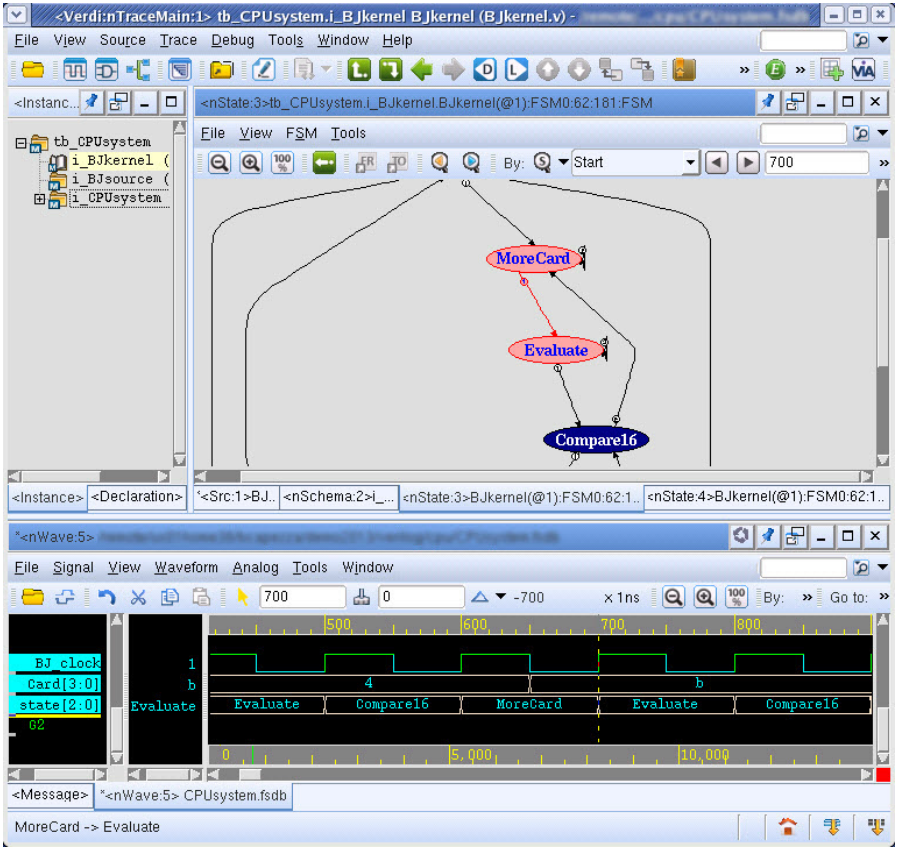


Figure: nState and nWave During State Animation

6. In *nState*, change the cursor time to *300ns* by typing *300* in the **Cursor Time** text box on the toolbar and press **Enter** on the keyboard.

   nState highlights in pink the state(s) and the transition at the specified cursor time in the *nState* frame. The *Start* state, *MoreCard* state are highlighted. The cursor time in the *nWave* window synchronizes with the **Cursor Time** text box in the *nState* window.

7. Click the **Next State** icon  to move to the next state event.

The *MoreCard* state, the *Evaluate* state, and the transition in between are highlighted. Notice that the cursor time changes to *400*. This time change indicates that the next value change occurs at the cursor time of *400*.

8. Click the **Previous State** icon  .

   The *Start* state, the *MoreCard* state, and the transition in between are highlighted, and the cursor time is *300*.

NOTE: Sometimes when the next/previous state is the same as the current state, the highlight on the *nState* frame remains the same but the cursor time changes.

In addition to **Next State** and **Previous State** icons, the **State List** text box can be used to go to a state. Following are the functions of the **State List** box:

- Finding the state in the *nState* window.
- Defining the state or state sequence for the **Search Forward** and **Search Backward** icons.

9. Set the **Search By**  criteria to *State* in the toolbar of the *nState* frame.
10. Use the **State List** box to find the state in the *nState* window. For example, select *Evaluate* in the **State List** box.

    nState finds and highlights the *Evaluate* state in the *nState* window (the state changes from a blue background to a red background).

11. Click the **Search Backward** icon  and **Search Forward** icon  .

    The **Cursor Time** text box shows the time when the state signal value changes for the *Evaluate* state.

12. Choose **FSM -> Edit Search Sequences** command to open the *Search* form.
13. In the *Search* form, click **New** to open the *New Search Sequence* form.
14. Type *test* in the **Name** text field.
15. Double-click *Morecard*, *Evaluate*, and *Compare16* (or single-click and then add by clicking on the **Add State** icon).
16. Click **OK** on the *New Search Sequence* form and **Close** on the *Search* form.

    The **Search By** automatically changes to **Search by Sequences** and the state list box displays the sequence, *test*.

17. Click the **Search Backward** and **Search Forward** icons.

    The **Cursor Time** text box shows the time when the state sequence occurs. This sequence occurs several times during this simulation.

State Machine Analysis

After the simulation results are loaded, you can analyze the state machine further based on the stimulus used.

1. Choose **FSM -> Analysis Report** command to open the *Analysis Report* form as shown in the following figure. Notice that there are three tabs - with the **Source Code** tab as the default tab.
2. Click the **State** tab.

Figure: Analysis Report Window

The number of times a *state* is accessed during the simulation is summarized. Also, any *states* that are not covered are listed.

3. Click the **Transition** tab.

   The number of times a transition is accessed during the simulation is summarized. Also, any transitions that are not covered are listed.

4. In the *nState* frame, choose **File -> Close Window** command to close the FSM window.

Smart Log Tutorial

Overview

The Smart Log engine can be used to read and traverse log files in a readable view. Smart Log loads a log file with a specific format that shows a structured presentation of the log contents, thus providing a flexible and convenient use model for debugging. The connection between the log file and the Verdi platform is built according to the hyperlink rule files and the partitioning rule files.

Click the hyperlink to the log content in the *Smart Log* window to link to the Verdi platform, for example, link to the source code location. The hyperlink rule can be further customized with Tcl commands to execute a specific action in Verdi. A partitioning rule can also be used to group the messages in Smart Log for systematic and convenient log navigation.

Figure: Smart Log shows an example where you can jump to the specified time and click the hyperlink in the *Smart Log* pane to directly link to the *Source Code* pane in Verdi.

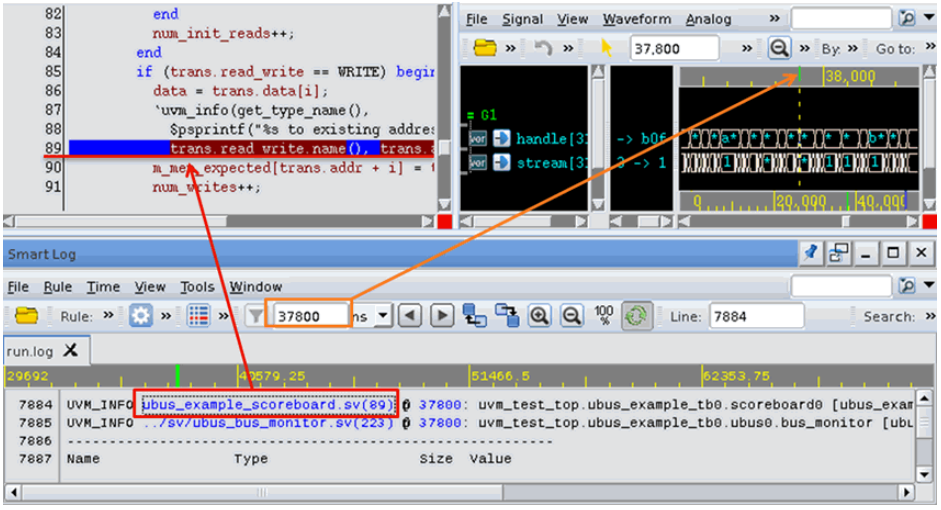


Figure: Smart Log

Smart Log offers the following features:

- Two browsing views to display log in the following modes:
  - *File View* for sequential information
  - *Structure View* for grouped information
- Identify and filter out the desired log messages
- Synchronize with the *cursor* in the *nWave* frame
- Unified usage for *interactive simulation console*
- Customization of hyperlink rules and partitioning rules

This chapter comprises of the following sections:

Invoking Smart Log

Figure: Smart Log Flow illustrates the stepwise flow for invoking Smart Log and loading a log file to Smart Log.

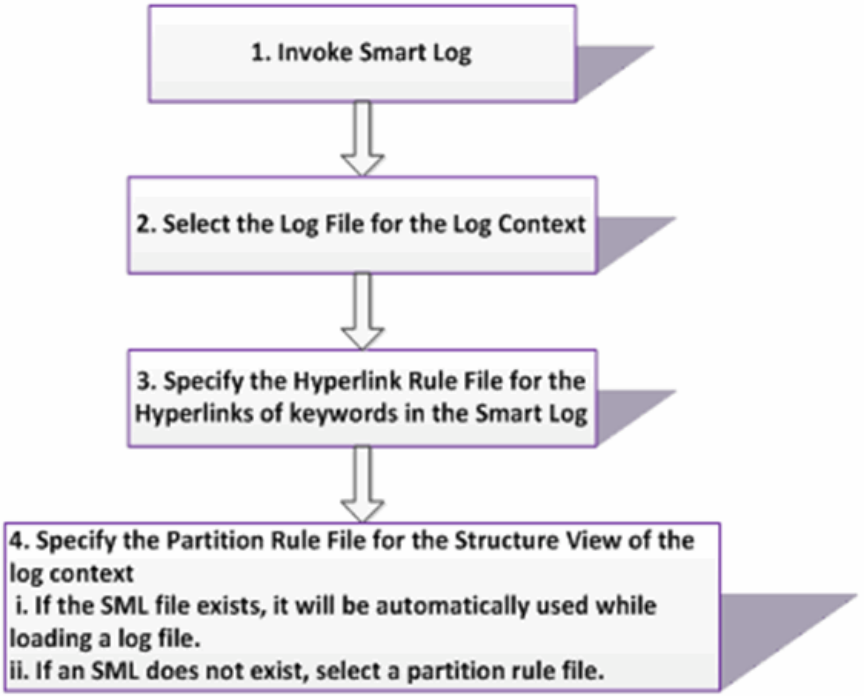


Figure: Smart Log Flow

The following steps describe how to invoke Smart Log and load a log file:

1. Click the **Smart Log** icon to open the *Smart Log* pane in the Verdi main window, as illustrated in *Figure: Invoke Smart Log Using Smart Log Icon*.

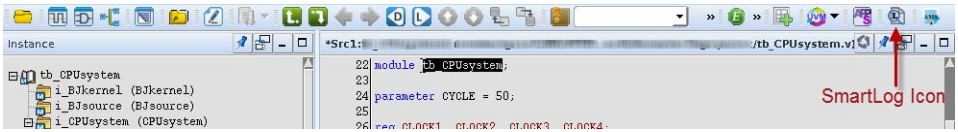


Figure: Invoke Smart Log Using Smart Log Icon

The Smart Log pane opens as illustrated in *Figure: Smart Log Pane*.

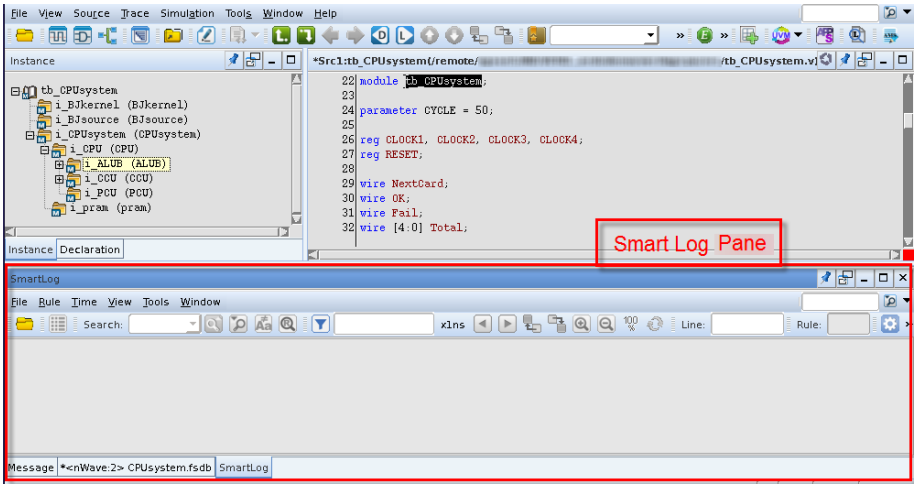
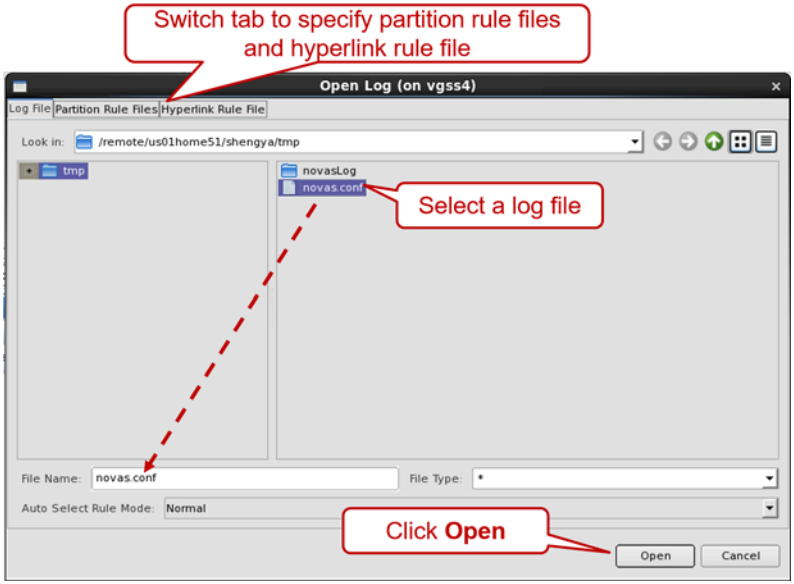


Figure: Smart Log Pane

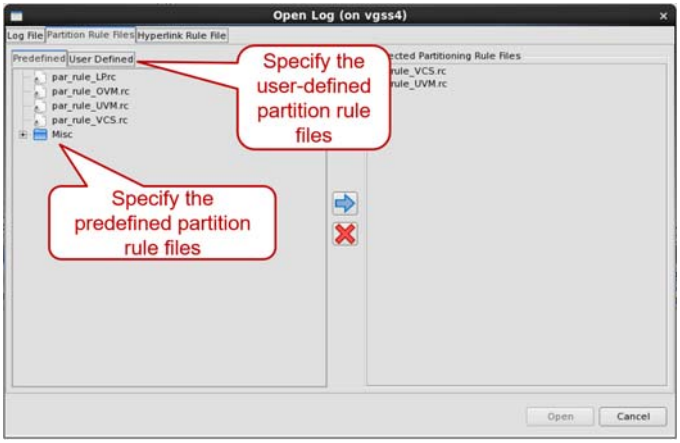
2. Click the  icon in the *Smart Log* pane to open a log file. The *Open Log* form is invoked where you can select a log file in the *Log File* tab, as illustrated in the following figure:



To specify partition rule files, click the *Partition Rule Files* tab.

- Specify a predefined partition rule file in the *Predefined* tab; click the file you want to open and click on the arrow to include it to the *Selected Partitioning Rule Files* section.

- Specify a user-defined partition rule file in the *User Defined* tab; click the file you want to open, click on the arrow to include it to the *Selected Partitioning Rule Files* section**.**

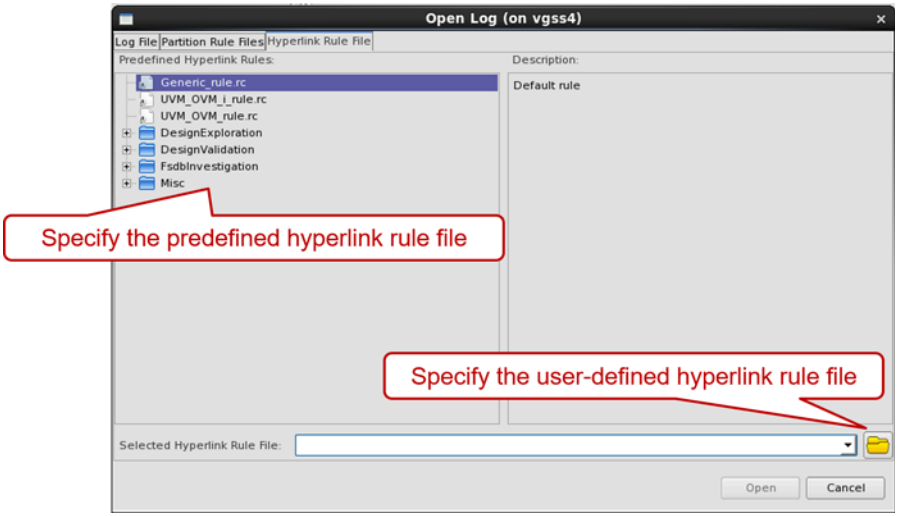


The *User Defined* tab in the *Partitioning Rule Files* section lists all directories and files with the *.rc* extension under the root directory. You can configure the root directory and the change is saved in the *novas.rc* file.

Smart Log records the previously selected partitioning rule files in the *novas.rc* file and displays these rule files in the *Selected Partitioning Rule Files* section by default.

To specify a hyperlink rule file, click the *Hyperlink Rule File* tab,

- Select a predefined hyperlink rule file in the *Predefined Hyperlink Rules* section.
- Select a user-defined or customized hyperlink rule file using the  icon.



3. Click the **Open** button in the *Open Log* form.

   If there is an SML file for the specified log file, the SML file is automatically used for partitioning the log file in Structure View.

NOTE: If the log file is generated by VCS with the `-sml -l <log_file_name>` simulation option, <*log_file_name*>*.sml* is also generated at the same time.

If the selected log file has a corresponding SML file, the **Partitioning Rule Files** section is hidden.

Figure: Displaying a Log File in Smart Log illustrates an example of Smart Log displaying a UVM log file with the rule file.

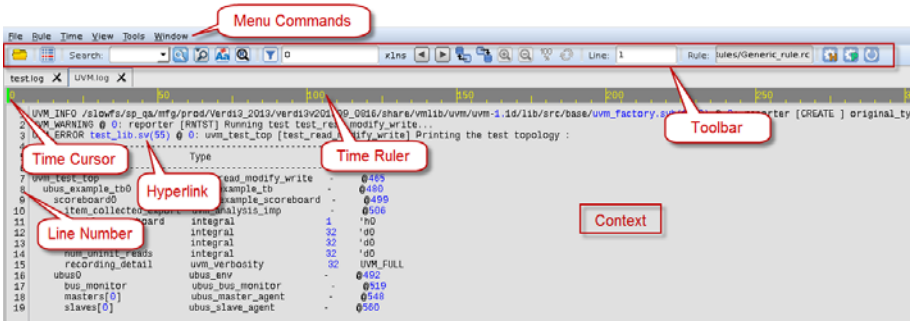
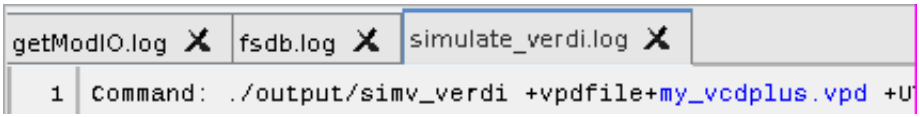


Figure: Displaying a Log File in Smart Log

Opening multiple log files is also supported. Each log file is loaded in the corresponding tab. The hyperlink rule file and the partitioning rule file specified in different tabs are independent.



Auto Select Rules

This section describes the auto select rules in the following sub-sections:

- *Checking Mapping Rule File*
- *Predicting Mapping Rule*
- *Auto Select Rule Mode*

Checking Mapping Rule File

A mapping rule file contains multiple mapping rule sets and a mapping rule set contains the following three items:

1. *patternInFileName* (regular expression) or *patternInLogHeader* (regular expression)
    1.1 *patternInFileName*: is used to match the log's file name
    1.2 *patternInLogHeader*: is used to match the log's first 30 lines
2. *hyperlinkRule*: is applied if the pattern matches
3. *partitionRules*: is applied if the pattern matches

The absolute path to the mapping rule file is specified in *novas.rc*, and *SmartLog* parses the specified file.

```
novas.rc

...
[VIA.smartLog.preference.mappingRuleFile]
vgifMappingRuleFile = absolute/path/to/mapping/rule/file
...
```

Example

- If the file name of the selected log starts with "sim" and ends with ".log", for example, "simulation.log", the following mapping rule set gets selected:

```
patternInFileName =     ^sim.*\.log$

hyperlinkRule =         absolute/path/to/hyperlinkRule_1.rc

partitionRules =        absolute /path/to/partitionRule_1.rc \
                        absolute /path/to/partitionRule_2.rc
```

- If any line of the first 30 lines of the log file contains "listModFile", the following mapping rule set gets selected:

```
patternInLogHeader =    listModFile

hyperlinkRule =         absolute/path/to/hyperlinkRule_1.rc

partitionRules =        absolute /path/to/partitionRule_1.rc \
                        absolute /path/to/partitionRule_2.rc
```

Predicting Mapping Rule

SmartLog predicts suitable partition rule files and hyperlink rule file for the selected log, if the selected log does not match any mapping rule set in the mapping rule file.

The user-defined partition rule files are included in predicting a rule, if the directory which contains the user-defined partition rule files is specified in the *novas.rc* file.

```
novas.rc

...
[VIA.smartLog.preference.mappingRuleFile]
vgifUserDefinedParRuleDir = absolute/path/to/directory
...
```

Auto Select Rule Mode

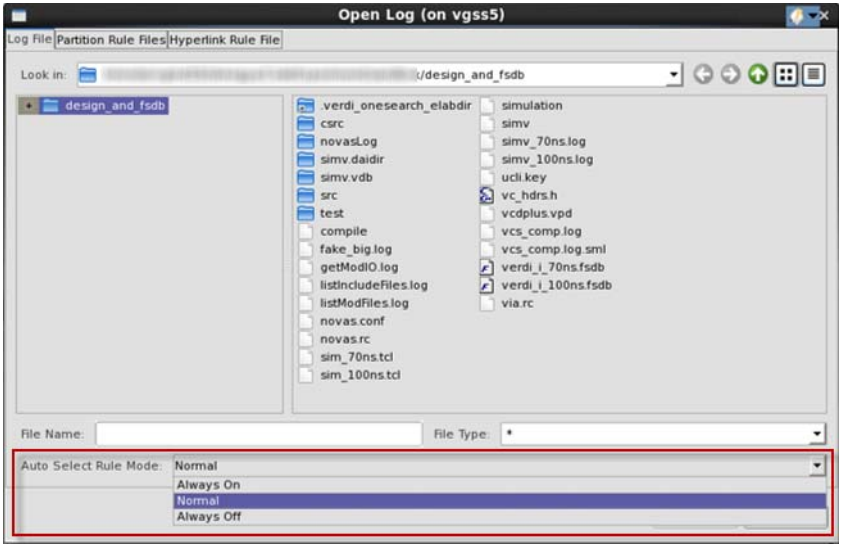
You can change the auto select rule mode using the *Auto Select Rule Mode* field in the *Open File* form.

The following options are available in the *Auto Select Rule Mode* field:

- **Always On**: Enables auto-select rule.
- **Normal**: Enables auto-select rule.

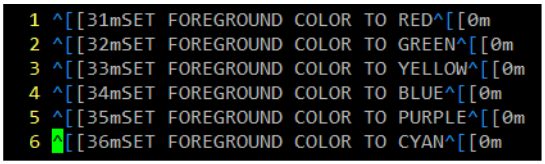
  The **Normal** mode changes to the **Always Off** mode, if any partition rule files or hyperlink rule file is added or removed in the *Open Log* form and if the **Open** button is clicked in the *Open Log* form.
- **Always Off**: Disables auto select rule.

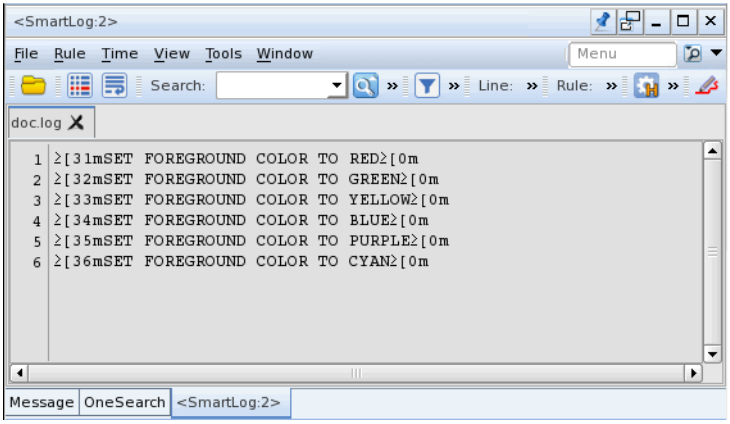
The *Auto Select Rule Mode* field is illustrated in the following figure:



Displaying ANSI Colors in SmartLog

If a log file has an ANSI flag, by default, *SmartLog* shows the ANSI flag directly, as illustrated in the following figure:

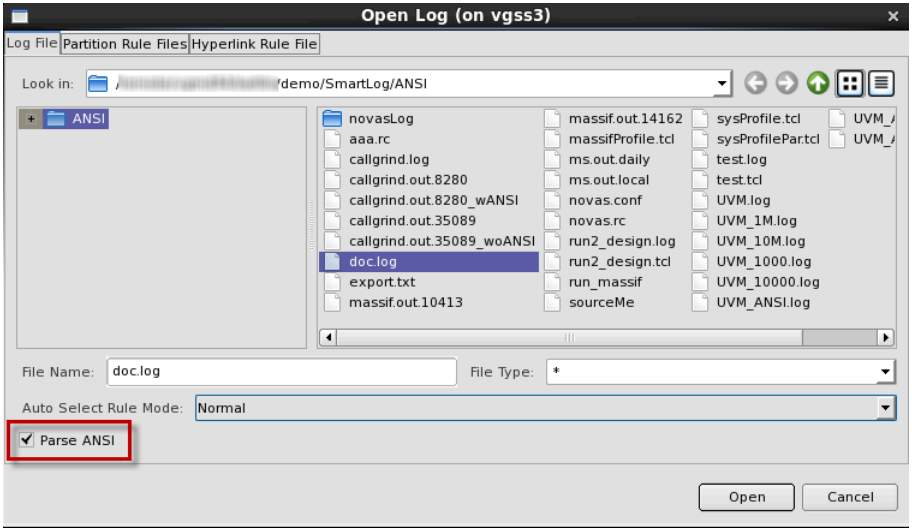




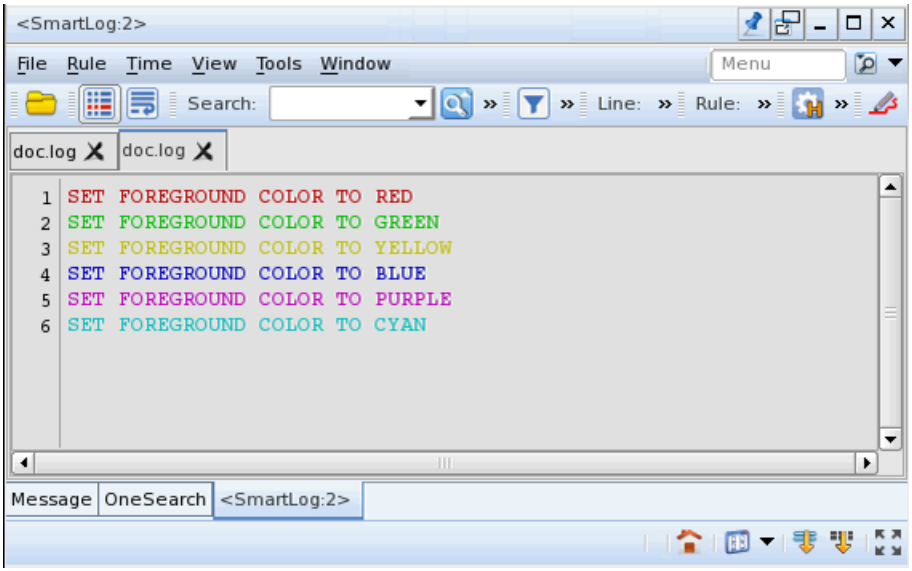
To enable *SmartLog* to parse the ANSI flags present in a log file and display ANSI colors in the opened log file, perform the following steps:

1. In the *Open Log* form, enable/check the **Parse ANSI** check box.
2. Select a log file and click **Open**.

The *Open Log* form is illustrated in the following figure:



On selecting the **Parse ANSI** check box, *SmartLog* displays the log file with ANSI colors as follows:



NOTE: Enabling the **Parse ANSI** option while opening a log file may cause performance issue, therefore, you must enable it only if required.

Opening a Log File Using Verdi Commands

You can use the following commands to open a log file in Smart Log:

- `-smlog logfile`

  Loads the specified log file with Smart Log.

- `-smlog_hyper | -smlog_h ruleFile`

  Specifies the hyperlink rule file for the specified log.

  NOTE: This option must be used with the `-smlog` option.

- `-smlog_partition | -smlog_p "ruleFile1 ruleFile2...ruleFileN"`

  Specifies the partitioning rule file(s) for the specified log. A pair of double quotes (") must be used to enclose the partitioning rule file(s).

  NOTE: This option must be used with the `-smlog` option.

Examples

The following are some examples to demonstrate use cases to open log file(s) using Verdi commands:

Example 1 - To open a specified log (*test.log*) with specified partitioning rule file (*parRule1.rc*) and hyperlink rule file (*hyperRule.rc*), use the following command:

```
%> verdi -smlog test.log -smlog_hyperlink hyperRule.rc
-smlog_partition “parRule1.rc”
```

Example 2 - To open a specified log (*test.log*) with specified partitioning rule files (*parRule1.rc*, *parRule2.rc*) and hyperlink rule file (*hyperRule.rc*), use the following command:

```
%> verdi -smlog test.log -smlog_hyperlink hyperRule.rc
-smlog_partition “parRule1.rc parRule2.rc”
```

Example 3 - To open specified log files (*test1.log*, *test2.log*) with the same specified partitioning rule files (*parRule1.rc*, *parRule2.rc*) and hyperlink rule file (*hyperRule.rc*), use the following command:

```
%> verdi -smlog test1.log -smlog test2.log -smlog_hyperlink
hyperRule.rc -smlog_partition “parRule1.rc parRule2.rc”
```

Example 4 - To open multiple log files (*test1.log*, *test2.log*) with different partitioning rule files and hyperlink rule files, use the following command:

```
%> verdi -smlog test1.log -smlog_hyperlink hyperRule1.rc
-smlog_partition “parRule1.rc” -smlog test2.log
```

```
-smlog_hyperlink  hyperRule2.rc -smlog_partition
“parRule2.rc”
```

NOTE: You can also use the `viaLogViewerOpenLog` Tcl command to open a log file. For details on SmartLog Tcl commands, see the *SmartLog* chapter in *Verdi and Siloti Tcl Reference Guide*.

Navigating Smart Log

Smart Log provides a toolbar with frequently used commands to navigate the log files, as illustrated in *Figure: Smart Log Layout*.

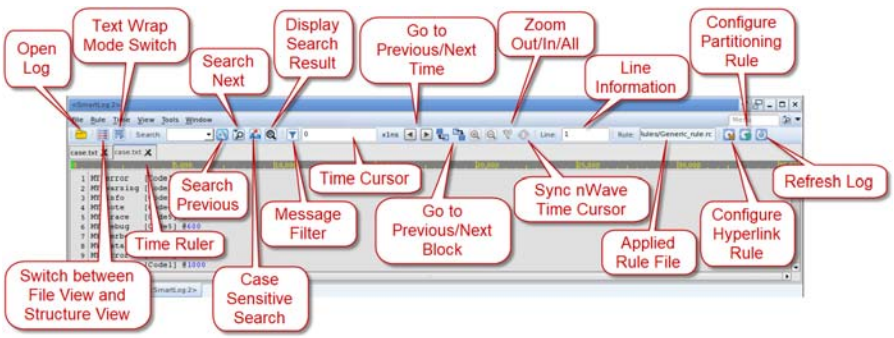


Figure: Smart Log Layout

The icons in the toolbar can be used to navigate the displayed content in Smart Log for the currently displayed log file.

Shortcut Keys for Smart Log

Following is the list of shortcut keyboard keys that can be used to navigate through the Smart Log tabs:

- **Ctrl** + **PageDown** - Use this shortcut to move to the next Smart Log tab.
- **Ctrl** + **PageUp** - Use this shortcut to move to the previous Smart Log tab.
- **Ctrl** + **w** - Use this shortcut to close the Smart Log tab.
- **Ctrl** + **f** - Use this shortcut to select the search field in the Smart Log pane.

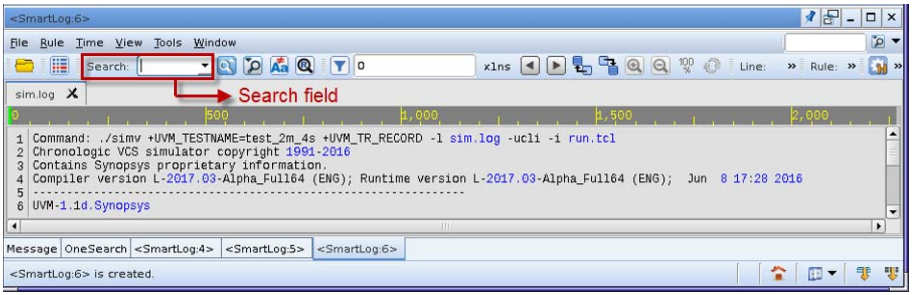


Figure: Search Field in SmartLog Window

Wrapped Text

In the *SmartLog* window, if the text is a long string which is too long to be displayed in the window, you can enable the text wrap functionality to fit the text in the available width of the window. To enable the text wrap functionality, click the **Text Wrap Mode Switch** icon, as illustrated in the following figure:

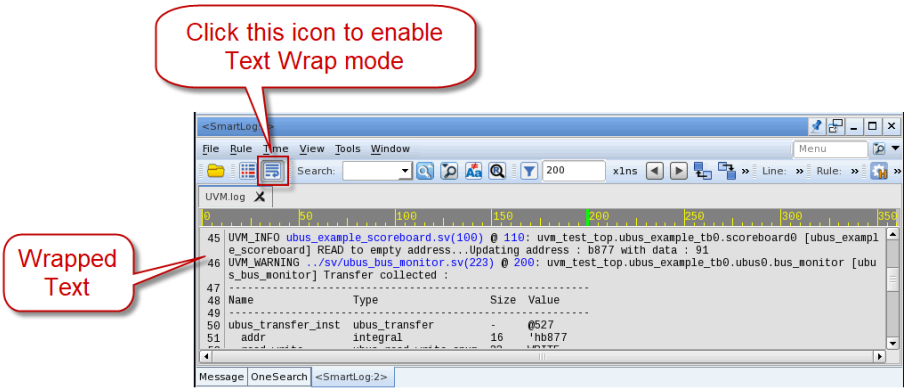


Figure: Text Wrap Mode in SmartLog Window

The wrapped text is also displayed in the ToolTip that appears next to it, as illustrated in the following figure:

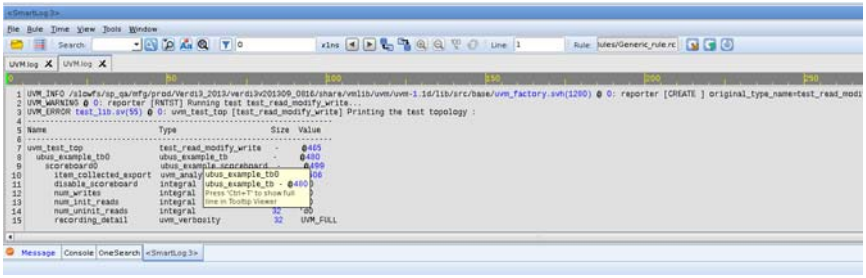


Figure: ToolTip in SmartLog Window

You can use the **Ctrl +T** shortcut key to open the *ToolTip Viewer* and read or copy the complete string. If the long string contains a hyperlink, you can view and click the hyperlink in the *ToolTip viewer*. The following figure shows the *ToolTip viewer*:

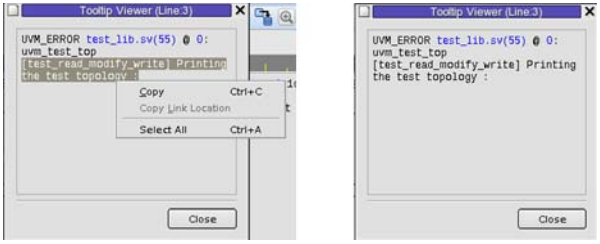


Figure: ToolTip Viewer

Browse Views

Smart Log has two browse views: *File View* and *Structure View*, which display the context of the log files. After loading a log file into the *Smart Log* pane, click the  icon to switch *File View* and *Structure View* to display the context. These views are described in the following sections:

- File View
- Structure View

File View

As illustrated in *Figure: Displaying a Log File in Smart Log*, *File View* displays the log text row by row, accompanied with some keywords as hyperlinks.

When a hyperlink is clicked upon, the corresponding action described in the hyperlink rule file is triggered. Smart Log uses the hyperlink rule written in a **.rc* file to define where to set the hyperlink, and what to do after clicking the hyperlink keyword. Click the  icon to configure an existing hyperlink rule or customize your own hyperlink rule (for details about the usage of the hyperlink rule file, see the Using Hyperlink Rule File section).

Structure View

The *Structure View* displays the content of the log file in a spreadsheet. As illustrated in *Figure: Structure View*, the content is grouped into several blocks based on attributes (such as time, severity, file name and so on) specified in the partitioning rule file. Each block delivers one message and contains time, severity, and entire content for the message. For details about the partitioning rule file, see the Applying Partitioning Rule section.

Figure: Structure View

The following columns are present in the *Structure View*:

- **Time**

  The simulation time of a block in the log file is displayed in the **Time** column, when the specified partitioning rule file contains the *Time* attribute. If the partitioning rule file does not contain the *Time* attribute, the value 0 is displayed in the field for each block.

NOTE: The value of the *Time* attribute is assumed as an unsigned integral type. The values should monotonically increase with the rows in the log file.

- **Severity**

  The severity of a block is displayed in the **Severity** column.

  The font color of the block is determined by the severity type, as follows:

  - If the severity type is *Fatal* or *Error*, the block font color is red.
  - If the severity type is *Warning*, the block font color is orange.
  - In other cases, the block font color is black.

  You can also customize the displayed text and the text color for the severity list using the **Severity** page in the *SmartLog* folder of the *Preferences* form in Verdi.

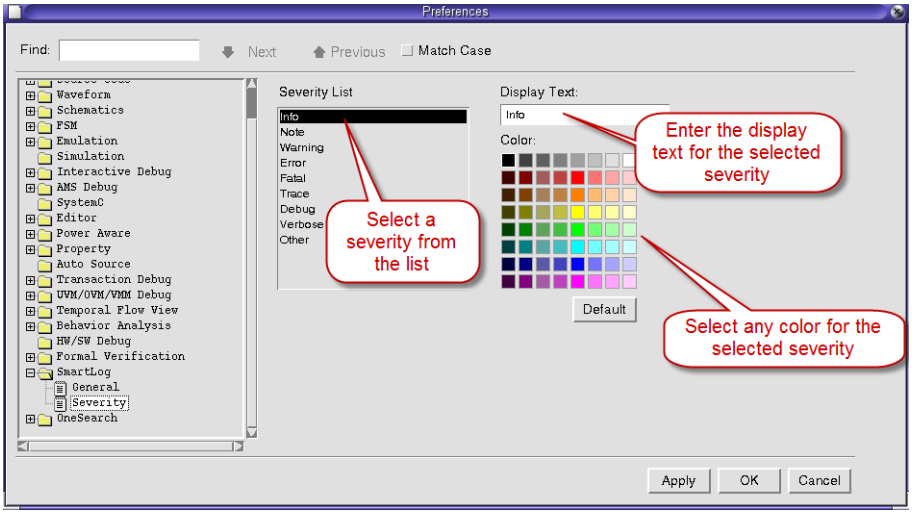


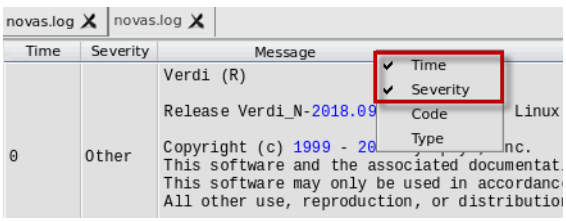
Figure: SmartLog Page in Preferences Form

NOTE: Empty spaces can not be entered in the **Display Text** field.

- **Message**

  The entire content of a block is displayed in the **Message** column. The hyperlink rule is also applied to the text. The behavior in the **Message** column is the same as that in *File View*.

Use **Time** and **Severity** right-click options on the heading of the columns to hide or display the corresponding column.



- **Code**

  The code of a block in the log file is displayed in the **Code** column. This column is hidden by default. To display this column, right-click on the heading of the columns and enable the **Code** option.

- **Type**

  The type of a block in the log file is displayed in the **Type** column. This column is hidden by default. To display this column, right-click on the heading of the columns and enable the **Type** option.

NOTE: You can rearrange the sequence of the columns in the *Structure* view by clicking and dragging a column.
You can restore the last display settings for columns including column width and column sequence from the *novas.rc* file.

User Specified Columns

SmartLog allows users to extract valuable information from a specific log and create a new column in the *Structure* view to store that information.

To add the user specified column,

1. Click the **Configure Partitioning Rule** icon present on the SmartLog toolbar.
2. In the *Rule Pattern Definition* section in the *Configure Partitioning Rule* form, click the icon.

3. In the *Configure User Specified Types* form, edit the user specified types by clicking **Add** and **Delete** buttons.

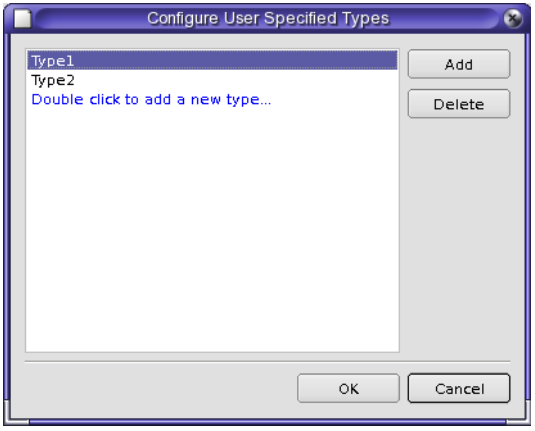
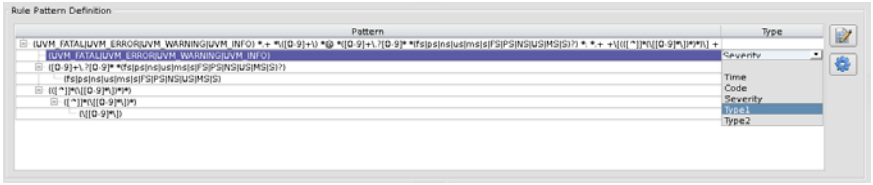


Figure: Configure User Specified Types Form

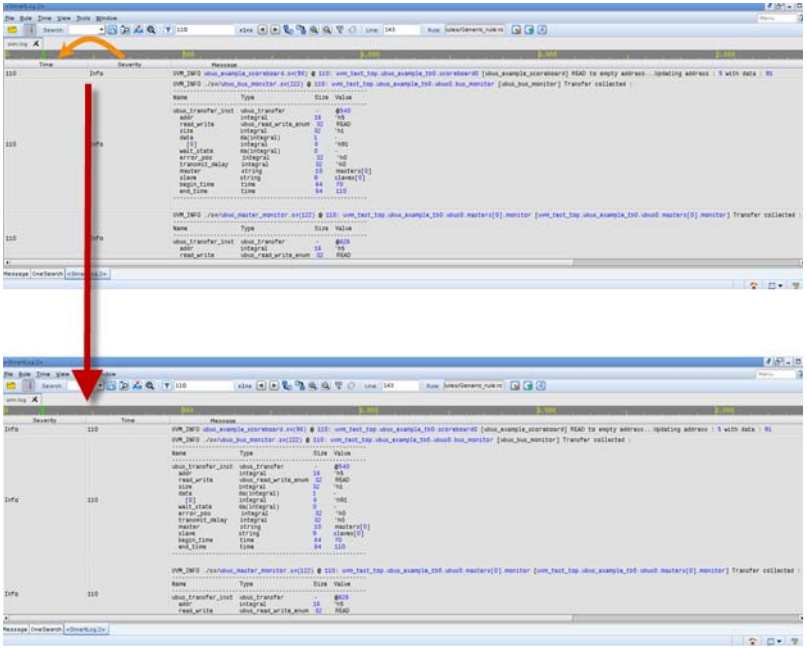
Once the configuration is done, the new types are added in the dropdown list, as illustrated in the following figure:



The user-specified columns (if any) are displayed by default in SmartLog.

Rearranging the Columns

You can also rearrange the column sequence as desired, by clicking and dragging the columns towards left or right. The following figures show rearrangement of **Time** and **Severity** columns.



Restoring Previous Column Display Settings

You can restore the previous settings for display of columns using the *novas.rc* settings file. The width of the columns, sequence of the columns, and the show/hide settings for columns are restored on using the *novas.rc* file.

Specifying Time Unit in UVM/OVM Log File

SmartLog acquires the timing information from the following time units;

- **Log Time Unit**

  The log time unit is applied on time without a unit and affects the time value.

  For example, if the time *10* without unit is recognized by *SmartLog* and log time unit is set as *ns*, *SmartLog* considers it as *10ns*.

  If the log time unit is not specified correctly, it breaks down the synchronization between *SmartLog*, *nWave*, and *Transaction and Protocol Analyzer*.

- **Display Time Unit**

  The display time unit does not affect the time value, it only impacts how the time is displayed.

  For example, if the time is *1000ns*, *SmartLog* shows *1000* with display time unit *ns*, and shows *1* with display time unit *ms*.

Specifying the Log Time Unit

You can specify the log time unit using any of the following methods:

- Click the **Time** -> **Setup Log Time Unit** menu command.
- Specify through the partition rule. For details, see Creating Log Time Unit Rule.

If the log time unit is not fixed, it is recommended to always print time with unit in the log file. For example, in SV HDL code, use `$timeformat` in the testbench to display time with unit in the log file.

Specifying the Display Time Unit

You can specify the display time unit using the **Time** -> **Setup Display Time Unit** menu command.

Using Hyperlink Rule File

For a loaded log file in Smart Log, the specified hyperlink rule file can be manually configured and updated as a new hyperlink rule file in the GUI interface. The configurations of currently loaded hyperlink rule file gets displayed in the *Configure Rule Set* form, when you click on the icon or use the **Rule** -> **Configure Hyperlink Rule** command in the *Smart Log* pane. This is also illustrated in *Figure: Configure Hyperlink Rule Form*.

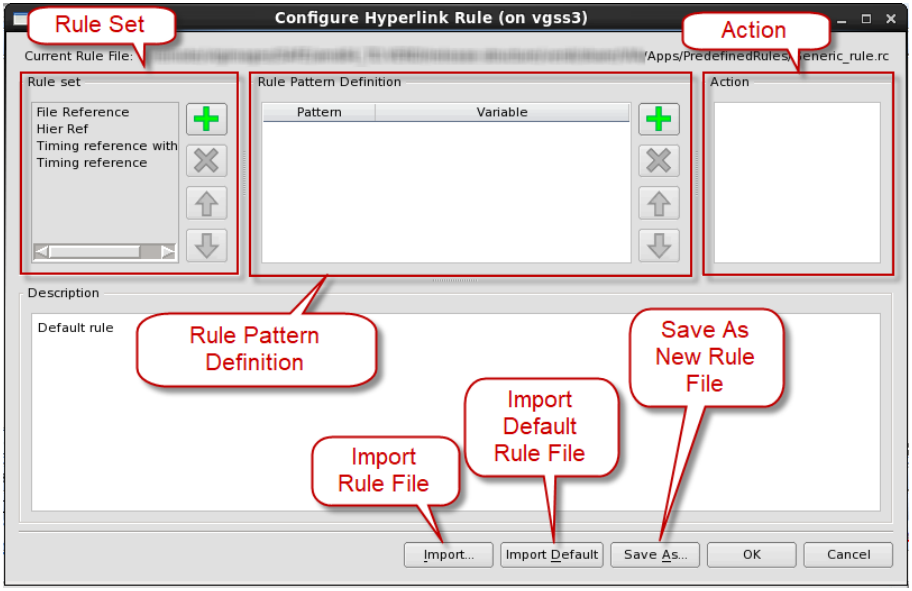


Figure: Configure Hyperlink Rule Form

Click the **Import** or the **Import Default** button to get the configurations in another hyperlink rule file or the default file. After importing the rule file, the configurations are displayed in **Rule Set**, **Rule Pattern Definition**, and **Action** sections. Edit the sections to customize the rules.

The following are rule sets, rule pattern definitions, and actions in the **Configure Hyperlink Rule** form:

- **Rule Set**: This section displays all rule sets in the imported hyperlink rule file. When a rule set is selected in the **Rule Set** section, the corresponding rule pattern is displayed in the **Rule Pattern Definition** section. The rule set in the higher row has a higher priority in this hyperlink rule file. Use or icons to increase or decrease the priority.

Use  or  icons to add or delete a rule set in the **Rule Set** section. Double-click a rule set to change the name of the rule set.

- **Rule Pattern Definition**: This section shows how to bind a hyper link in the log file using regular expressions. If the content in the log file matches the pattern specified in the **Pattern** field, a hyper link binds to the content and is displayed in the *Smart Log* pane, as illustrated in *Figure: Hyperlink Binding*.

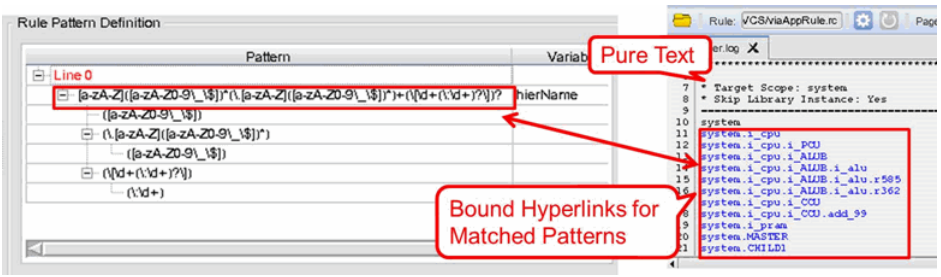


Figure: Hyperlink Binding

When a pattern is selected in the **Rule Pattern Definition** section, the corresponding action is displayed in the **Action** section. The variable name is specified in the **Variable** field for a pattern or sub-pattern as the variable used in the corresponding action. Use  or  icons to add or delete a rule pattern.

If a rule includes multiple patterns, the content in the log file matching pattern is bound to the hyper link line by line. As illustrated in *Figure: Line By Line Pattern*, the matched text in the log file matches the pattern specified in Line 0 and then matches the pattern specified in Line 1 in-order. Use  or  icons to move up or down the orders of the patterns.

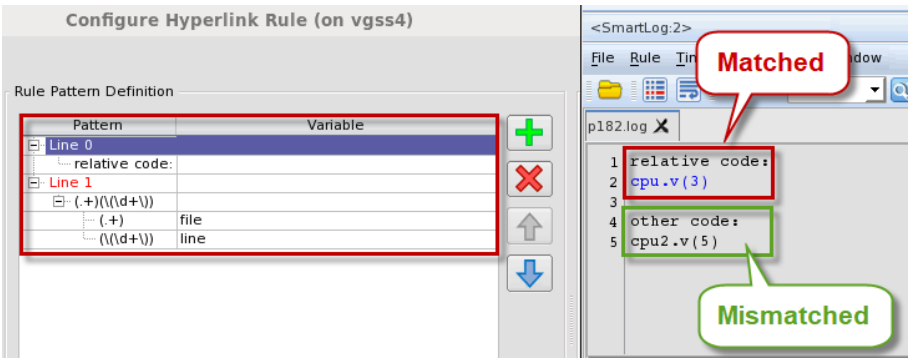


Figure: Line By Line Pattern

- **Action**: This section shows what action (written in Tcl) is executed in Verdi after clicking the hyperlink shown in the *Smart Log* pane. The variable specified in the **Rule Pattern Definition** section can be used in the Tcl commands with the dollar sign ($) character prefix, as illustrated in *Figure: Variable Usage*.

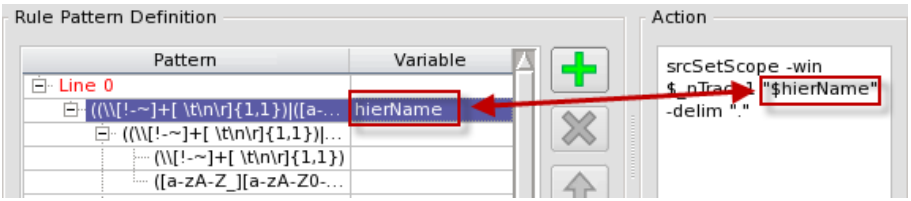


Figure: Variable Usage

After changing the configurations, click the **OK** button to save the changes to the hyperlink rule file and apply the rules to the loaded log file. The current log file and the other log files that use the same hyperlink rule file are automatically refreshed based on the updated hyperlink rule file.

Additionally, use the **Save As** button to save as another hyperlink rule file with *.rc* file name extension.

NOTE: If the *Configure Rule Set* form is closed by clicking the **Cancel** button, *save* and *save as* actions are not reverted.

Configuring a New Partitioning Rule

You can also create or edit multiple partitioning rule files to partition a log into several blocks and extract the properties (Time, Code, Severity) into the blocks. You can save these partitioning rule files so that these can be used based on the requirement later.

Perform the following steps to configure or write a new partitioning rule:

1. Click the **Configure Partitioning Rule** icon present on the Smart Log toolbar. The *Configure Partitioning Rule* form opens up as follows:

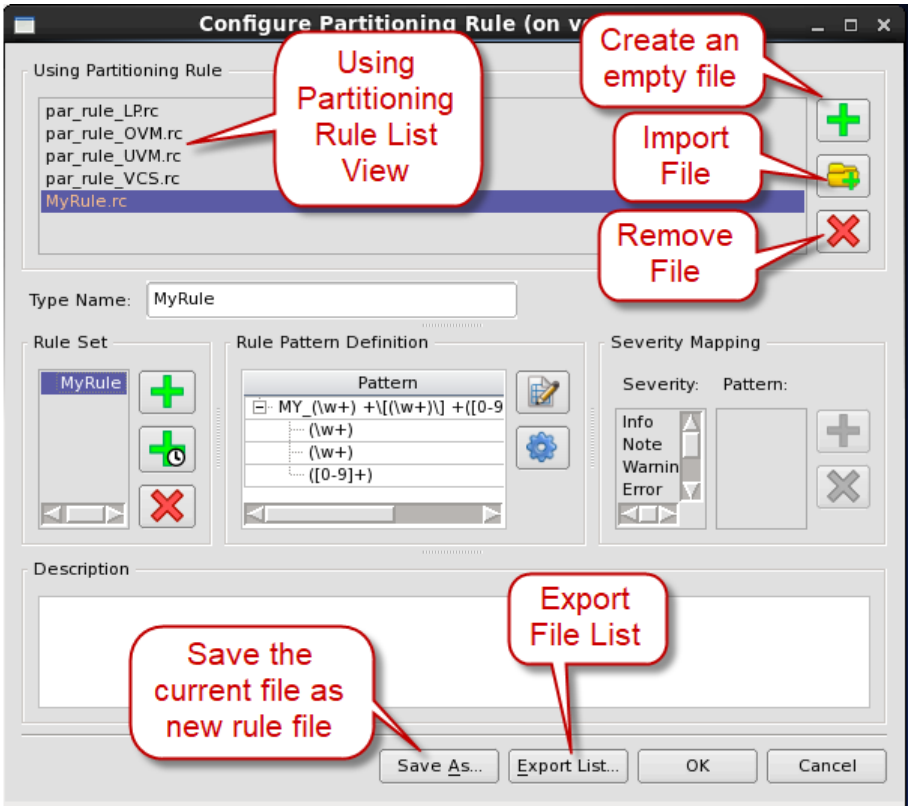
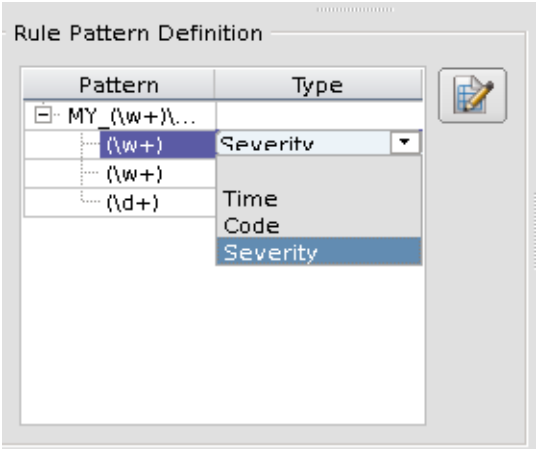


Figure: Configure Partitioning Rule Form

NOTE: Each log file can imply multiple partitioning rule files.
Each partitioning rule file has multiple rules and must have exclusive type name.
Each rule has one pattern.

2. In the *Rule Pattern Definition* field, select the type of sub-pattern using the predefined combo box (as illustrated in the figure below). You can select one of the following types:
   - Severity
   - Time
   - Code

NOTE: Severity/Time/Code can be selected once in one pattern.



Example to Demonstrate Creating a Customized Partition Rule File

If you want to partition your log file, perform the following steps:

1. Click the icon in the *Smart Log* pane to open the log file.
2. In the *Open Log* form, select the log file and click **Open**.

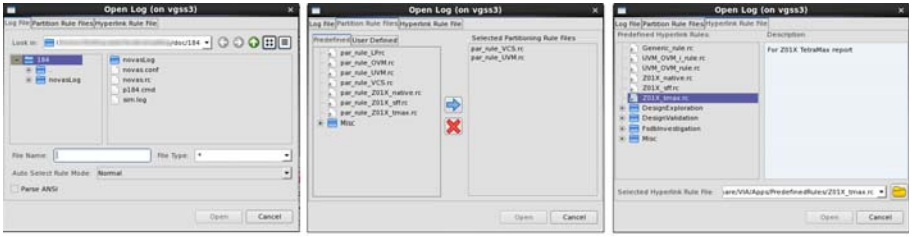
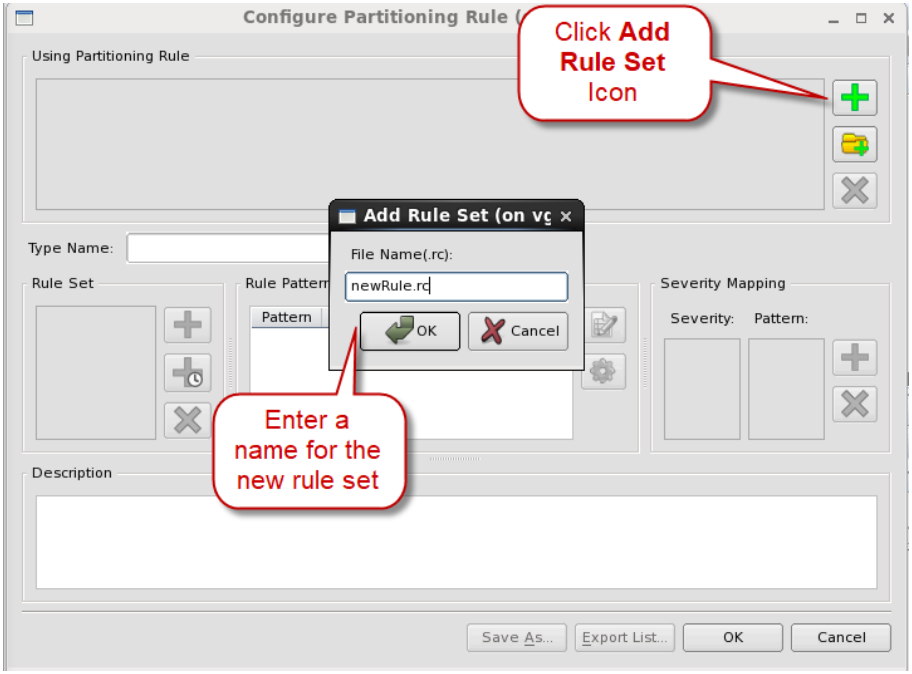


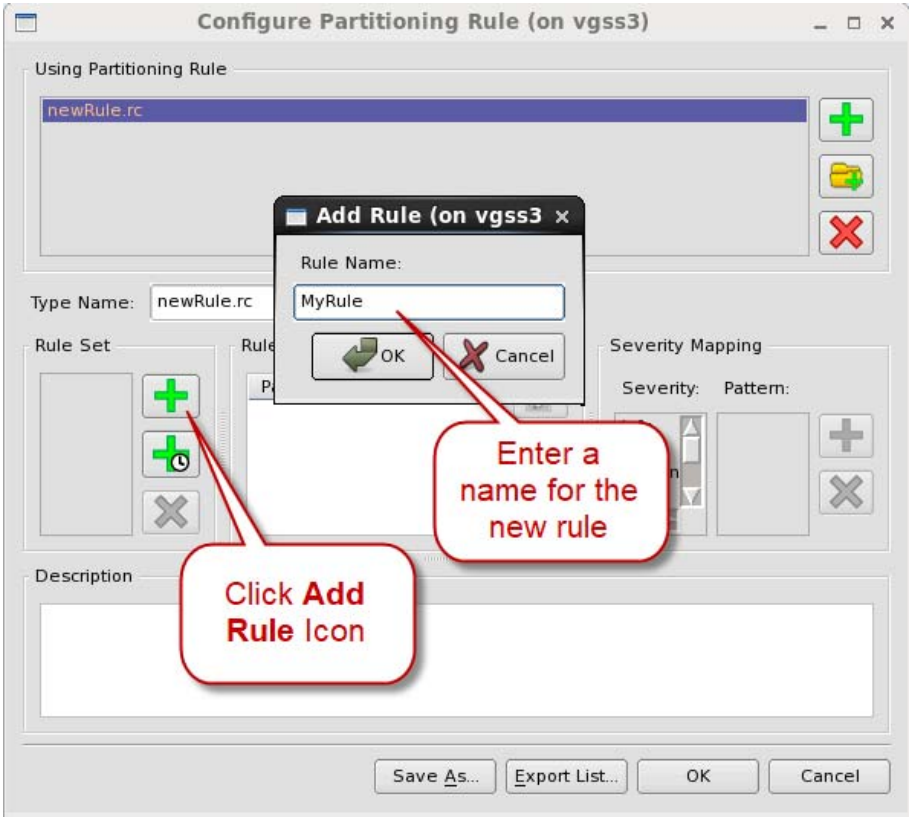
Figure: Open Log Form

3. Click the **Configure Partitioning Rule** icon present in the Smart Log toolbar.

4. In the *Configure Partitioning Rule* form, click the **Add Rule Set**  icon, enter a name for the new rule set in the **File Name(.rc)** field, and click **OK**, as illustrated in the following figure:



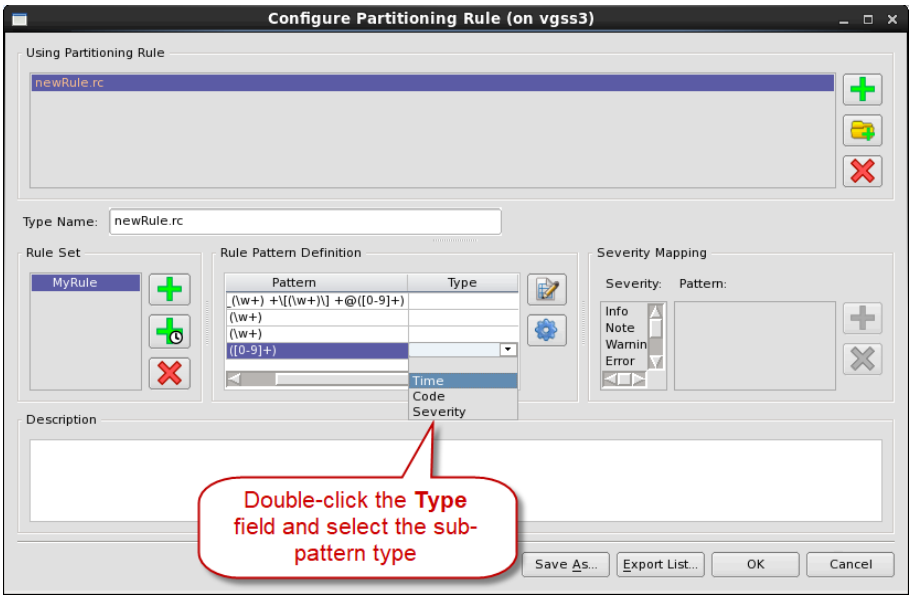
5. In the *Configure Partitioning Rule* form, click the **Add Rule** icon in the *Rule Set* section, enter a name for the new rule in the **Rule Name** field, and click **OK**, as illustrated in the following figure:



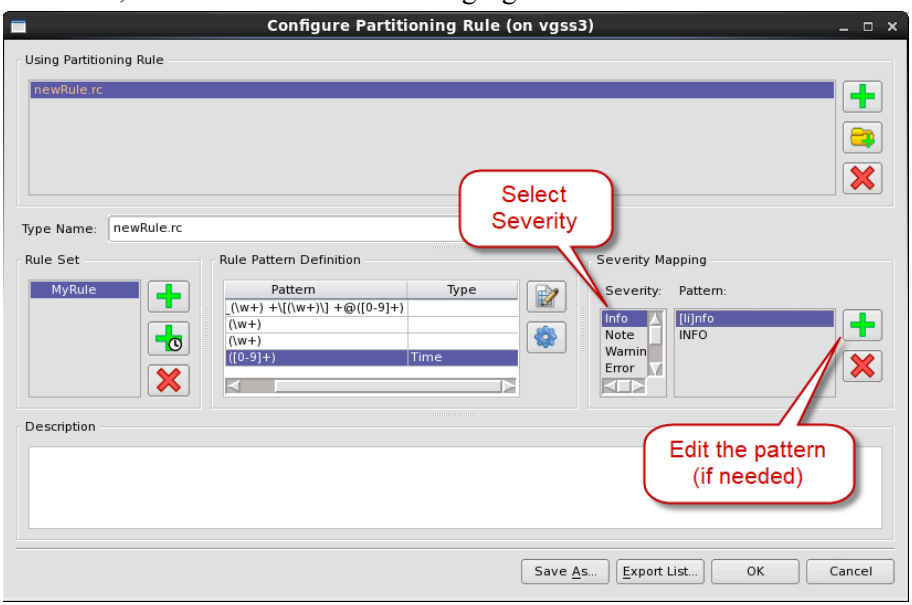
6. To edit the partitioning pattern based on the syntax of the opened log file, click the **Edit Rule Pattern** icon in the *Rule Pattern Definition* section, enter the partitioning pattern, and click **OK**, as illustrated in the following figure:

   In this example, the `MY_(\w+) +\[(\w+)\] +@([0-9]+)` pattern is used.

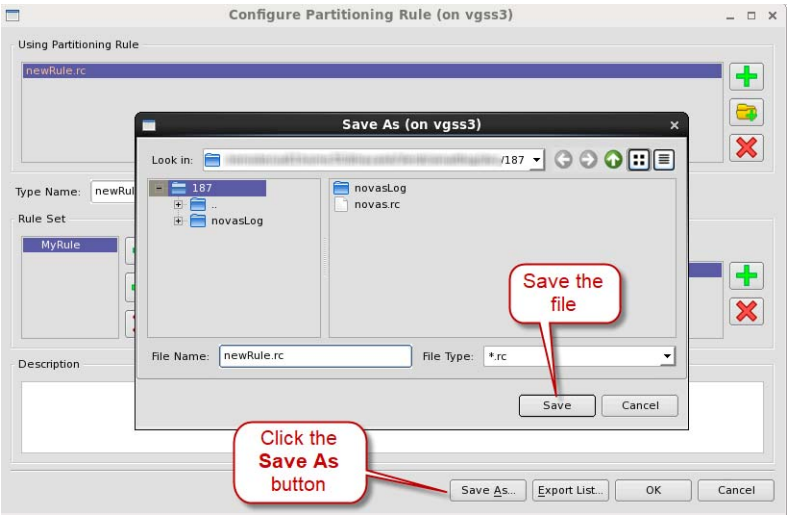
Double-click the **Type** field in the *Rule Pattern Definition* section and select the sub-pattern type, as illustrated in the following figure:



7. Select the severity and edit the pattern (if needed) in the *Severity Mapping* section, as illustrated in the following figure:



8. Click the **Save As** button in the *Configure Partition Rule* form. The *Save As* form opens up, enter the file name and click the **Save** button in the *Save As* form. Then, click on the **OK** button in the *Configure Partition Rule* form.



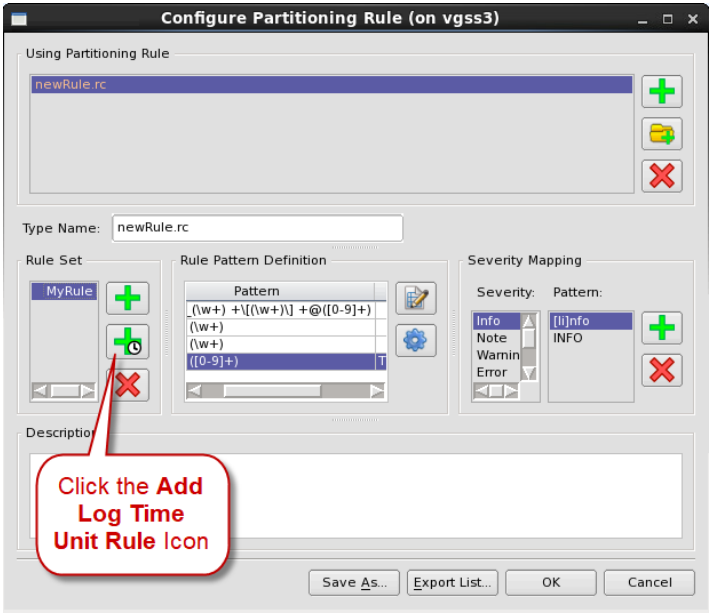
The new partitioning rule file gets saved with the specified customized settings mentioned in the preceding steps.

Creating Log Time Unit Rule

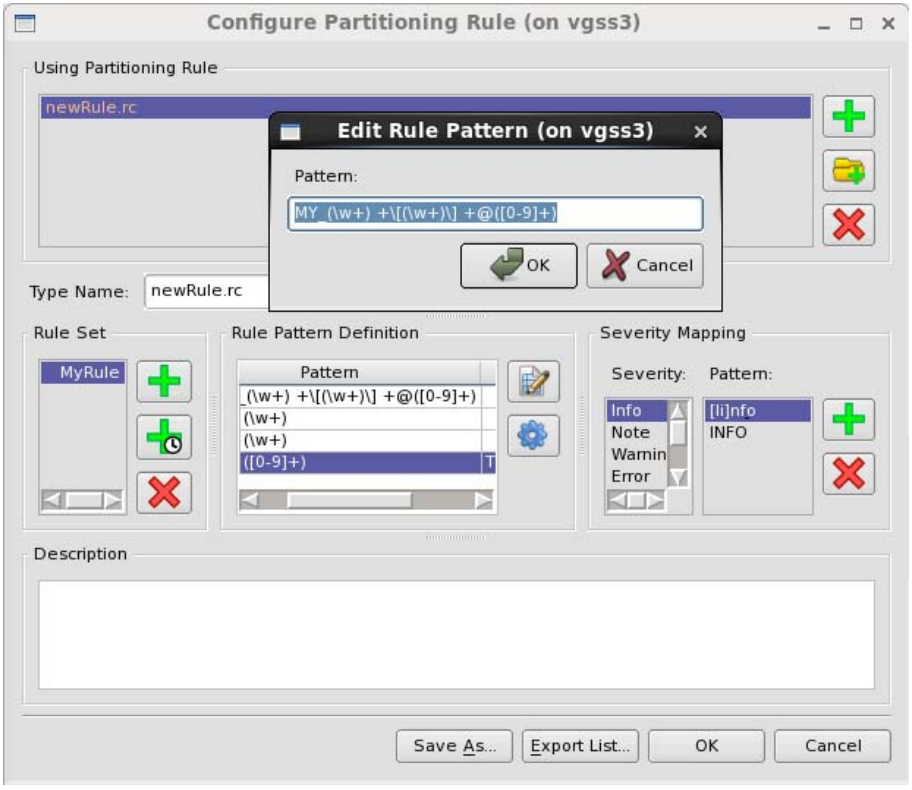
SmartLog allows you to create a log time unit rule and if the specified pattern in the rule matches a line in the opened log file, the log time unit is set automatically.

To create a log time unit rule, perform the following steps:

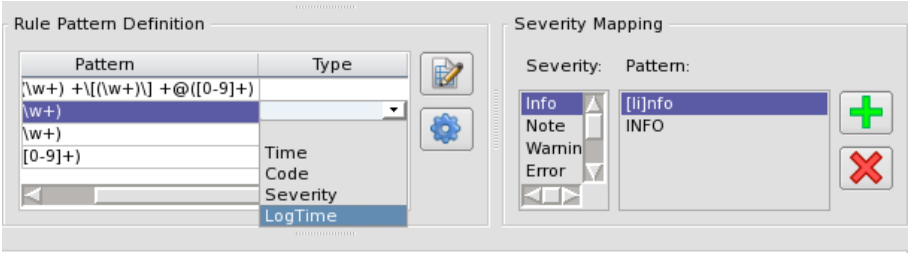
1. Click the **Configure Partitioning Rule** icon present in the Smart Log toolbar.
2. In the *Configure Partitioning Rule* form, click the partitioning rule for which you want to create a log time unit rule in the *Using Partitioning Rule* section and then click the **Add Log Time Unit Rule** icon, as illustrated in the following figure:


Configure Partitioning Rule (on vgss3)
Using Partitioning Rule
newRule.rc
Type Name:
newRule.rc
Rule Set
MyRule
Rule Pattern Definition
Pattern
_(\w+) +\[(\w+)\] +@([0-9]+)
(\w+)
(\w+)
([0-9]+)
Severity Mapping
Severity:
Pattern:
Info
Note
Warnin
Error
[Ii]nfo
INFO
Description
Click the Add Log Time Unit Rule Icon
Save As...
Export List...
OK
Cancel

3. Click the **Edit Pattern** icon in the *Rule Pattern Definition* section and enter the pattern in the *Edit Rule Pattern* form, as illustrated in the following figure:



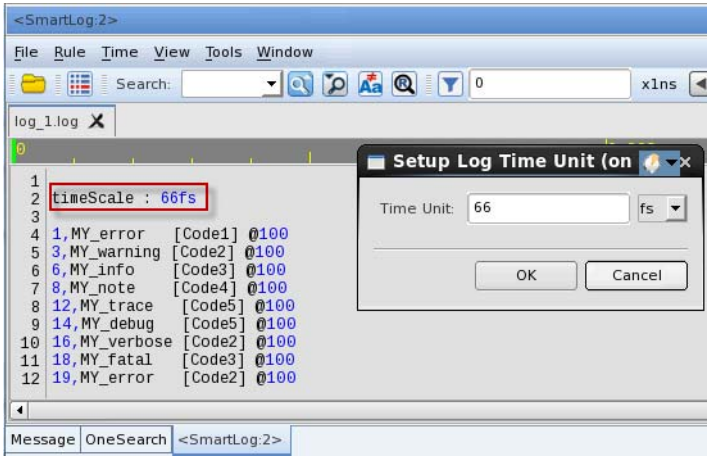
4. Double-click the *Type* column in the *Rule Pattern Definition* section and select *LogTime* for the defined pattern, as illustrated in the following figure:



5. Click the **Save As** button to save the log time unit rule.

SmartLog searches the log file for the pattern specified in the newly created log time unit rule; if any line in the log file matches the pattern, the log time unit would be set automatically.

As an example (illustrated in the previous steps), the pattern of log time unit rule is: `timeScale : (\d+(fs|ps|ns|us|ms|s|FS|PS|NS|US|MS|S)` and the Type of *LogTime* is set to `(\d+(fs|ps|ns|us|ms|s|FS|PS|NS|US|MS|S))`. As a result, the timescale in the log file gets set to 66fs automatically, as illustrated in the following figure. You can view the log time unit for the log file using the **Time** -> **Setup Log Time Unit** menu option.



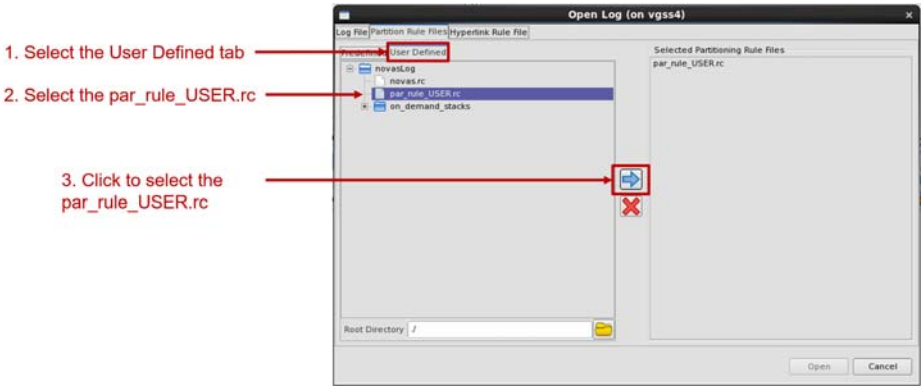
Backward Compatibility for Customized Partitioning Rule File Created Using Tcl Commands

If a customized/user-defined partitioning rule file is created using Tcl command entry, the `par_rule_USER.rc` file gets generated and saved in the `novasLog` directory.

You can open the customized/user-defined log file and apply the partitioning rule using the `par_rule_USER.rc` file by performing the following steps:

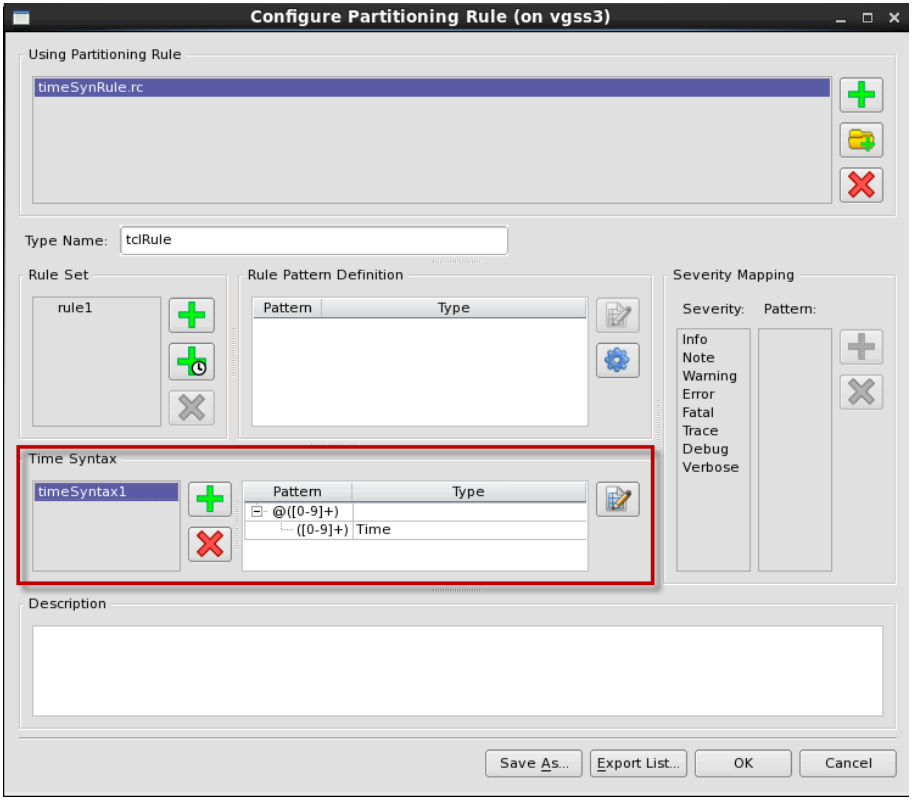
1. Open the `par_rule_user.rc` file using the icon in the *Smart Log* pane.

2. In the *Open Log* form, select the *User Defined* tab. In the *User Defined* tab, click the `par_rule_USER.rc` file and click the  icon to select the `par_rule_USER.rc` file.



3. Click the **Configure Partitioning Rule**  icon present in the Smart Log toolbar. The *Configure Partitioning Rule* form opens that displays the `par_rule_USER.rc` partitioning rule and it's rule set.

If time syntax is applied in the customized partitioning rule, the *Configure Partitioning Rule* form displays the **Time Syntax** field. This is illustrated in the following figure:



Applying Partitioning Rule

The *Structure View* in Smart Log groups and displays the log file based on the partitioning rule. If the line text in the log file matches the specified format in the partitioning rule, a block is created and the text in the following line that does not match the specified format belongs to the block.

As described in the beginning of this chapter, select one partitioning rule in the invoked *Select Partitioning Rule* form while loading a log file.

Figure: Applied Partitioning Rule illustrates an example for a UVM log file applying the UVM partitioning rule.

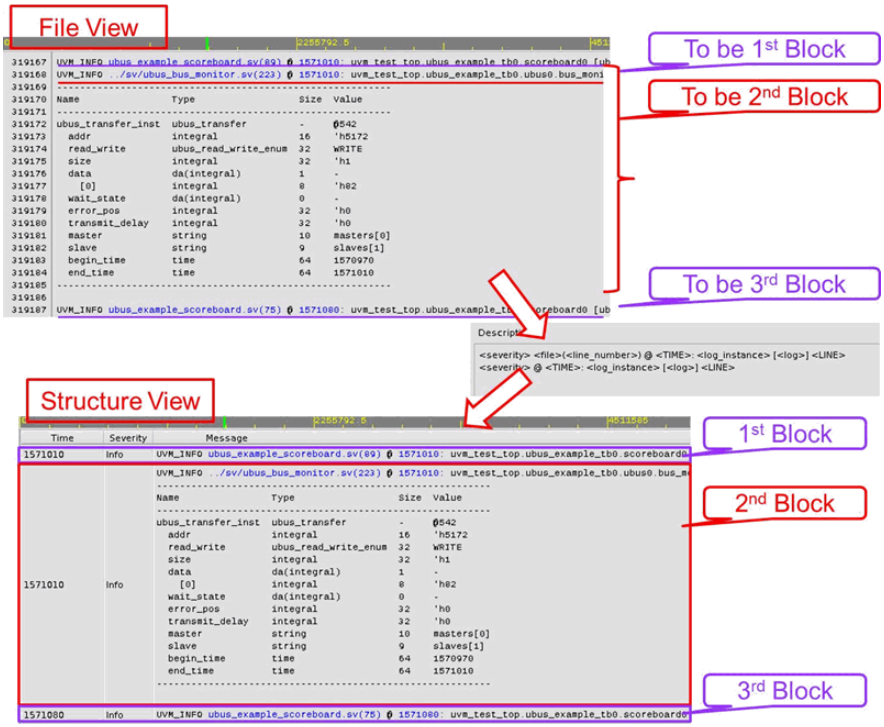
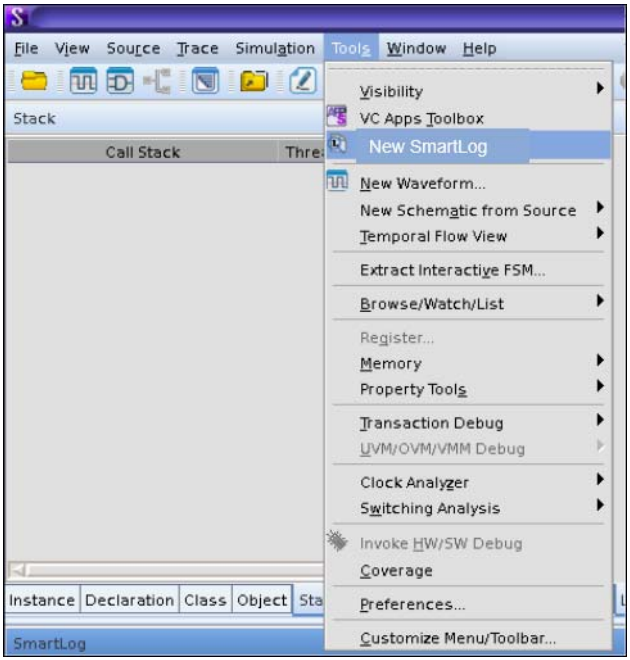


Figure: Applied Partitioning Rule

Opening Multiple Smart Log Windows and Synchronizing with nWave

You can also debug multiple logs and their corresponding waveforms at the same time by opening multiple Smart Log windows and synchronizing each log to a certain waveform.

To open a new Smart Log window, click the **Tools** -> **New SmartLog** menu option, as illustrated in the following figure:



Alternatively, you can open a new Smart Log window using the **New SmartLog**  icon present on the Verdi toolbar.

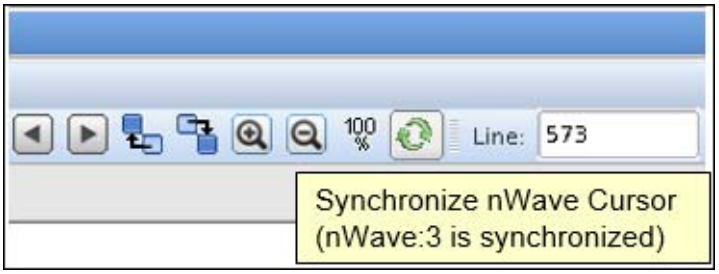
To enable or disable synchronization between multiple *Smart Log* panes and *nWave* panes, click the synchronization  icon in the *Smart Log* pane.

If multiple nWave windows are opened, the *Sync to Waveform* form opens up (as illustrated in the figure below) and you can select the nWave window to be synchronized.

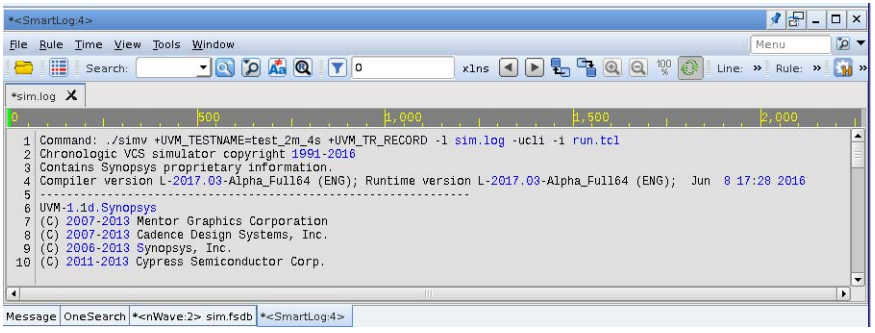
When enabling synchronization with nWave for the first time, by default, none of the synchronized waveforms is selected in the *Sync to Waveform* form; otherwise the previously specified nWave window is selected in this form.

Figure: Sync to Waveform Form

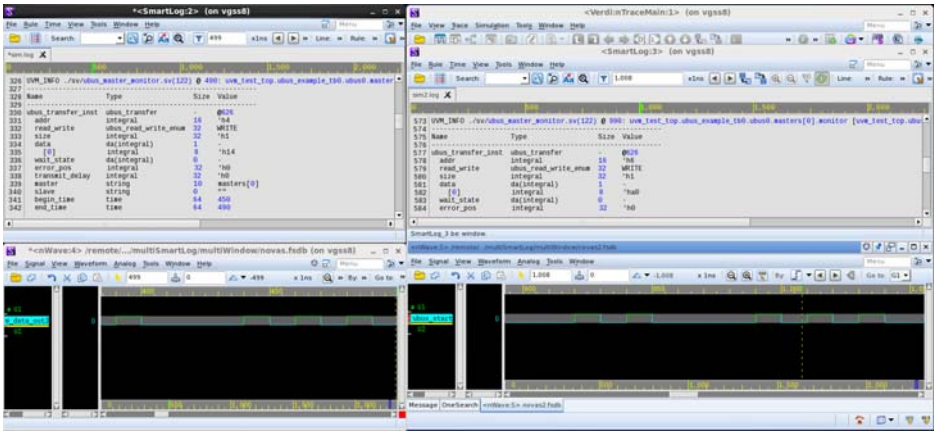
The tooltip of the synchronization icon shows the synchronization status of an already synchronized nWave window as illustrated in the following figure:



Additionally, the *SmartLog* window and the log file tab that are synced to the primary *nWave* window display the * symbol in the title, for example, `*<SmartLog:4>` and `*sim.log`.



The following figure illustrates an example of multiple Smart Log windows synchronized with multiple nWave windows:



Locating Objects

The following sections describe how to locate a specified point in the log file using Smart Log:

- *Locating a Specified Line Number*
- *Locating a Specified Time Point*
- *Synchronizing the Cursor in the nWave Pane*
- *Locating the Previous/Next Point*

Locating a Specified Line Number

The line numbers are displayed in the *File View* of Smart Log. Specify a line number in the **Line** field and press **Enter** to jump the cursor to the specified line and highlight the same, as illustrated in *Figure: Jumping to the Specified Line*.

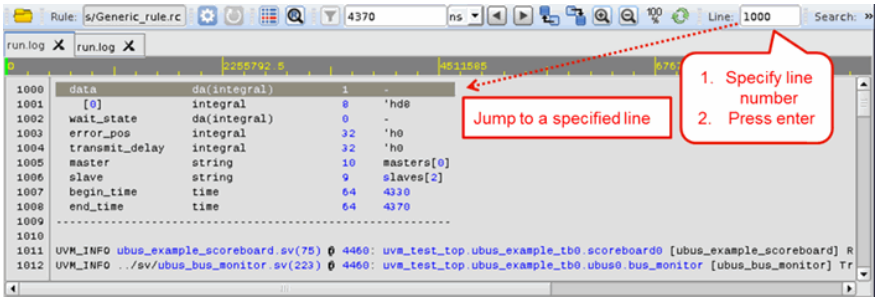


Figure: Jumping to the Specified Line

In *Structure View*, use this feature to jump to the corresponding line, however, the line number is not displayed. Specifying a line that is filtered out has no effect and no action is performed.

Locating a Specified Time Point

The **Time Cursor** field indicates the value of the time cursor in the **Time Ruler** section. You can jump the cursor to a block for which time is the closest and does not exceed the specified time using any of the following methods:

- Specify a time in the **Time Cursor** field and press **Enter**.
- Click a point within the *Time Ruler* section to move the time cursor.

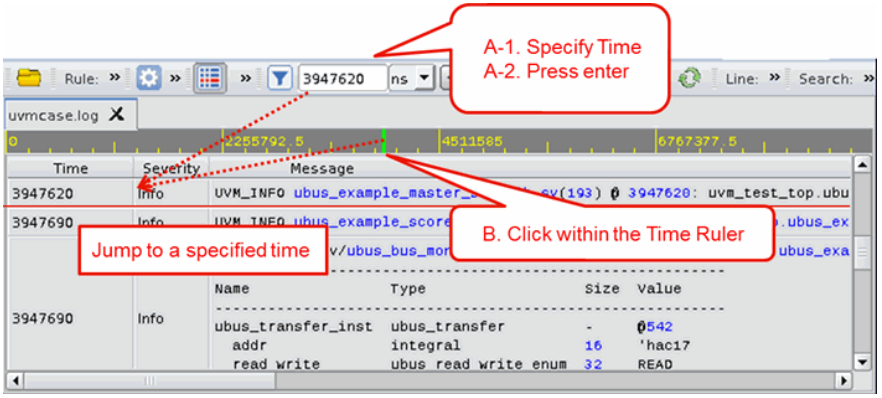
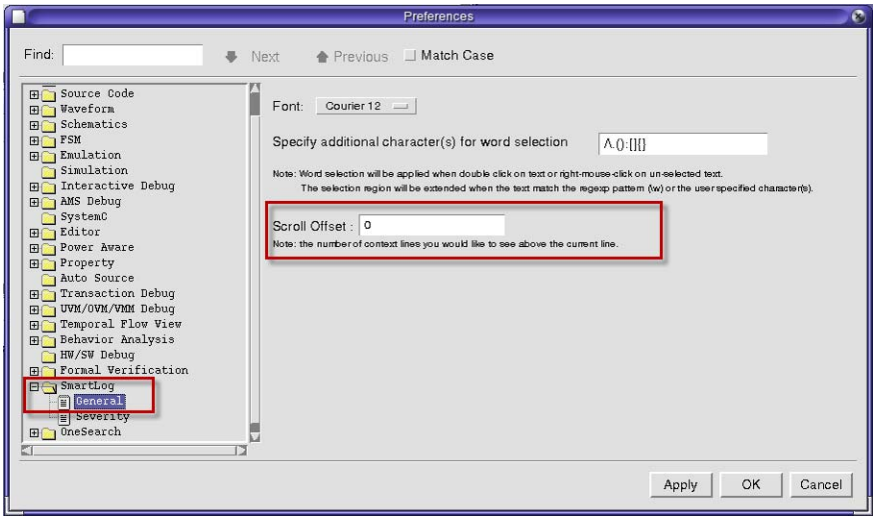


Figure: Jumping to the Specified Time

Setting the Scroll Offset

You can define or set the scroll offset as follows:

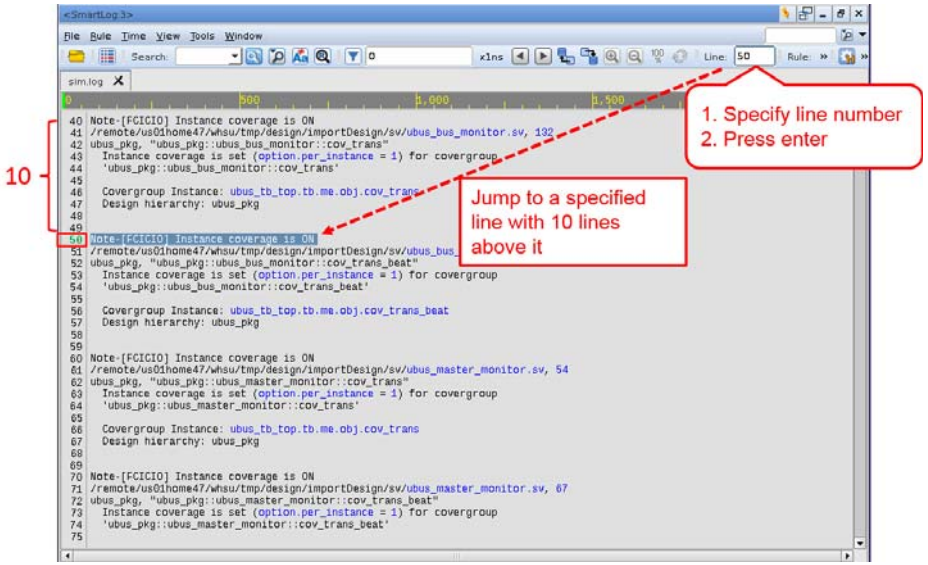
1. Click the **Tools** -> **Preferences** command to open the *Preferences* form.
2. In the **General** page in the **SmartLog** folder, specify the scroll offset in the *Scroll Offset* field, as illustrated in the following figure:



3. Click **Apply** and **OK** to save the settings.

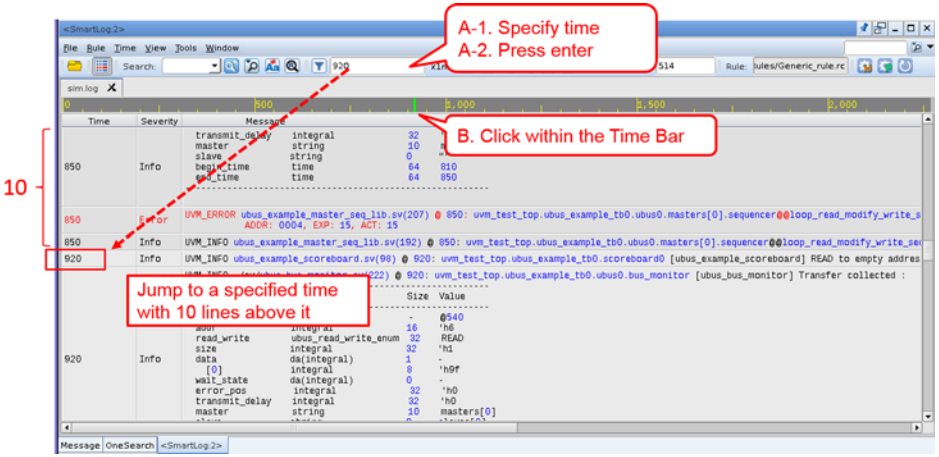
Locating the Specified Line Number With Scroll Offset

After the scroll offset is set, you can locate the line number specified in the *Line* field easily. For example, if the scroll offset is set to 10, the cursor jumps to the specified line with 10 lines above it. This is also illustrated in the following figure:



Locating the Specified Time Point With Scroll Offset

After the scroll offset is set, you can locate the time point specified in the *Time Cursor* field easily. For example, if the scroll offset is set to 10, the cursor jumps to the specified time with 10 lines above it. This is also illustrated in the following figure:



Synchronizing the Cursor in the nWave Pane

Click the  synchronization icon in the *Smart Log* pane to enable or disable synchronization between the time cursor in the *Smart Log* pane and in the *nWave* pane. When it is enabled, the *nWave* cursor is synchronized with the time specified in the **Time Cursor** field or with the time that is specified by clicking a point in the **Time Ruler** section and vice versa. This is also illustrated in *Figure: Time Cursor Synchronization*.

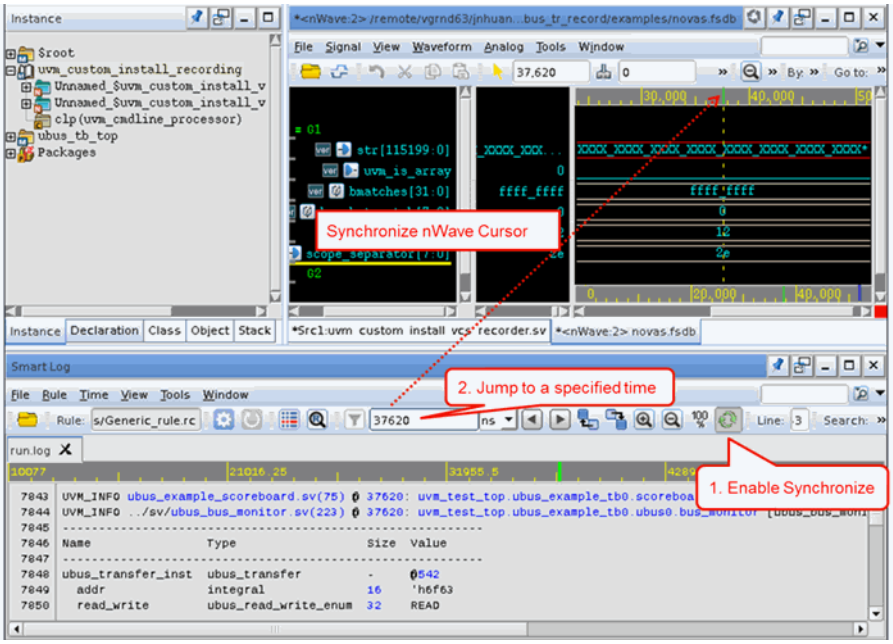


Figure: Time Cursor Synchronization

In *Structure View*, click the **Time** column to synchronize the time cursor in the *nWave* pane, as illustrated in *Figure: Time Cursor Synchronization by Clicking Time Column*.

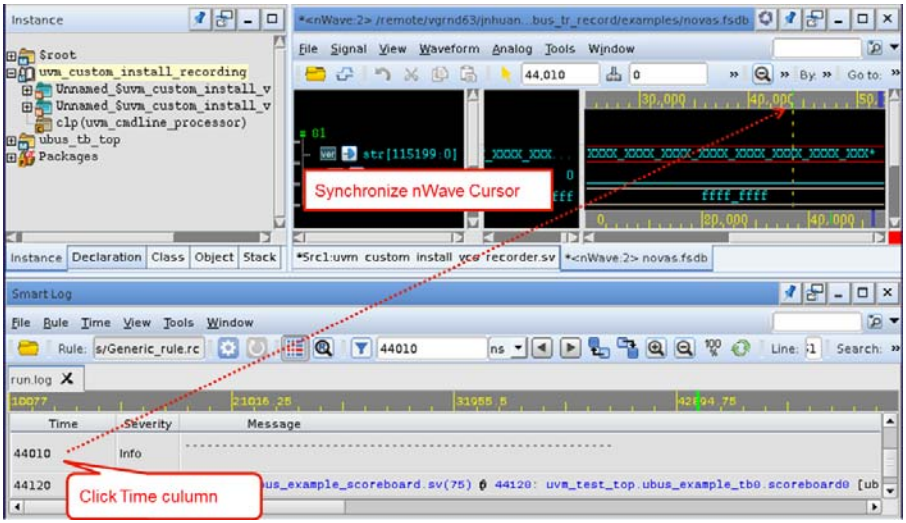


Figure: Time Cursor Synchronization by Clicking Time Column

The default time unit of the log file is *ns*. Time unit mismatches between *nWave* and *Smart Log* panes causes synchronization mismatches. Configure the log time unit by using the **Time** -> **Setup Log Time Unit** command in Smart Log to invoke the *Setup Log Time Unit* form, as illustrated in *Figure: Log Time Unit Configuration*.

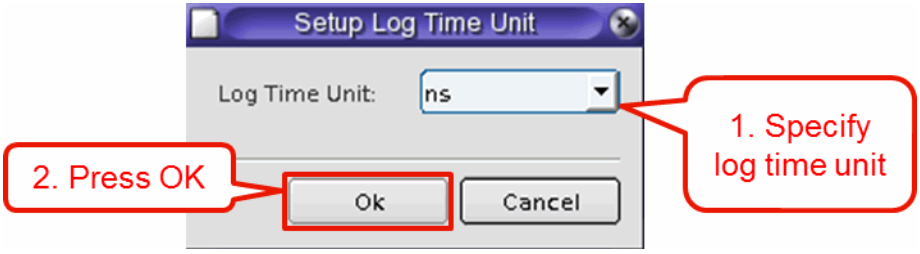


Figure: Log Time Unit Configuration

Locating the Previous/Next Point

Use the following features to jump the cursor to the previous/next time or block:

- Jump Cursor to Previous/Next time

  Different lines in the log file may have the same time information. The current time is the time of the first message (line) in the displayed content. Use  or  icons to jump to the previous or the next time for the first message in previous or next messages group with the same time.

- Jump Cursor to Previous/Next block

  Use  or  icons to jump cursor to the previous or the next blocks.

Searching, Filtering, and Reloading the Log File

The following sections describe how to search and filter out the content of the log file in Smart Log:

- *Setting the Message Filter*
- *Setting the Time Filter*
- *Searching*
- *Displaying the Search Result Section*
- *Reloading the Log File*

Setting the Message Filter

In *Structure View*, click the  icon to filter out the log file with the specified message type, message severity, and error code. These three filter categories interact with AND logic and have level order. The higher level filter impacts the number of count in the lower level filters.The level sequence is: *Message type* > *Message severity* > *Error code*. As illustrated in *Figure: Message Filter Setting*, select the specified type, severity, and error in the invoked *Message Filter* form and click the **OK** button. Smart Log displays the blocks that match the filtering criteria in *Structure View*.

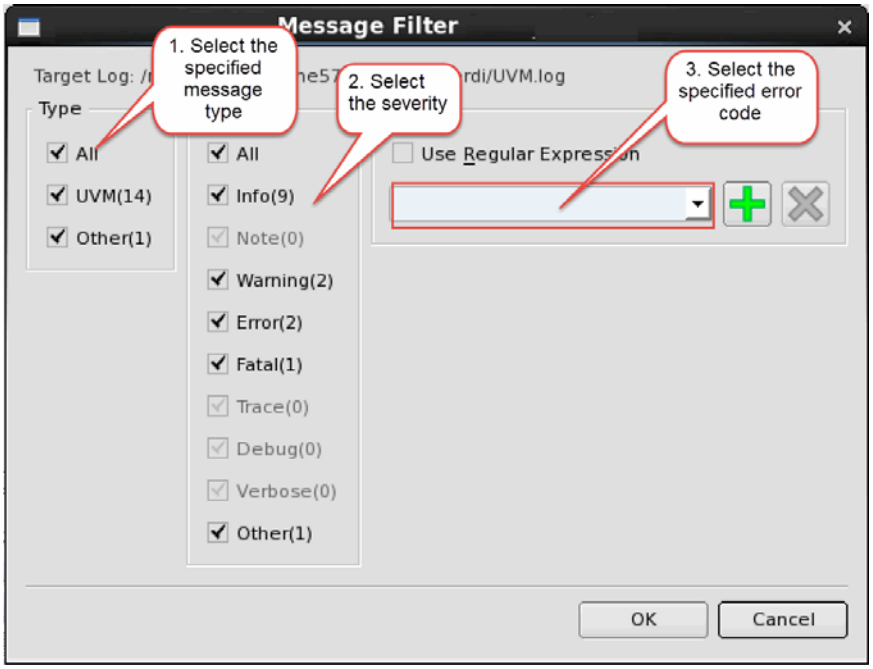


Figure: Message Filter Setting

Select the *Use Regular Expression* checkbox to filter code by regular expression patterns. For example, you can enter TE to match **TE**ST and CREA**TE**. You can also click  to create a new code filter and you can filter multiple codes simultaneously.Click  to remove codes for filtering.

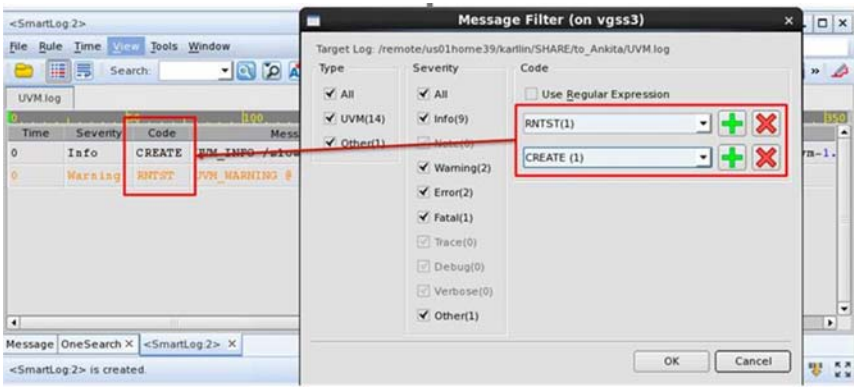


Figure: Message Filter-Multiple Codes Filter

If user specified columns are added in Smartlog, the *Message Filter* form shows the *User Specified* section, as illustrated in the figure below. You can filter the user specified columns as per the specified keywords.The number of filter objects within the User Specified section is decided by the number of user-specified columns specified by the user.

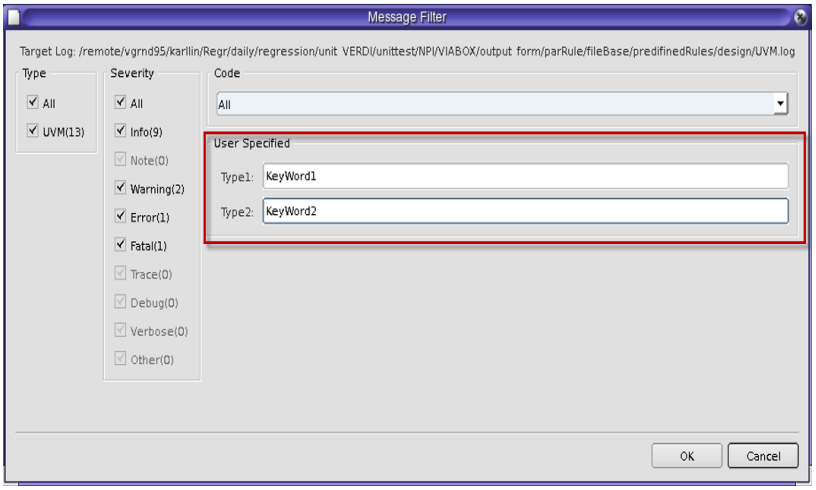


Figure: Message Filter for User Specified Columns

Setting the Time Filter

Click  ,  , or  icons to zoom in/out/all the range of the time ruler.

In *Structure View*, use the range of the time ruler to filter out a block if the time of the block is out of the range of the time ruler using the following method:

- Click within the *Time Ruler* section.
- Drag and drop the region to create a blue rectangle. The time ruler duration occupied by the blue rectangle is the new range of the time ruler.

This is also illustrated in *Figure: Time Filter Setting*.

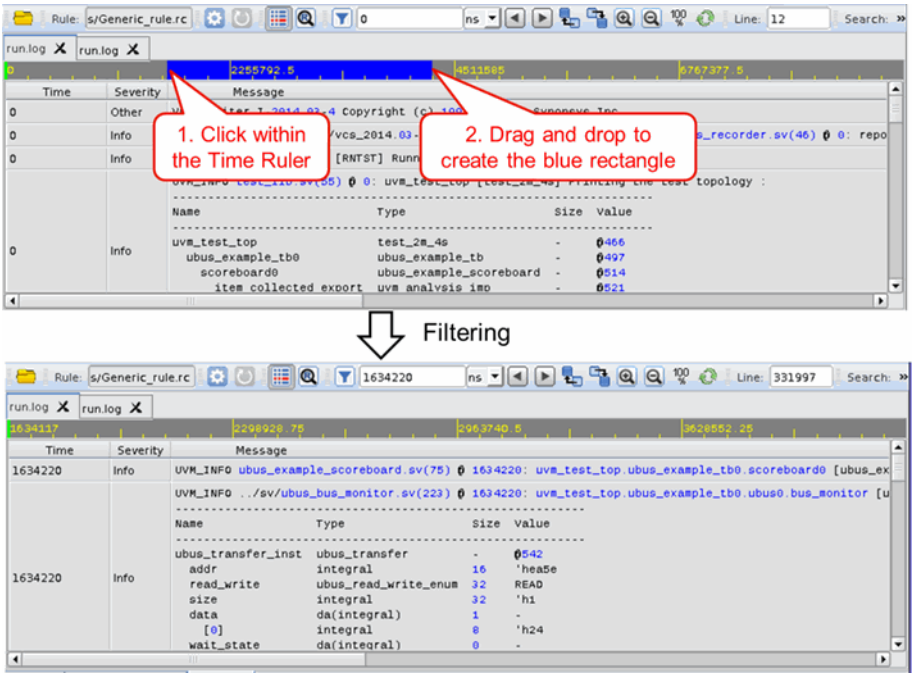


Figure: Time Filter Setting

Searching

In *Structure View*, you can search the log file to highlight a string and filter out the blocks that do not contain the matched string. Specify a string (with or without the plus character (+) prefix) in the *Search* field and click the icon or press **Enter**. Only the blocks with messages that meet the searched criteria are displayed, as illustrated in *Figure: Searching the Log File*. Continue clicking the icon or press **Enter** to search the result in another line. Click the icon to go back to the previously searched result.

In *File View*, you can search a string without filtering the other content.

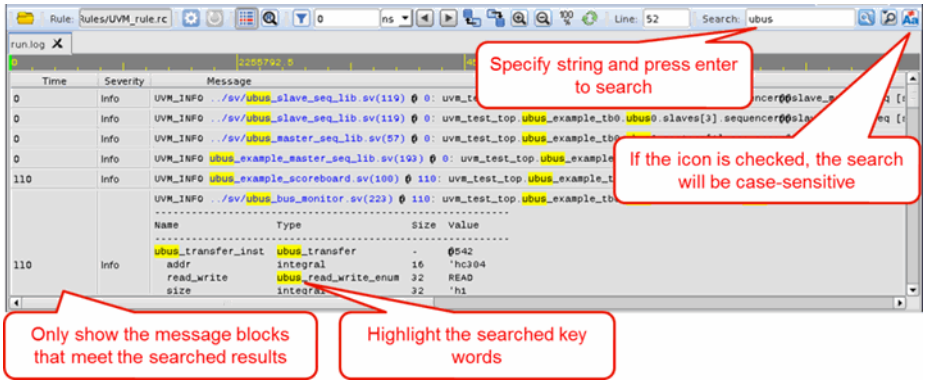


Figure: Searching the Log File

The option of searching with multiple strings is also available. The multiple strings are separated by space. To restore the original content, search an empty string.

Furthermore, to search the exception string, add the minus character (-) before the exception string, as illustrated in *Figure: Searching With an Exception String*. The string following the minus character (-) is considered as a must-not-have pattern for search.

NOTE: If the plus character (+) is the prefix of a string, the string following the + character with the whole pattern is recognized as a key word of the inclusion string. The string is considered as a must have pattern for search.
To search for the exact match of a string, enclose it in double quotes (such as, "test").

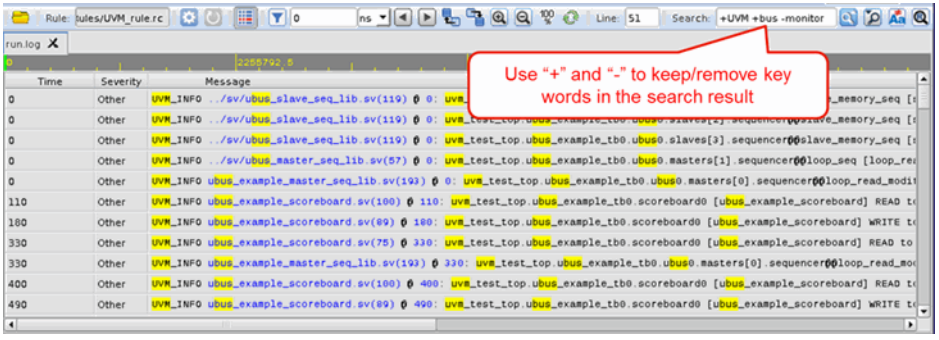


Figure: Searching With an Exception String

Example Use Cases

The following are some examples to demonstrate the search functionality:

- **Search string: +clk +module +top**

  SmartLog searches for the line that contains "clk", "module", and "top". It highlights "clk", "module", and "top" for the matched line.

- **Search string: clk module top**

  SmartLog searches for the line that contains "clk" or "module" or "top". It highlights "clk", "module" and "top" for the matched line.

- **Search string: -clk -module -top**

  SmartLog searches for the line that does not contain "clk", "module", and "top". It highlights nothing.

- **Search string: +clk +module top**

  SmartLog searches for the line that contains "clk" and "module". It highlights "clk", "module" and "top" for the matched line.

- **Search string: +clk +module -top**

  SmartLog searches for the line that contains "clk" and "module", but not "module". It highlights "clk" and "module" for the matched line.

- **Search string: clk module -top**

  SmartLog searches for the line that contains "clk" or "module", but not contain "top". It highlights "clk" and "module" for the matched line.

- **Search string: +clk module -top**

  SmartLog searches for the line that contains "clk" but not "top". It highlights "clk" and "module" for the matched line.

- **Search string: "A = 3"**

  SmartLog searches for the line that contains "A = 3". It highlights "A = 3" for the matched line.

Displaying the Search Result Section

Click the  icon to display the *Search Result* section that shows the lines containing the matched string, as illustrated in *Figure: Search Result Section*.

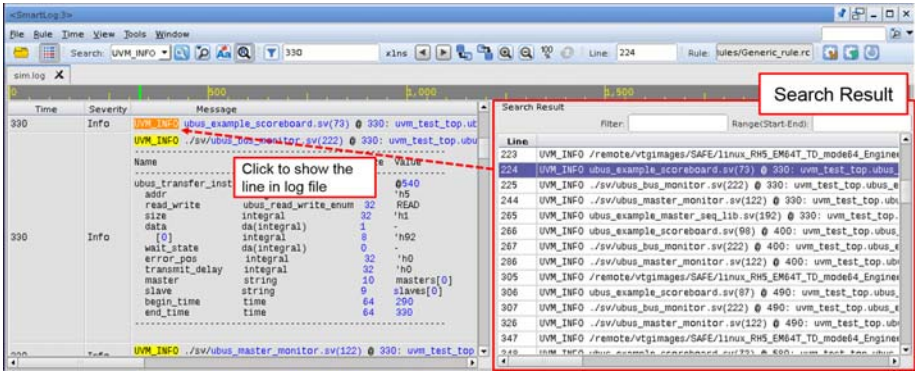


Figure: Search Result Section

In the *Search Result* section, click a row to highlight it in the log file. The search result can be further filtered by specifying a string in the **Filter** field. Set the numbers of the start and end lines in the **Range (Start-End)** field to view the search result with the specified range, as illustrated in *Figure: Filtering Search Result*.

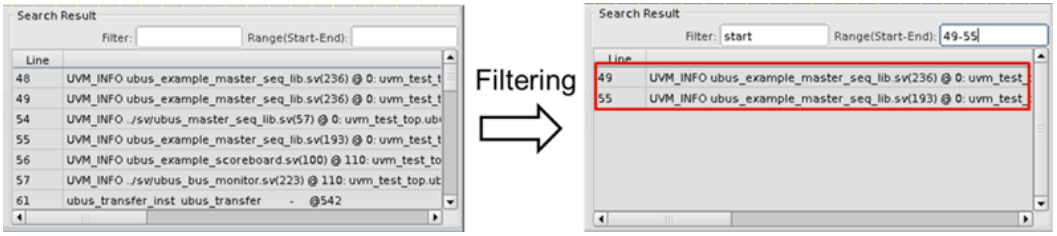


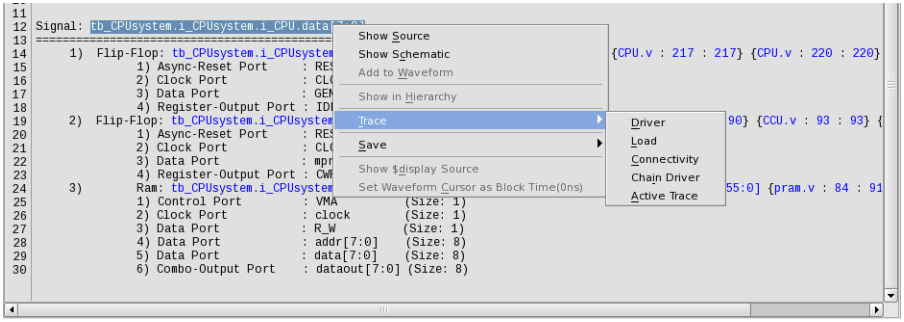
Figure: Filtering Search Result

Reloading the Log File

If a log file is changed or appended, you can reload the log file using the **Refresh Log**  icon present in the Smart Log toolbar or using the **Rule -> Refresh Log** command. The log file gets refreshed with the original hyperlink/ partitioning rule settings. Smart Log automatically refreshes the log when the hyperlink/partitioning rule changes.

Debugging in Verdi Frames

Use the *right-click command menu* to display the object in the corresponding panes for a selected signal or an instance.



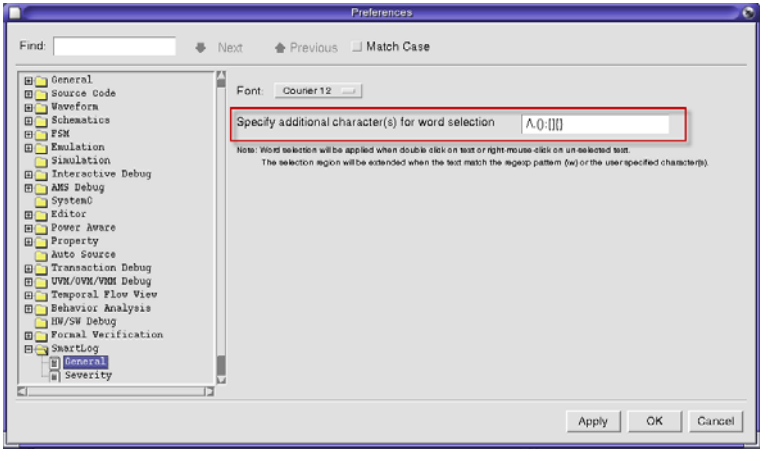
Use the following steps to debug a selected item:

1. Select the full hierarchy path of a signal or an instance string.

   Alternatively, right-click on a full hierarchy path of a signal or an instance string and the string is selected automatically.

   You can change the auto-selection rule for a string as follows:

   1. Click the **Tools** -> **Preferences** command to open the *Preferences* form.
   2. In the **General** page in the **SmartLog** folder, specify the additional characters in the **Specify additional character(s) for word selection** field, as illustrated in the following figure:



NOTE: Auto-selection rule on a word is applied when you double-click on a word/text or right-click on a word/text.
The auto-selection region is extended when the text matches the regular expression pattern (`\w`) or the character(s) specified by users in the *Preferences* form.

For example, if you right-click on the `g` character of the `Signal[0:9]` string,

- The word `Signal` gets selected, if additional characters are not specified in the *Preferences* form.
- The word `Signal[0:9]` gets selected, if additional characters specified in the *Preferences* form are `:[]`

2. Use the **Show Source** right-click menu option to show the source code in the *source code* pane.
3. Use the **Show Schematic** right-click menu option to show and highlight selected signal or instance in the *nSchema* pane.
4. Use the **Add to Waveform** right-click menu option to add selected signal or scope to the *nWave* pane.
5. Use the **Show in Hierarchy** right-click menu option to show the selected instance in the hierarchy tree of the *Verdi Instance* tab.
6. Use the following **Trace** right-click menu option to perform the tracing features for the selected signal in the Verdi platform:
   - **Driver**: Traces all possible drivers for the selected signal.
   - **Load**: Traces all possible loads for the selected signal.
   - **Connectivity**: Traces all possible drivers and loads for the selected signal.
   - **Chain Driver**: Traces through buffer/inverter chains and stops at the first non-single input driver.
   - **Active Trace**: Locates the active driver in the *source code* pane for the selected signal.
7. Use the following **Save** right-click menu options to perform the saving features:
   - **Contents**: Saves the entire content of the log file into a text file.
   - **Selected Text**: Saves the selected text of the log file into a text file.
8. When the *SML* file exists, use the **Show $display Source** right-click command to show the source code in the *source code* pane which prints the selected message.

9. Use the **Set Waveform Cursor as Block Time** right-click command in a block to synchronize the time of the block to the cursor time of the *nWave* pane.

The drag and drop feature is also available. Drag one of the following texts from Smart Log and drop it to other panes in the Verdi platform:

- A signal with full hierarchy path
- An instance with full hierarchy path
- The Top module

Using Smart Log in Interactive Debug

Smart Log is also available in *Interactive Console* in the Verdi platform and the new Interactive console is enhanced with Smart Log engine. If the VCS simulator is used to generate an SML file, you can also use it with the SML file.

NOTE: This is available starting with the Verdi J-2014.12-SP1 version.

The following sections describe how to invoke Smart Log in interactive debug in Verdi and how to use its features:

Invoking Interactive Debug

You can invoke Verdi interactive debug with `-i` and `-simType` options in the Verdi command line. For example:

```
%> verdi -sv test.sv -i -simType VCS
```

To invoke interactive debug with the VCS-SML flow, add the `-sml=verdi` simulation option. For example:

```
%> verdi -sv test.sv -i -simType VCS -simOpt
"-sml=verdi"
```

Alternatively, configure interactive simulation settings in the **Simulation** page of the *Preferences* form as follows:

1. Use the **Tools** -> **Preferences** menu command to invoke the *Preferences* form.

2. Select the **Simulation** page and enable the **Enable Smart Log** option.
3. To invoke the VCS-SML flow, select VCS as the simulator in the **Simulator** field.

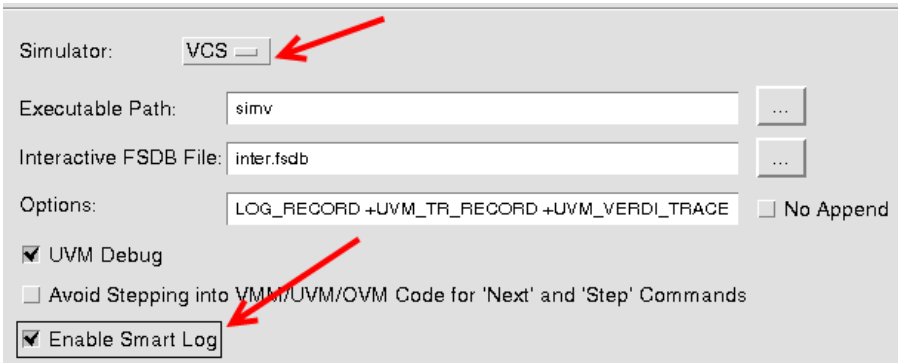


Figure: Enable Smart Log from Preferences

4. Use the **Simulation** -> **Invoke Simulator** command to enable Interactive Simulation mode.

Smart Log is invoked and the heading of the pane is *Interactive Console*. The predefined UVM hyperlink rule stored in the *UVM_i_rule.rc* file is automatically applied to create the hyperlinks.

If VCS-SML flow is enabled, the partition rule in the *Interactive Console* frame follows the rules recorded in the VCS generated SML file.

If the `-sml=verdi` option has not been specified, all the predefined partition rules are considered as partition rules.

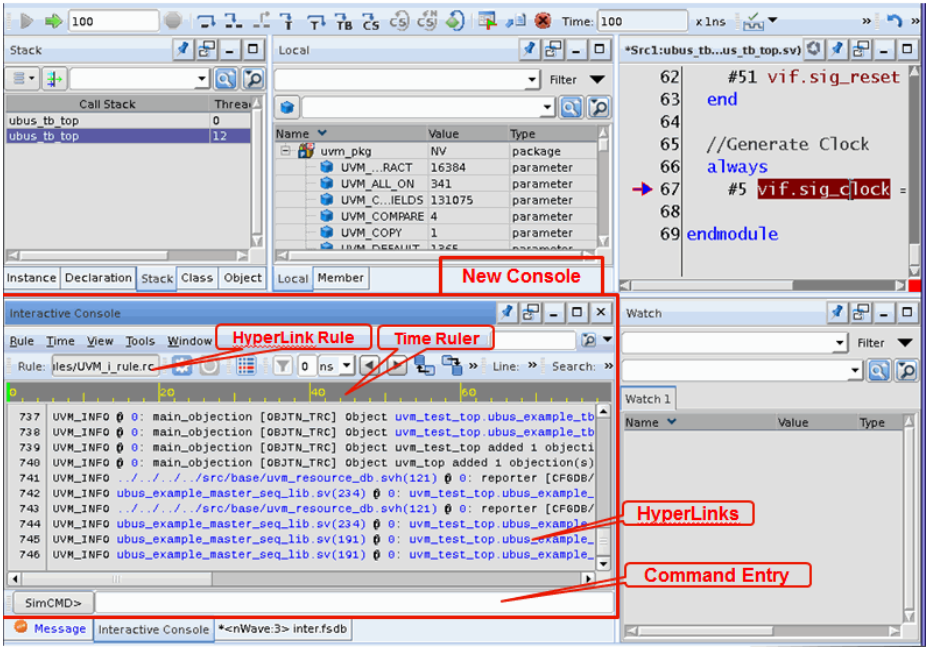


Figure: Interactive Console of Smart Log

The synchronization icon is turned *on* by default, that is, the cursor time of *nWave* and *Interactive Console* panes are synchronized by default.

You can start to debug your design and monitor the log with Smart Log in the Interactive Simulation mode.

Command Entry

At the bottom of the *Interactive Console* pane, as illustrated in *Figure: Interactive Console of Smart Log*, enter the simulation commands in the **SimCMD>** field and press **Enter** or click the **SimCMD>** button to execute the commands. The command string and returned result are displayed in the *Interactive Console* pane.

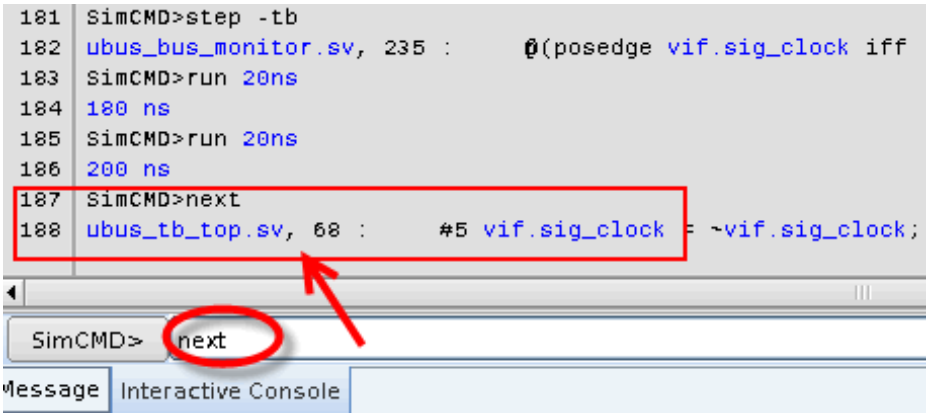


Figure: Command Entry

To get the historical typed-in commands, use up "↑" and down "↓" keys in the command entry.

Scroll Bar Behavior

When Interactive Mode is enabled, behavior of the *scroll bar* in the *File View* is as follows:

- While the simulation is running, scroll up the *scroll bar* to read previous lines.
- When simulation output is received in the *Interactive Console* pane and the *scroll bar* is not in the bottom, the output message is printed without scrolling down to the bottom.
- If a command is entered in the command entry, the command output is appended and the *scroll bar* is scrolled to the bottom.

Applying User Defined Partitioning Rules

In Interactive Mode, you can apply the user-defined partitioning rules by performing the following steps:

1. Specify a directory to save the user-defined partitioning rules under the `[VIA.parRule]` section with the `parRulePathInterForm` key. The default directory is ""; Interactive console does not use user defined partitioning rule file if the default value is "".
2. Change the directory using the *novas.rc* configuration file.

3. Interactive console uses the predefined partitioning rule files and all the rule files under the user-defined directory to perform partitioning.

Example

Predefined Rules ($Verdi_path/share/VIA/Apps/PreDefinedParRules/):

```
par_rule_UVM.rc
par_rule_OVM.rc
par_rule_VCS.rc
par_rule_LP.rc
```

User Defined Rules (/CAD/IAParRule):

```
myRule_1.rc
myRule_2.rc
```

If you do not specify *novas.rc*, Interactive console uses the predefined rules (mentioned above) and *novas.rc* (mentioned below).

novas.rc

```
...
[VIA.parRule]
parRulePathInterForm = ""
...
```

If you specify novas.rc, Interactive console uses both predefined and user defined rules as follows:

```
myRule_1.rc
myRule_2.rc
par_rule_UVM.rc
par_rule_OVM.rc
par_rule_VCS.rc
par_rule_LP.rc
```

Additionally, Interactive console uses *novas.rc* as follows:

novas.rc

```
...
[VIA.parRule]
parRulePathInterForm = "CAD/IAParRule/"
...
```

Known Issues and Limitations

The following known issues and limitations exist in the *Smart Log* feature:

- When multiple keywords are entered in the search field, if the second or later keywords are numbers, the line number of the log file is considered as a part of context and is searched. For example, if "UVM_INFO 200" key words are entered in the search field, the lines of the blocks meeting the following situations are displayed in the search result:
  - The line with the 'UMV_INFO' string
  - The line number of the line with the '200' string
- To solve performance issues caused by regular expressions, a maximum number of characters in each hyperlink line of the *Smart Log* pane should be specified. If a character number exceeds the specified value, the hyperlink in the line is not available (that is, only text is shown). The default is 4096. The following environment variable is set to modify the maximum number:

  ```
  %> setenv VGIF_MAX_LENGTH_PER_LINE 4096
  ```

OneSearch

Overview

The OneSearch capability in Verdi works like a web search engine and helps you to search through various aspects or domains within your design. The OneSearch interface is simple, intuitive, and easy to use; wherein you can enter a search query and get ranked and sorted results, view the results, modify your query, and refine your search.

OneSearch supports search in **Sources**, **Logs**, **Identifiers**, and **Docs** domains with domain-specific search engines, generates ranked matches, and stores results in domain-specific results file.

OneSearch is also aware of the current Verdi *Work Mode* (coverage/power debug and so on) and generates results based on the work mode. For example, the coverage search engine is available only in the Coverage mode.

The following figure illustrates the default view of the OneSearch pane that appears in the Verdi GUI:

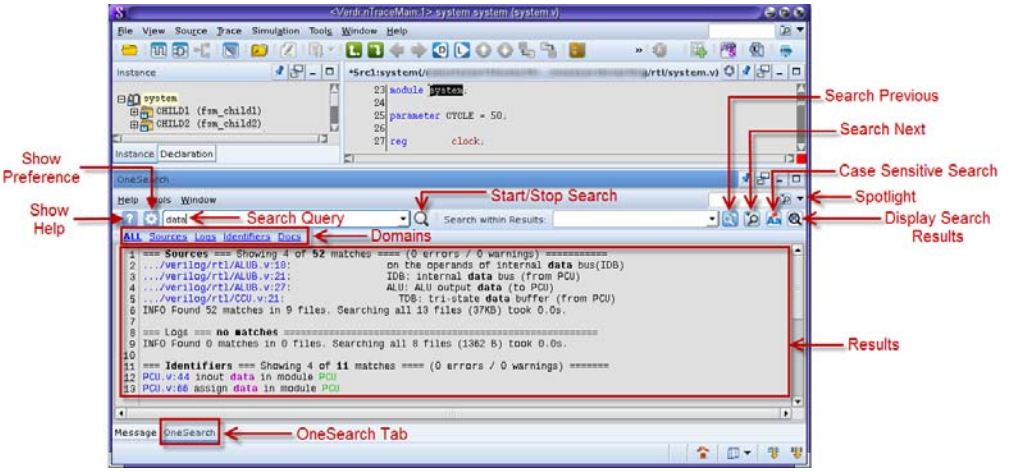


Figure: OneSearch Default View

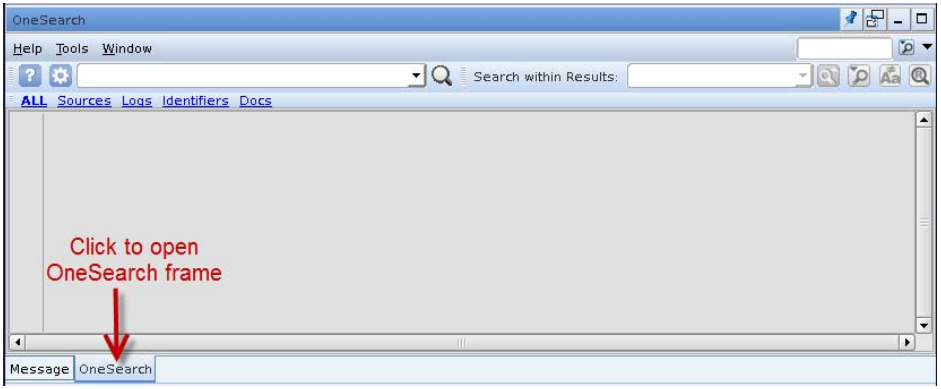
This chapter comprises of the following sections:

- GUI Use Model
- Search Domains

GUI Use Model

The **OneSearch** tab appears by default when the Verdi GUI is invoked.

To open the OneSearch pane, click the **OneSearch** tab as illustrated in the following figure:



To search for a keyword using OneSearch capability,

- Enter the desired keyword or search query (such as, "clk", "clk module Top" and "clk files:*.sv") in the left text field in the *OneSearch* pane.
- Click the icon or press the **Enter** key.

The top ranking matches for each domain are displayed in the **ALL** domain.

To view more matches and more information for each match in a specific domain, click the desired domain from **Sources**, **Logs**, **Identifiers**, or **Docs** link.

Alternatively, to search your design using the OneSearch capability,

- Click the **Spotlight** icon and select the **Verdi OneSearch** option as illustrated in the following figure.
- Enter the search query and click the **Spotlight** icon.

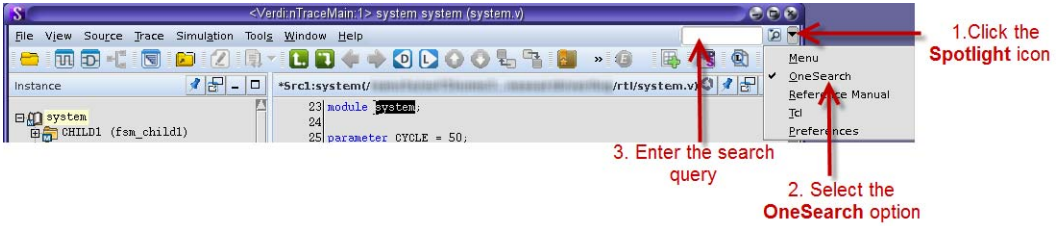


Figure: ONE Search Shortcut

Results matching the specified query are displayed in the *OneSearch* pane.

The following figure illustrates an example of the results appearing in the OneSearch pane:

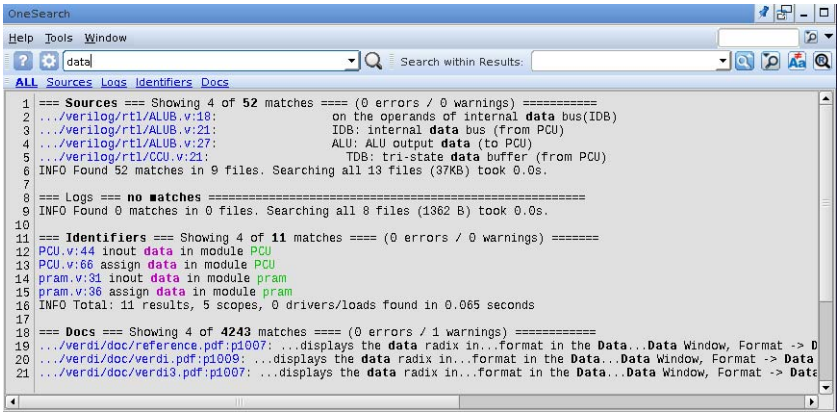


Figure: Example of OneSearch Results

NOTE: To view the help information for OneSearch, click the **Show Help** icon present in the OneSearch toolbar.
Clicking on this icon in the *Overview Mode* displays the help information in the **All** domain OneSearch results pane.
Clicking on this icon in the *Single-domain Mode* displays the help information in the selected domain OneSearch results pane.

You can apply preference settings for OneSearch using the **Show Preferences** icon. Clicking on this icon opens the *Preferences* form (*OneSearch* page). For details, see the *OneSearch* chapter in *Verdi and Siloti Command Reference Guide*.

Search Domains

The search domains for OneSearch capability are described as follows:

- **Sources**

  The **Sources** domain covers all (System)Verilog, VHDL, or other input files used in the design. The file list is extracted from the KDB (if available) or the VCS elaboration (if available) files. If none of these files is available, all files matching *\.[ch]|\.[ch]pp|\.[ch]xx|\.(cc|hh)|\.svh|\.pli|\.v|\.vrh|\.sv|\.sva|\.vhdl|\.vhd* and *Makefile|synopsys_sim\.setup|\.tcl|\.[a-z]?sh|\.upf* are searched.

  To specify an additional file list, use the `VERDI_SEARCH_SOURCES_FILE` environment variable.

  file: and *files:* are the supported keywords to filter the list of files matching the specified pattern only (such as, *files:a*.sv* matches only files starting with *a* and ending with *.sv*).

- **Logs**

  The **Logs** domain covers log files generated from compilation and simulation matching ucli\.key|.*\.rpt|.*\.log(\.gz).

  To specify an additional file list, use the `VERDI_SEARCH_LOGS_FILE` environment variable.

  file: and *files:* are supported keywords to filter the list of files matching the specified pattern only (such as, *files:sim** matches only files starting with *sim*).

- **Identifiers**

  The **Identifiers** domain covers HDL design and testbench elements of VCS simulation. Any identifier (that is, anything with a name) in the design is searched.

- **Docs**

  The **Docs** domain covers all Verdi documentation (PDF) files.

  To specify an additional file list, use the `VERDI_SEARCH_DOCS_FILE` environment variable.

  file: and *files:* are supported keywords to filter the list of files matching the specified pattern only (such as, *files:*plan** matches only files containing *plan*).

Search Modes

OneSearch supports the following search modes that decide where to search and how to display the search results:

- Overview Mode
- Single Domain Mode

Overview Mode

This is the default mode for OneSearch.

This mode is intended as the first step in a search and provides a quick overview of the results in the entire design. In this mode, scrolling in the OneSearch results window moves entire domain groups but does not move matches within a group. When you refine (modify) the search query, such as, by adding another pattern to the query, and click the search button again, then a new search over all the domains is started.

Every group ends with a summary line that indicates the total number of results (not just the visible count). It also contains an indicator for ongoing searches.

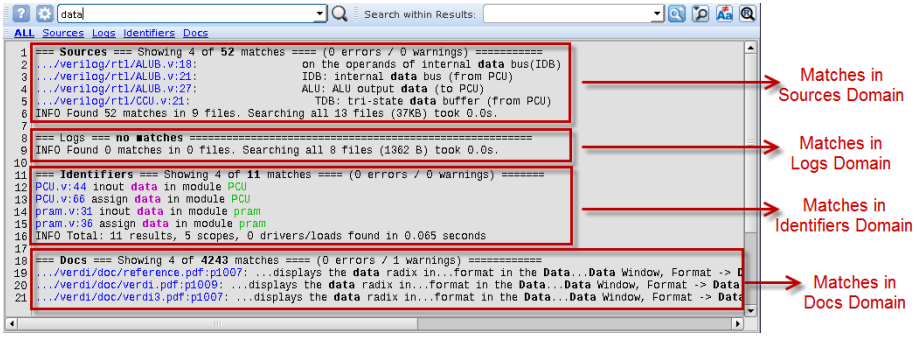


Figure: Overview Mode

Single Domain Mode

This mode is intended as the second step to view more matched results from a single domain. To switch from Overview mode to Single-domain mode, click on any of the domain links: **Sources**, **Logs**, **Identifiers**, or **Docs**.

On clicking on a domain name, OneSearch does not trigger a new search but displays the matches from the previous search for the selected single domain. All matches from the selected domain are displayed.

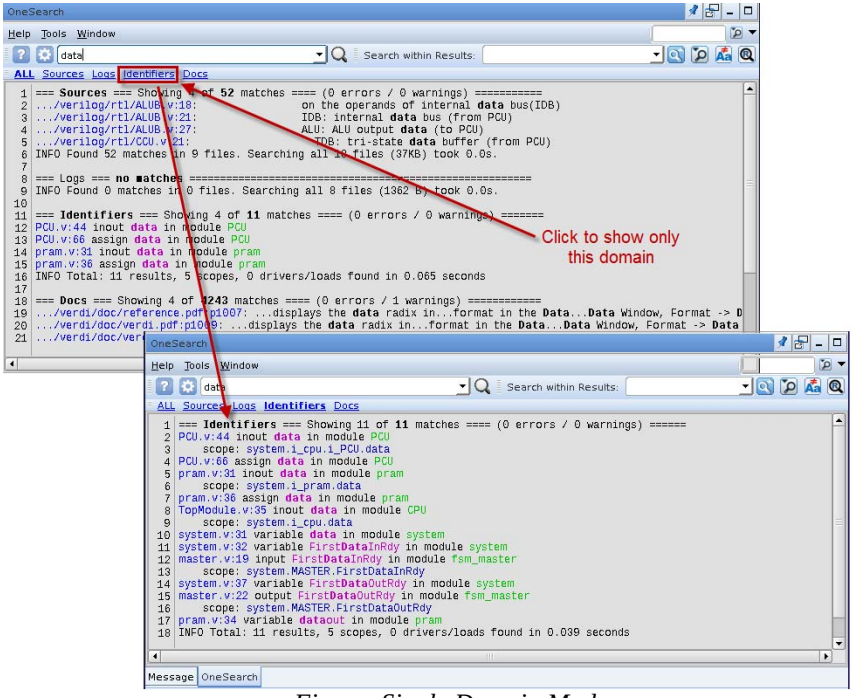


Figure: Single Domain Mode

Support for Multiple Line Results

OneSearch ranks the matches for documentation not only within a single line, but also combines the matches for multiple lines based on the distance between the matching words; therefore, multiple matches within a paragraph are assigned a higher ranking. A configurable upper limit ensures that these words are not too far from each other.

Overview Mode

In Overview mode, Verdi only displays the first line of the search result, which contains a brief summary of all matches.

The following code snippet illustrates an example output in overview mode for the search term: *loading*.

```
=== Docs === Showing 3 of 278 matches ==== (0 errors / 0 warnings) =====
/verdi/doc/CoverageTut.pdf:p4 (text): when loading succeeds …the loading
times…when loading fails …

/verdi/doc/CoverageTut.pdf:p1 (text): After loading an exclusion… after
loading related...

/verdi/doc/FsdbReader.pdf:p118 (text): and loading fails. It is therefore
highly recommended
INFO Found 278 matches in 24 files. Searching all 24 files (12MB) took
0.7s.
```

A four line, a two line, and a single line result is illustrated in the preceding snippet.

- Single-line results are displayed in the same manner as in previous releases (see result 3).
- Multi-line results include a brief summary of the matches to fit a maximum of 3 highlighted matches into the line. This limit is imposed by onegrep due to space limitations (see result 1).

The following code snippet illustrates an example output in overview mode for multiple search terms:

```
=== Docs === Showing 4 of 518 matches ==== (0 errors / 0 warnings) ======
/verdi/doc/CoverageTut.pdf:p36 (text): reviewed after the reset… property
of this module…

/verdi/doc/CoverageTut.pdf:p96 (text): After resetting an exclusion file…
The property of the system to…

/verdi/doc/FsdbReader.pdf:p18 (text): when the reset property of… when
reset succeeds… another property is…

/verdi/doc/FsdbReaderSummary.pdf:p1 (text): A property of an FSDB…
incomplete resetting of …
INFO Found 518 matches in 24 files. Searching all 24 files (12MB) took
0.7s.
```

Single-Domain Mode

In Single-domain mode, Verdi displays the complete details of the search result.

The following code snippet illustrates an example output in single-domain mode for the search term: loading.

```
=== Docs === Showing 278 of 278 matches ==== (0 errors / 0 warnings) ====
/verdi/doc/CoverageTut.pdf:p4  (text): when loading succeeds …the loading
                                       times…when loading fails …
                                       handled otherwise? The main idea
                                       is that when loading succeeds,
                                       the system has to take to the
                                       loading times into account.
                                       Otherwise,
                                       loading is deferred until the
                                       system has decided based on this
                                       property. In general, loading of
                                       these data structures is based on

/verdi/doc/CoverageTut.pdf:p1 (text): After loading an exclusion… after
                                      loading related...by the system,
                                      which is done automatically. After
                                      loading an exclusion file and after
                                      loading related data (see section
                                      5.3.1), it

/verdi/doc/FsdbReader.pdf:p118 (text): and loading fails. It is therefore
                                       highly recommended
[the other 275 matches are truncated in this figure]
```

Support for Synonyms

OneSearch automatically searches synonyms for terms in search queries based on a predefined synonym database. This functionality provides you the capability to search results for words with the same (or very similar) meaning.

NOTE: This feature is applicable to **Sources**, **Logs**, and **Docs** domains.

If the search query entered by you contains terms for which synonym(s) are defined in *OneSearch*, an additional message appears in the *Results* pane indicating that the search results also include synonym(s)' search results. If you do not want to view the search results for the indicated synonym(s), you need to enclose the search term in double quotes (""). An example of the message is illustrated in the following figure:

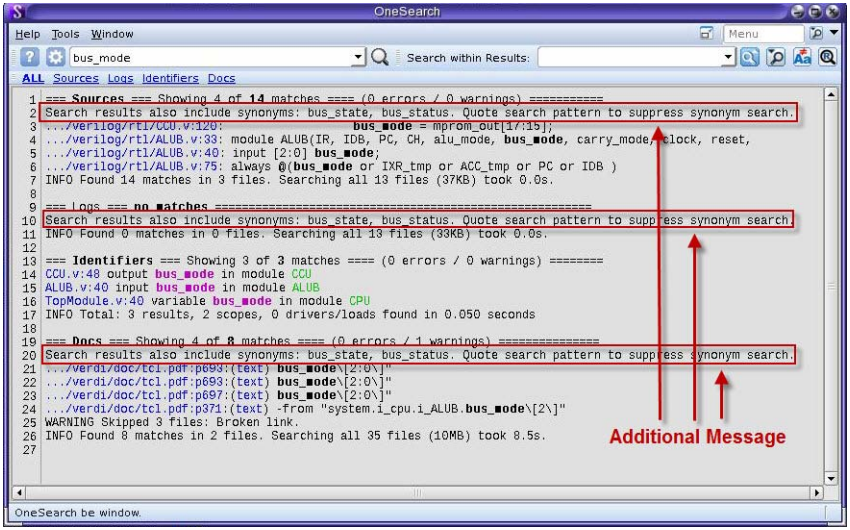


Figure: Example for OneSearch Query

NOTE: Quoted search terms as well as keywords and their arguments are not subject to synonym search.
OneSearch ranks exact matches/results to the search query higher than the search query's synonyms' results.

Example Use Case

If the search query is *start_transaction*, *OneSearch* looks for the synonyms of *start* and *transaction*. If *start* yields the synonym *begin* and there is no synonym defined for *transaction*, then *OneSearch* shows results for two query terms: *start_transaction* and *begin_transaction*.

If additional synonyms for *transaction* are also found, these are searched in the same manner.

Specifying an Additional Synonym Database

You can specify an additional synonym search database file (text file) using the `VERDI_SEARCH_SYNONYM_FILE` environment variable. The set of synonyms listed in this text file are added to the original pre-defined synonyms database, however, the predefined synonyms cannot be removed.

The text file must contain a set of synonyms (comma-separated list of words) in every line. All words in a line can be replaced by each other.

For example, assume that the synonym database contains the following three lines, which signifies that *A*, *B*, and *C* are synonyms to each other; *B* and *D* can be used interchangeably; and *D*, *E*, and *F* can also be used interchangeably.

A,B,C

B,D

D,E,F

If the search query contains the search term *B*, the synonym search checks the database line by line for the occurrence of the search term *B* and includes all synonyms into the search query. Thus, the search for *B* includes the synonyms: *A*, *C*, and *D*, but not *E* and *F*.

Limitations

The following limitations apply to this feature:

- Synonym search only applies to alphanumeric words. Special characters, such as `-`, `_`, `!`, `@`, `#`, `$` are not supported.
- Stemming of search terms is not supported, that is, inflection of nouns, pronouns, adjectives, and articles is not recognized. Derived forms of verbs are also not recognized.
- Words and sub-words are only subject to synonym search, if they are completely capitalized, or uncapitalized, or only the first letter is capitalized. For example, the following words (and their sub-words) are not subject to synonym search: *BeGIn*, *AMbA_tRANSMISSION*, *pOpULate*, *mY_OwN_USBRegister*.
- Using this feature may impact OneSearch performance.

Command Line Use Model

Besides the integration of OneSearch into the Verdi GUI, the `onesearch` command line option is also available, which can be executed in the design root directory.

The `onesearch` command line option supports the following options:

The options listed below are optional.

Option	Description
--help	Prints brief help messages explaining all the search domains.
--help <*domain*>:	Prints the detailed help messages for the specified search domain.
--maxresults [<*matches*>][:<*lines*>]	Specifies the maximum number of matches and maximum number of lines per match (in case of multi-line matches). 0 means no limit. Settings apply to both overview (four matches, only one line per match by default) and an individual domain (3000 matches, no line limit per match).
--plain	Creates pure ASCII output without color and/or bold format.
--regexp	Specifies the regular expression style for wildcards. The default is glob style (such as, "*.sv").
--rootdir <*dir*>	Specifies the root directory to search. The default is the current working directory.

The option listed below is mandatory

Option	Explanation
[--] <*query*>	Specifies the words and keywords to query. Use double quotes to enclose multiple words (such as, "as is").

Usage Examples

The following are some of the examples for search capability:

- To search for words: *xbus_transfer* and *addr* in all **Sources**, **Logs**, **Identifiers**, and **Docs** domains and report top ranking matches for each domain, enter the following command:

  ```
  onesearch xbus_transfer addr
  ```

- To search for words: *xbus_transfer* and *addr* in all files matching **.svh* by using all **Sources**, **Logs**, **Identifiers**, and **Docs** domains and report top ranking matches for each domain, enter the following command:

  ```
  onesearch xbus_transfer addr "files:*.svh"
  ```

- To search for words: *xbus_transfer* and *addr* only in the **Identifier** search domain and report top ranking matches, enter the following command:

  ```
  onesearch xbus_transfer addr Identifiers:
  ```

Temporal Flow View Tutorial

Overview

To debug a design, you need to understand the structure and the behavior of the design. Understanding the structure allows you to visualize the connection between blocks or signals. Also, understanding the behavior allows you to know the relationship between driver-signals, loader-signals, driver triggers, and the value transition. Visualizing the structure and behavior of the design is very useful and convenient to become familiar with the design and then debug it.

The flow views are unique temporal views and analysis tools that allow the visualization and analysis of the design's behavior through time. The *Temporal Flow View* identifies and displays causal control and data paths - the registers and signals that actually caused the erroneous value to occur - through multi-level combinational logic within one or more register-to-register transfer stages. The mechanism helps to quickly find the bug without repeatedly looking at the driver or fan-in signals of the signal of interest and the intelligently filtered temporal representation allows you to locate and identify problems in the most efficient manner possible without going through the different source code, schematic and waveform windows. However, the flow views also can be used to drive the source code, *nWave* (waveforms), *nState* (state machines) and *nSchema* (schematics) frames as needed.

The display in the *Temporal Flow View* frame is similar to the following example:

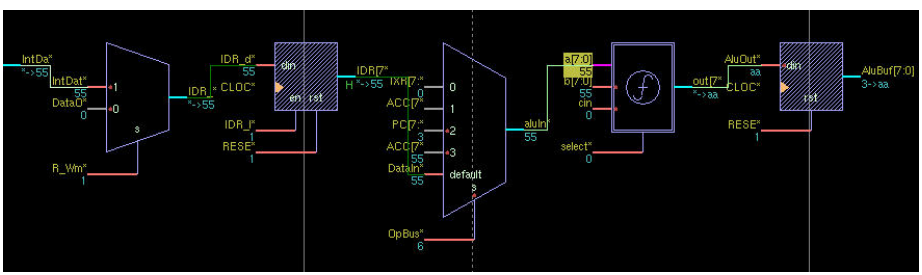


Figure: Example Temporal Flow View Frame With Expanded Statements

In the *Temporal Flow View* frame notice the following items:

- The input data signals are on the left and output data signals are on the right of the symbol.

- On top of the gray vertical line, the number shown indicates that the clock active edge for that signal is at that time in *ns*. If you click the symbol associated with the register, the number appears in the box of the toolbar, That is, the value of the register changed at that time.
- The input control signals are on the bottom of the symbol. (An example of a control path is a multiplexer's select input, and an example of a data path is a multiplexer's data signals.)
- Some input ports are pink (signal is actively contributing to the output value) or gray (signal is not actively contributing). (Turn on the **View -> Signal -> Active Nodes Only** toggle command to see all nodes or just active ones.)
- The red dot associated with some ports indicates that these fan-in registers have value transition in their previous clock edges.
- The clock that drives the located register can be shown without tracing through combinational logic cones and correlating the clocks manually.
- The annotated simulations values (blue numbers).
- The signal names.
- The bus contention and active fan-in information can be clearly viewed in a single view.
- The cause of a specific data pattern on a partial bus can also be isolated.

The *Temporal Flow View* traces to the root cause automatically with two major mechanisms. That is, behavior analysis engine and automatic cause and effect tracing.

Behavior Analysis Engine

From the KDB and FSDB, Verdi builds an internal model of actual design behaviors using the synthesis and formal technology. Traditionally, the tracing stops at the problematic signals when debugging. If further tracing is required, then you must invoke and configure the tracing manually. Now, with the behavior analysis engine, the *Temporal Flow View* traces based on the behavior of the design automatically.

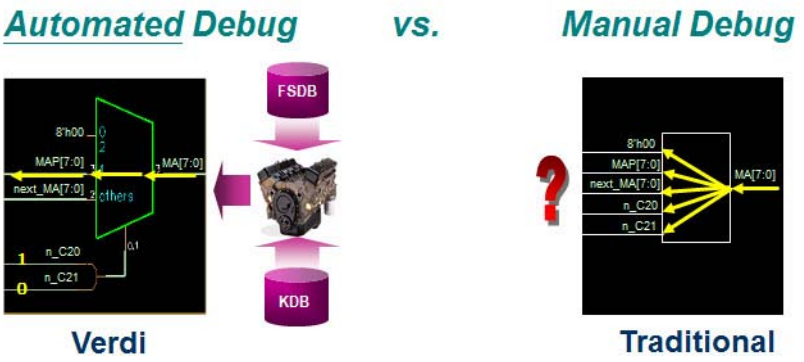


Figure: Temporal Flow View Traces Automatically to Debug

The following tracing features are available:

- Unrolls the function over time.
- Differentiates the data from control signals.
- Determines the clocking.
- Prunes inactive elements.

Automatic Cause and Effect Tracing

The *Temporal Flow View* traces not only to the problematic signal but to the root cause of the error. The *Temporal Flow View* provides a complete environment that helps in visualizing the behaviors with time and structure easily. The root cause and the trace path can be viewed holistically.

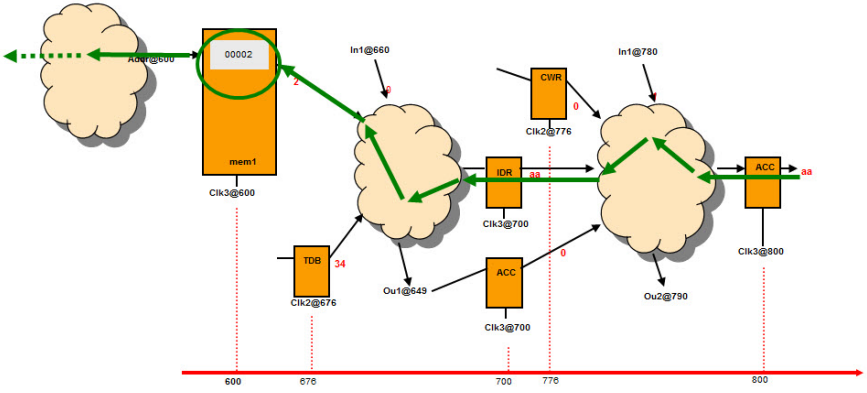


Figure: Temporal Flow View Traces to Root Cause of Error

The following scenarios show some of the best situations to benefit from advantages of root cause tracing of the *Temporal Flow View*:

- Understand the design behaviors and locate the root cause of a wrong value.

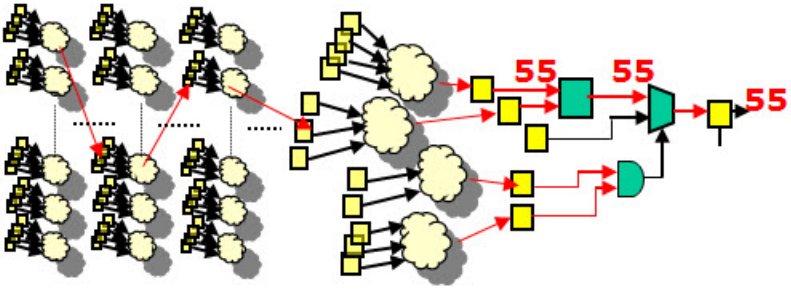


Figure: Locate Root Cause of Wrong Value

- Trace the memory content that is not in FSDB and locate the memory write.

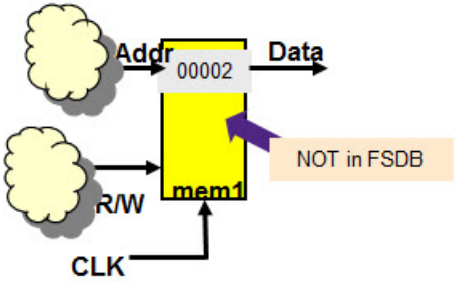


Figure: Trace Memory Content That is not in FSDB

- Locate the root cause of unknown (X) values.

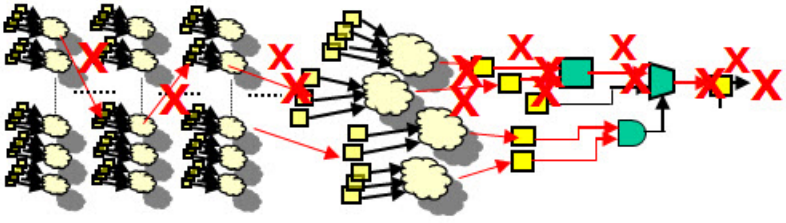


Figure: Locate Root Cause of Unknown Values

- Locate the cause of different results between two simulation runs.

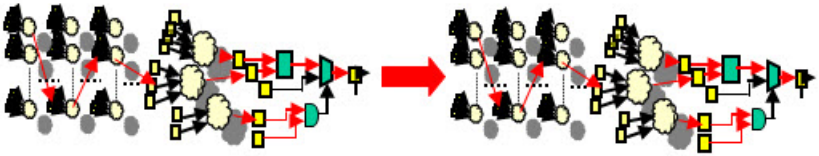


Figure: Locate Root Cause of Different Results Between Two Simulation Runs

Trace to Root Cause Automatically

From nTrace

In *nTrace*, the root cause of an issue can be traced automatically with one of these commands: **Trace This Value** or **Trace X**. Select a signal in the *nTrace* source code pane, and click the **Auto Trace** toolbar icon. The *Temporal Flow View* opens and automatically performs the trace as described in the following commands:

- **Trace This Value:** For the current time if the current value is known.
- **Trace X:** For the current time if the current value is unknown.

NOTE: The icon is only available when the **Active Annotation** option is turned *on*.

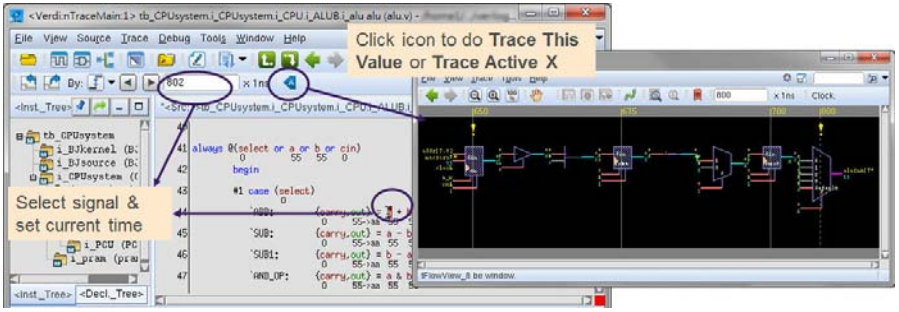


Figure: Temporal Flow View Traces Root Cause With One Command in nTrace

From nWave

In *nWave*, the root cause of issue can be traced automatically with one of these commands: **Trace This Value** or **Trace X**. Select a transition in the waveform

pane, and click the **Auto Trace** toolbar icon. The *Temporal Flow View* opens and automatically performs the trace as described in the following commands:

- **Trace This Value:** For the current time if the current value is known.
- **Trace X:** For the current time if the current value is unknown.

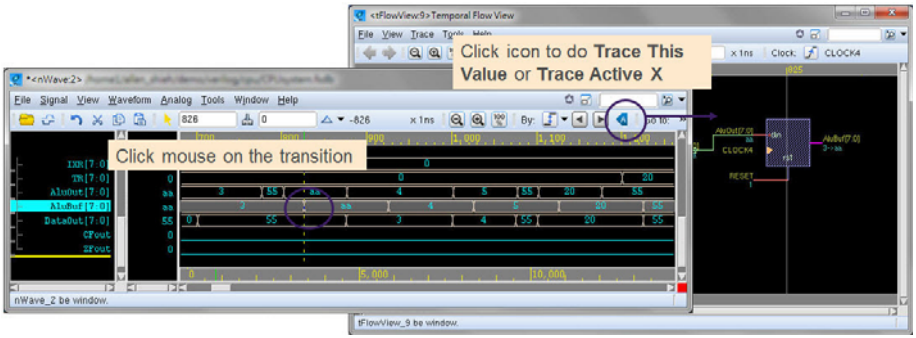


Figure: Temporal Flow View Traces Root Cause With One Command in nWave

Open Temporal Flow View

Setup Temporal Flow View

1. Invoke the *Preferences* form with the **Tools** -> **Preferences** command.
2. In the **Preferences -> Trace** page, select trace by cycle or transition by specifying the **Cycle-based** or **Transition-based** option in the **Default Trace Method** section. The default is *Transition-based*.

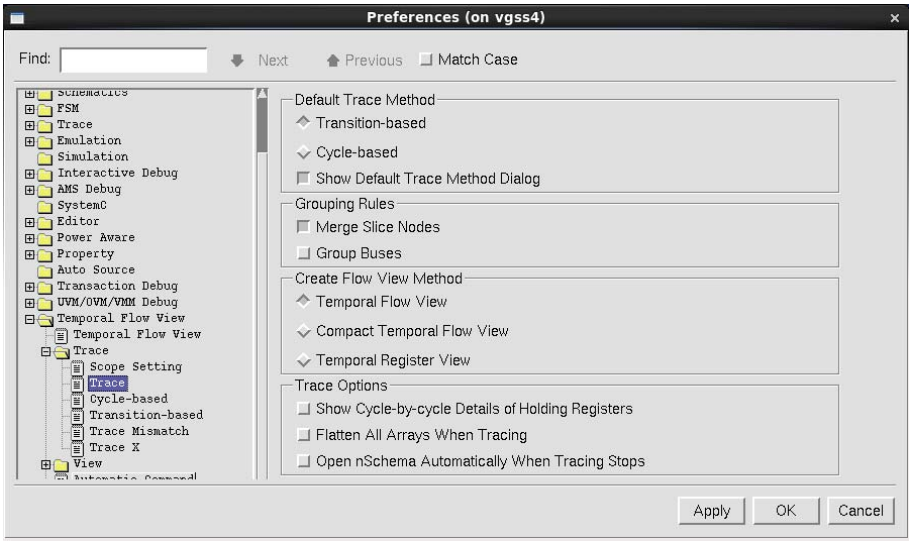


Figure: Specify Trace Method in Preferences Form

Create Temporal Flow View

1. In *nTrace* or *nWave*, select a signal transition and invoke the **Temporal Flow View -> New Temporal Flow View** command in the right-click menu command.
2. The *Temporal Flow View* frame appears in the bottom of the main window of *nTrace* and *nWave*.

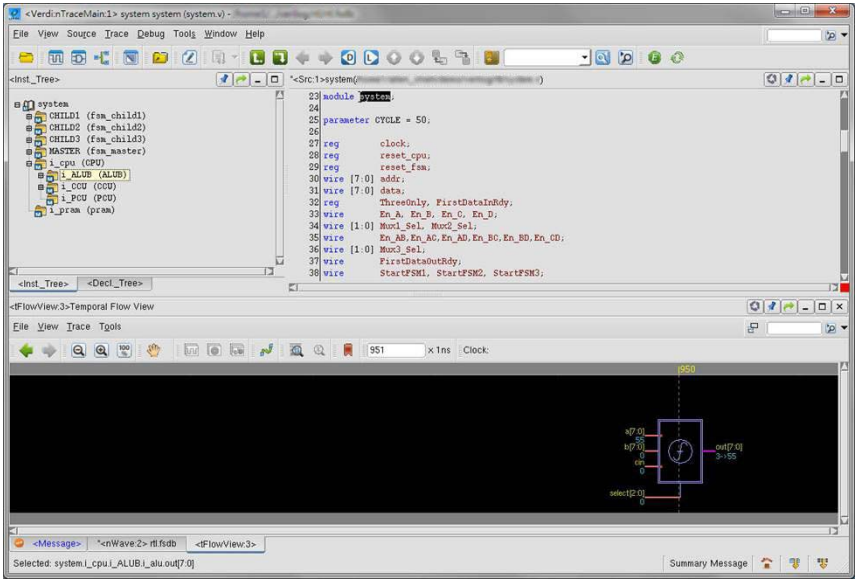


Figure: Temporal Flow View Created in nTrace

Trace with Cycle-Based Method

Tracing with the cycle-based method, the design behavior is unrolled based on clock cycles. Signals are displayed and traced according to the final stable value within a single clock cycle. The *Temporal Flow View* collects the clock domain information for each signal.

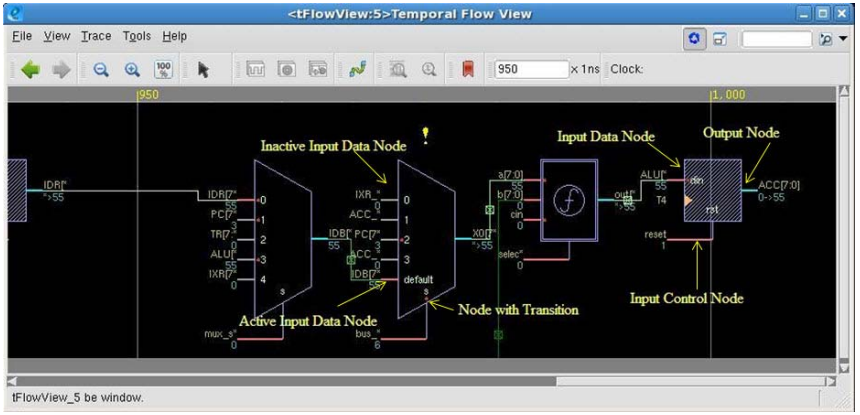


Figure: Trace With Cycle-Based Method

Trace With Transition-Based Method

Tracing with the transition-based method, input signals that trigger the output transition are automatically traced. The transition-based method are used to visualize the propagation of signal transitions throughout the design over time and is very useful for gate-level debug. The note distinguishes the transition-based tracing from the cycle-based tracing in the Temporal Flow View.

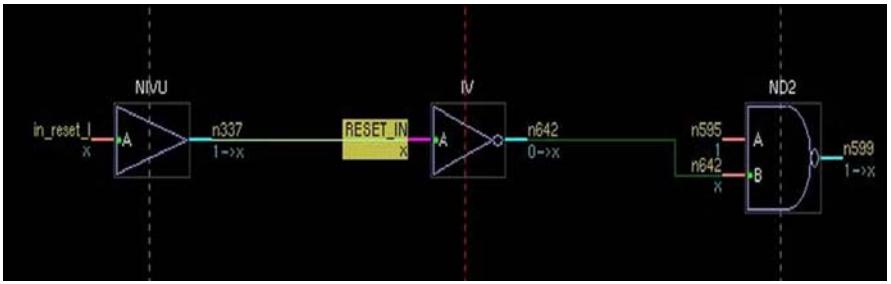


Figure: Trace With Transition-Based Method

Temporal Flow View Features to Trace Automatically

Trace This Value

Use the **Trace** -> **Trace This Value** command to trace back the data path automatically.

- Unroll the path on time axis, and find the root cause.
- Use the **View** -> **Signal -> Active Data Nodes Only** command or the **View Signal** -> **Active Nodes Only** command to simplify the view and display active nodes only.

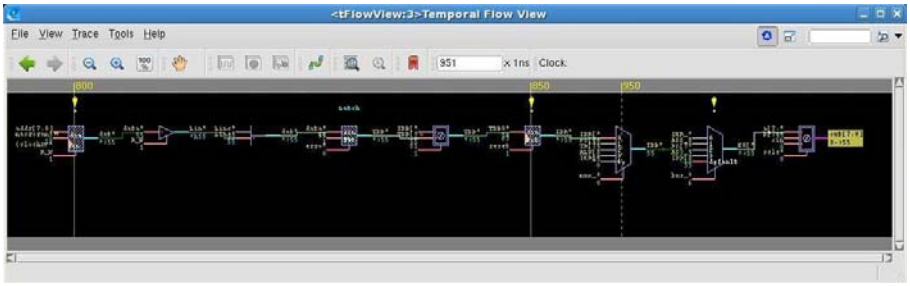


Figure: Trace This Value in Temporal Flow View

Show Marker for Control with Transition

Perform the following steps to show the output nodes with transition on the control signal in the tracing path:

1. Invoke the *Preferences* form with the **Tools** -> **Preferences** command.

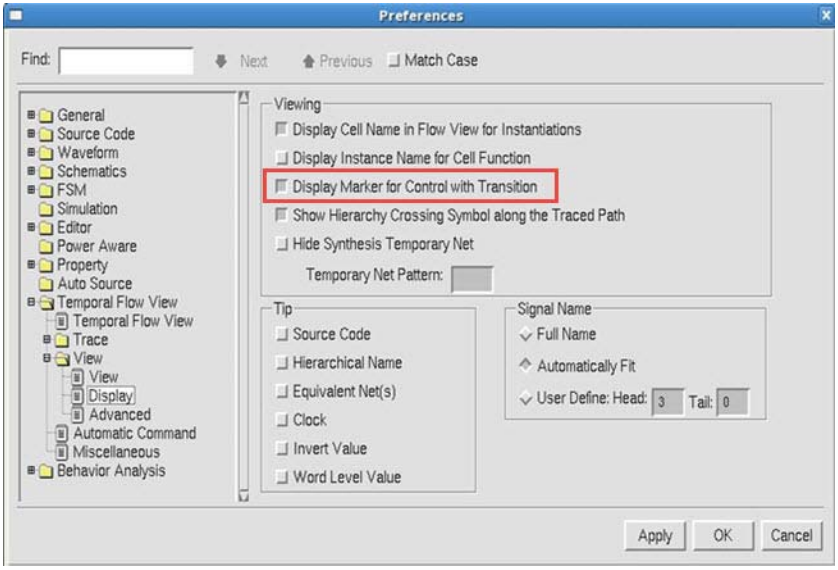


Figure: Enable Display Marker for Control with Transition in Preferences Form

2. In the **Preferences -> Temporal Flow View -> View -> Display** page, enable the **Display Marker for Control with Transition** option. A yellow exclamation point are placed over output nodes in the *Temporal Flow View* pane.

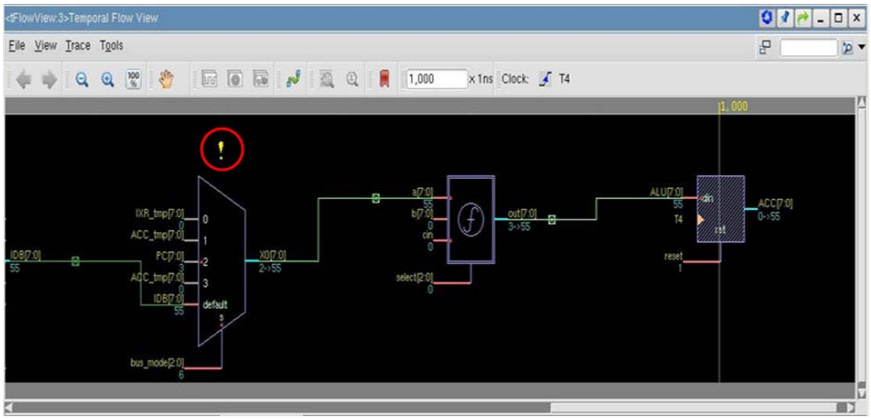


Figure: Yellow Exclamation Point Placed over Output Notes

Correlate Other Panes

Display Source Code Automatically

Toggle on the **Show Source Code Automatically** icon or the **View** -> **Show Source Code Automatically** option to view the corresponding source code of a selected node.

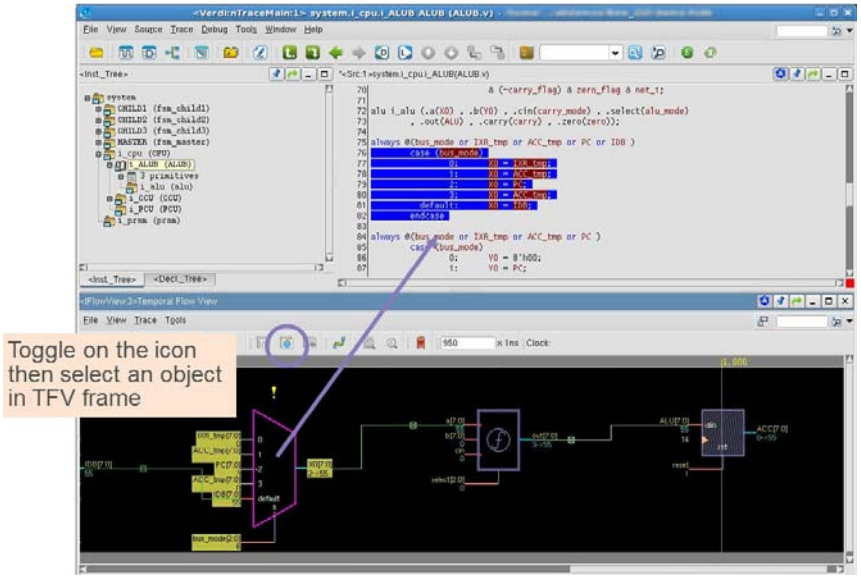


Figure: Corresponding Source Code of the Selected Node

Show Traced Signals in nWave

In the Temporal Flow View, invoke the **Tools** -> **Show All Traced Signals on nWave** command to add traced results to *nWave* to show traced signals in the waveform.

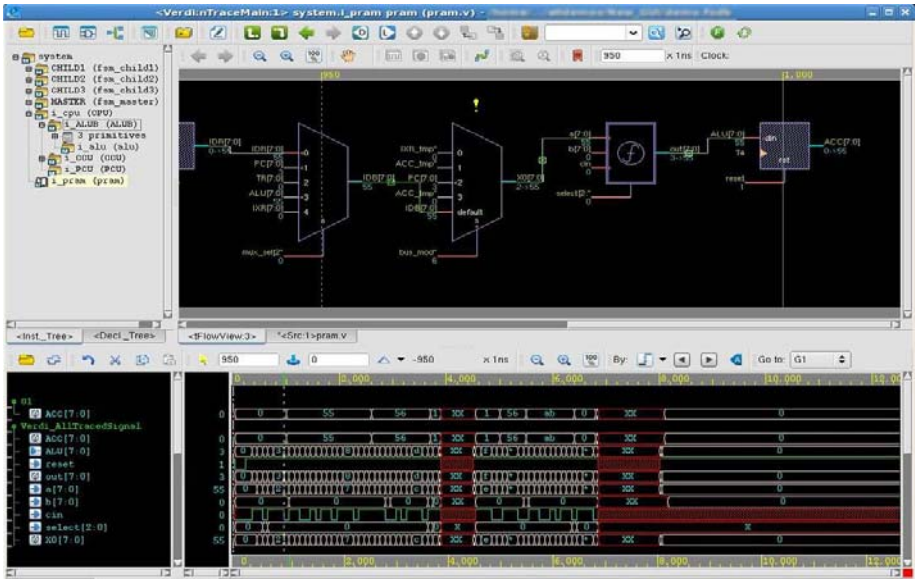


Figure: All Traced Signals Shown on nWave

Show Fan-in Signals in nSchema

Invoke the **Tools** -> **Show Fan-ins on nSchema** -> **Active Only** command in *nSchema* after selecting a signal to view the fan-in of a signal.

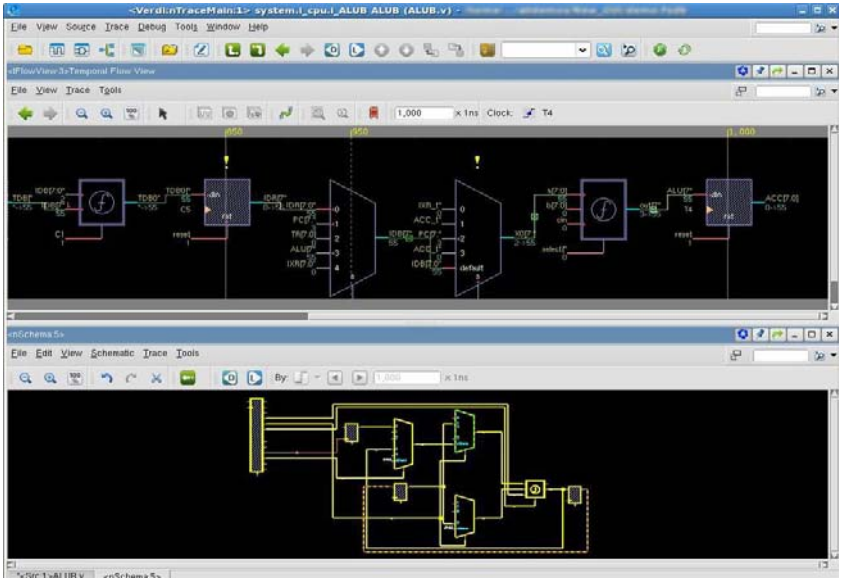


Figure: Show Fan-ins of Signal on nSchema

Temporal Flow View Application

Trace Unknown Values

Traditionally, unknown values (Xs) typically propagate through the design over multiple combinational statements.

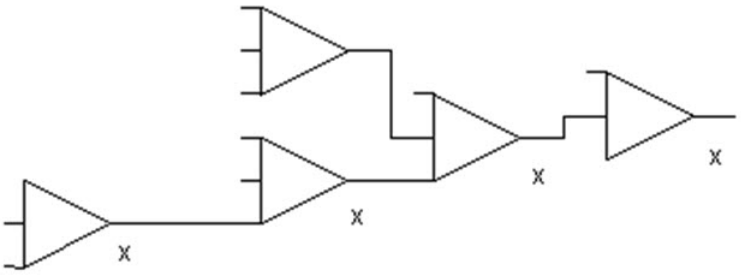


Figure: Traditional Unknown Value Trace

However, the following issues occur to debug unknowns using traditional methods:

- Display the original signal in the waveform.

- Search the source code to find all the driving signals.
- Display the driving signals in the waveform to identify the signals that are unknown.
- Repeat this process until the root cause of the unknown is found.

Tracing features of the Temporal Flow View can solve the above mentioned issues. The following methods are available:

Trace X in Verdi

1. Select the unknown signal in the Temporal Flow View and then invoke the **Trace X** command in the right-click menu.
2. Select the unknown transition in the waveform pane in *nWave* and invoke the **Trace X** command in the right-click menu.

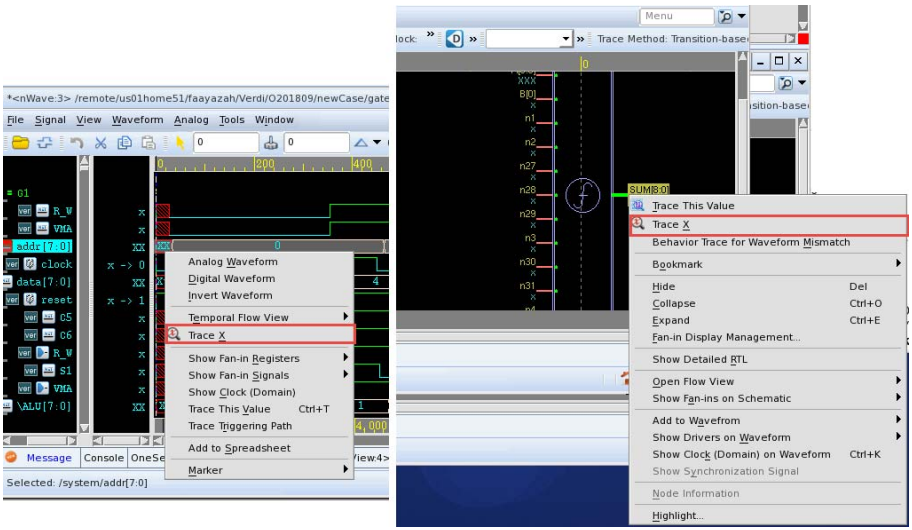


Figure: Trace X in Verdi

Trace X Setting

After invoking the **Trace X** command in the right-click menu to open the *Trace Triggering X Setting* form, the following settings can be configured:

- Follow default settings for tracing X.
- Disable the **Show Paths on Flow View** option to only view the final result in the *Trace Triggering Results* frame.

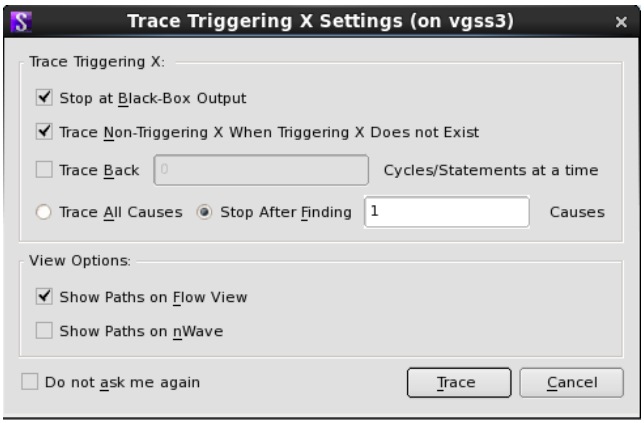


Figure: Trace Triggering X Settings Form

Trace Triggering X Results

The *Trace Triggering X Results* pane opens and display the cause(s) of the X (unknown) in a tabular format. The reason of root cause is explained in the *Note* field.

The following right-click commands are available to continue tracing in Verdi:

- Show Source Code on nTrace
- Add All Fan-in Signals to nWave
- Add Active Fan-in Signals to nWave
- Add Reference Signal
- Continue to Trace Selected Signal
- Expand Selected Cause / Merge Causes on the Same Signal

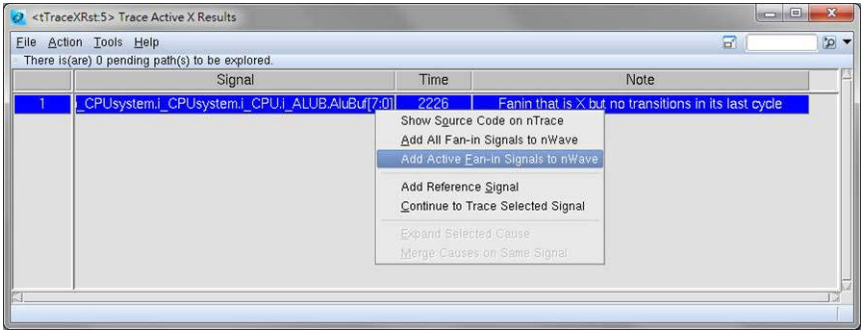


Figure: Trace Trigerring X Results Form

Summary of Steps to Trace Unknown Values

1. Select the unknown signal in Temporal Flow View, invoke the **Trace** X right-click command to trace the root cause of the unknown.
   - Set tracing constraints in the resulting *Trace X Triggering* Settings form.
2. The cause is listed in a tabular form.
   - The reason is explained in the *Note* field.

Debug Memory

Locate Last Write on Specific Address Location

When a memory output value is incorrect, the possible reason includes:

- The wrong data is written into a location.
- The data is read from a wrong location.

There are two ways to locate the last write to a memory location:

- Double-click the memory signal in the Temporal Flow View.
- Select the memory signal in *nTrace* and invoke the **Debug Memory** -> **Trace Memory Write** right-click command.

Verdi summarizes the results of the **Trace Memory Write** command in the Information form.

Verdi performs the following steps to capture the last write:

- Analyzes the control logic for the memory.
- Locates the values of these signals in the FSDB file.
- Determines when the last write to this location occurred.

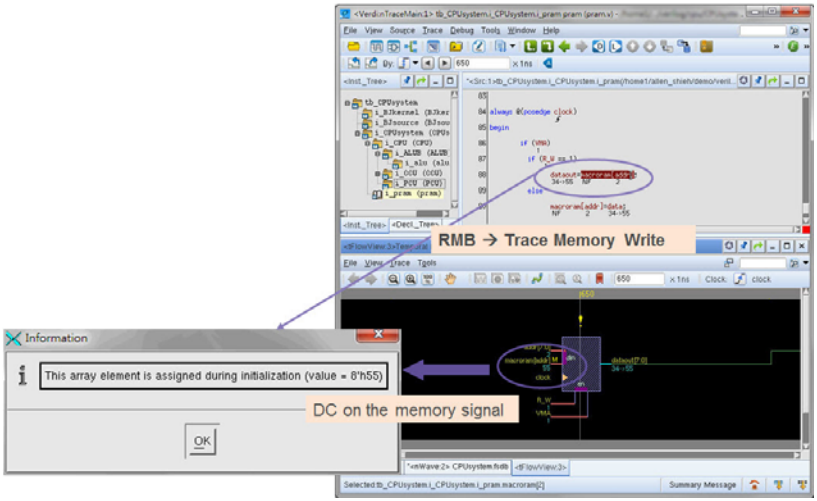


Figure: Trace Memory Write in nTrace

Display Calculated Memory Content

The following methods are available to display the calculated memory content:

- In *Temporal Flow View*, invoke the **Tools** -> **Show Memory Contents** command.
- In *nTrace*, select the memory signal and invoke the **Debug Memory** -> **Show Memory Content** right-click command.

Select the **Calculated by Verdi** tab in the *Get Memory Variable* form. The memory content is shown in the tabular format. Specify the time to display correct values in the *Time* field for displaying the correct memory content. The display mode can be changed by commands under the **Options** menu option.

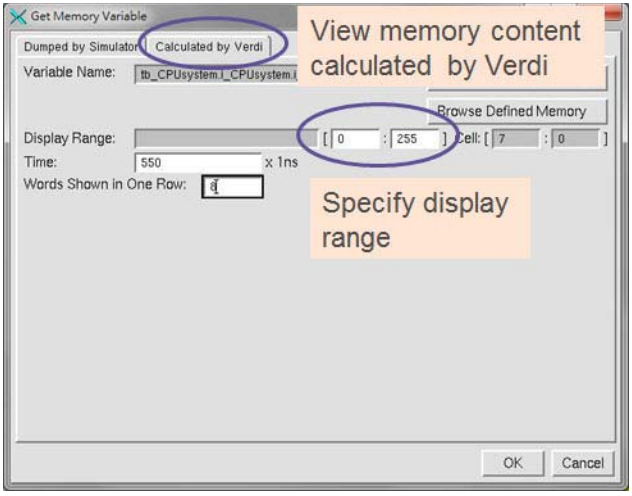


Figure: Configure Settings in Calculated by Verdi Tab

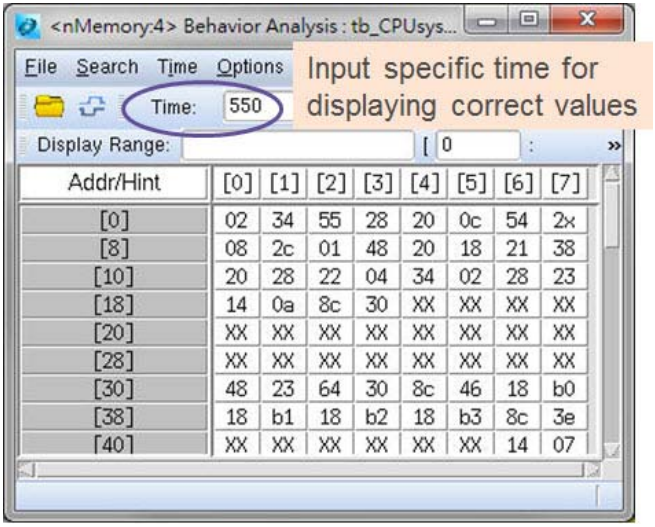


Figure: Specify Time to Display Correct Values in Time Field

Dump Memory Content to FSDB

The following methods are available to dump the calculated memory content to a new FSDB file:

- In *Temporal Flow View*, invoke the **Tools** -> **Dump memory Waveform To FSDB** command.

- In *nTrace*, select the memory signal and invoke the **Debug Memory -> Dump memory Waveform To FSDB** right-click command.

Enter the memory range and time period for the calculated memory content. FSDB file can be loaded and shown on *nWave* or *nMemory* automatically.

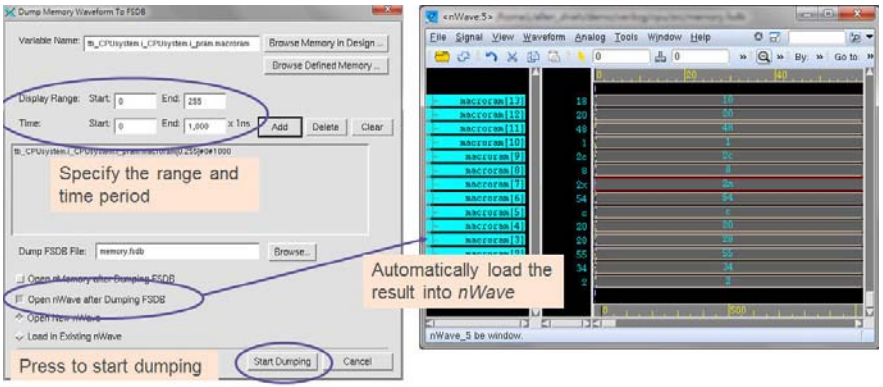


Figure: Dump Memory Content to FSDB

This tutorial focuses on the *Temporal Flow View*. The same commands can be applied to the *Temporal Register View* and the *Compact Temporal Flow View*. After reviewing the tutorial, refer to the *Behavior Trace for Root Cause of Simulation Mismatches*, *Debug Memories*, *Debug Unknown (X) Values* and *Searching Backward for Value Causes* sections in the *Application Tutorials* chapter for examples of how the *Temporal Flow View* can be applied in different debug scenarios.

Before you begin this tutorial, follow the instructions in the *Before You Begin* chapter.

Open a Temporal Flow View

1. Change your context to the *verdi_mixed* sub-directory, in which the demo source code files are located:

   ```
   % cd <working_dir>/demo/verdi_mixed
   ```

2. Compile the mixed design to create a work.lib++:

   ```
   % ./compile.verdi
   ```

3. Start the Verdi platform by referencing the compiled design and the FSDB file *CPUsystem.fsdb* (contains a set of simulation results) on the command line:

```
% verdi -lib work -top tb_CPUsystem -ssf CPUsystem.fsdb
-workMode hardwareDebug &
```

4. Resize the *nWave* window to a comfortable viewing size and locate it under the source code frame on your screen such that you can view both the windows.
5. In the **Instance** tab of the design browser frame, click the plus symbol to the left of the *i_cpusystem* block instance name to reveal its *i_cpu* and *i_pram* sub-blocks.
6. Click the plus symbol to the left of the *i_cpu* block instance name to reveal its *i_ALUB*, *i_CCU*, and *i_PCU* sub-blocks.
7. Double-click *i_ALUB* to display the associated source code.
8. In the **Find String** box on the toolbar, enter *AluBuf*.
9.  Click the **Find Next** icon (see left) to find the signal.
10. Click middle mouse button to drag and drop *AluBuf* from the *Source Code* frame to the *nWave* window.
11. Click the *AluBuf* signal in *nWave* window close to the transition from the *3* value to the *aa* value and observe the following:
    - A vertical *cursor* appears in the waveform pane.

NOTE: By default, the cursor snaps to the closest transition on the selected signal - the transition from *3* to *aa* in this case. (You can turn off the **Waveform -> Snap Cursor to Transitions** toggle command to allow you to set the cursor to any location.)

    - The simulation time of 825 associated with the cursor's current location is displayed in *nWave*'s toolbar.
12. Right-click *alubuf* in the waveform pane on the transition from *3* to *aa* at time 825 and choose the **Temporal Flow View -> New Temporal Flow View** command from the right mouse button menu.

NOTE: You can also access the *Create Temporal Flow View* form from the *nTrace* main window through the **Tools -> Temporal Flow View -> New Temporal Flow View** command.

A *Temporal Flow View* window opens as a new tab in the same area as the *nWave* window.

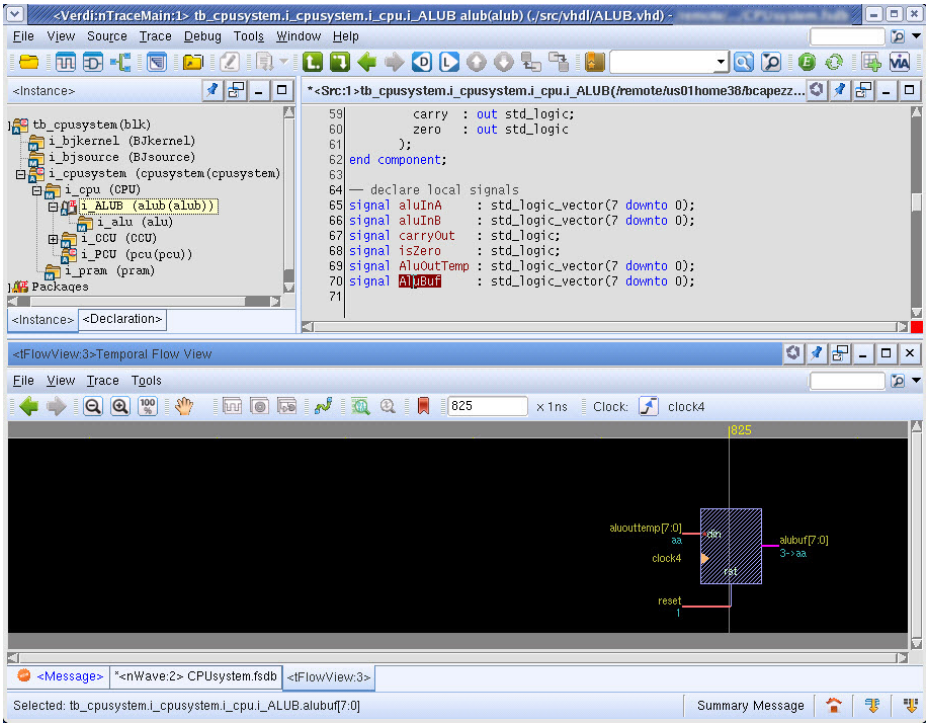


Figure: Temporal Flow View Window

Manipulate the View

You can change the view of the *Temporal Flow View* using the following zoom commands:

- **Zoom In** - View more details of the *Flow View* by moving the view 50% from the center point in both the horizontal and vertical directions. Invoke this command in one of following three ways:
  - Toolbar icon
  - Bind key "Z"
  - Menu **View -> Zoom -> Zoom In** command

- **Zoom Out** - View more contents of the *Flow View* by expanding the view 2X from the center point, both horizontally and vertically. Invoke this command in one of following three ways:
  - Toolbar icon
  - Bind key "z"
  - Menu **View -> Zoom -> Zoom Out** command

- **Zoom All** - View the entire contents of the *Flow View*. Invoke this command in one of following three ways:
  - Toolbar's icon
  - Bind key "f"
  - Menu command **View -> Zoom -> Zoom All** command
- **Zoom Area** - View more details in a specific area of the *Flow View* by dragging-left to form a rectangle over the area in pointer mode.

You can move the viewing area of the *Flow View* in different directions as follows:

- **Scrolling** - Click or drag the scroll bar of the *Flow View* window horizontally or vertically.
- **Panning** - Move the viewing area up, down, left, or right using the arrow keys on your keyboard or dragging left on the *Flow View* in Pan mode.

NOTE: You can switch between pan/pointer mode by toggling the **View -> Switch to Pointer Mode/Pan Mode** command or using the *Flow View* toolbar icon.

Display More Information

Extra information about your design can be shown using the TIP feature. You can enable the TIP feature by turning on the **View -> Turn on Tip** toggle command from the *Temporal Flow View* window. When the option is enabled, a yellow tip window is displayed when you put the mouse on top of a node.

1. In the *Temporal Flow View* frame, choose the **Tools -> Preferences** command.
2. In the *Preferences* form, click the **Display** page under the **Temporal Flow View -> View** folder.
3. Check the **Source Code** option in the **Tip** section. The corresponding line of source code is shown on the tip.
4. Check the **Hierarchical Name** option to see the full path to a signal.

   The *Preferences* form is similar to the following:

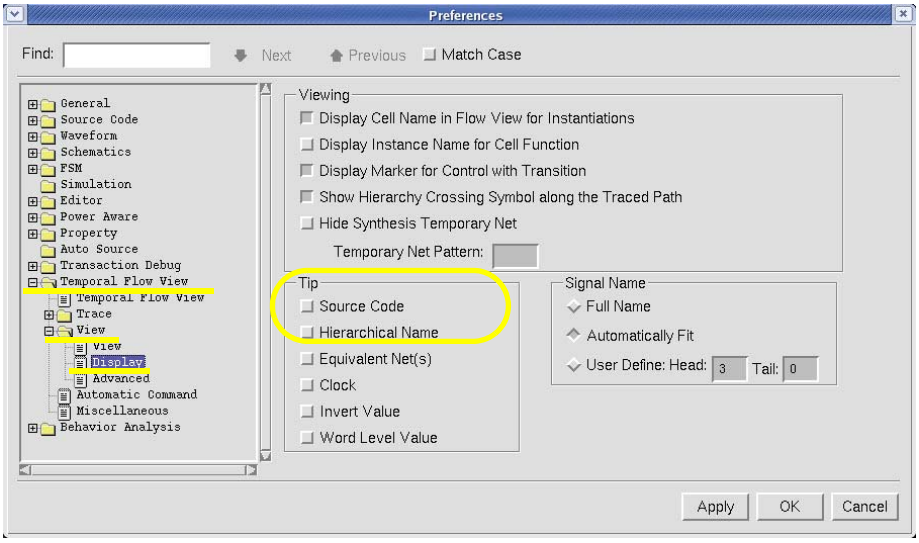


Figure: Temporal Flow View Preferences Form

5. Click **OK**.
6. In the *Temporal Flow View* frame, click the **View -> Turn on Tip** command to turn on the TIP feature.
7. Move your cursor over the *alubuf* node to view the associated line of code and hierarchical path.

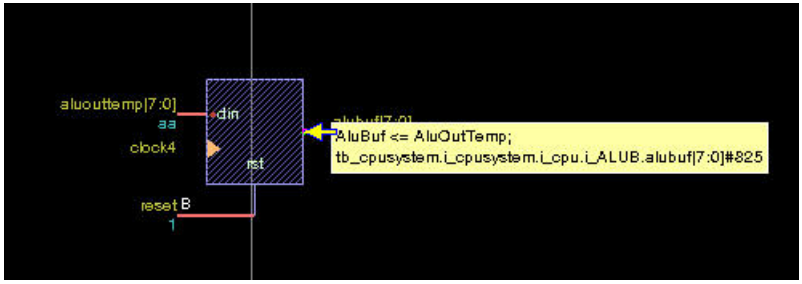


Figure: Tip for alubuf

Show Active Statements

An active statement is the logic statement that generates the value for the driver signals of the current statement.

1. Double-click the *aluouttemp[7:0]* signal on the *Temporal Flow View*. You can also right-click the *aluouttemp* node in the *Temporal Flow View* and choose the **Show Active Statement** command.

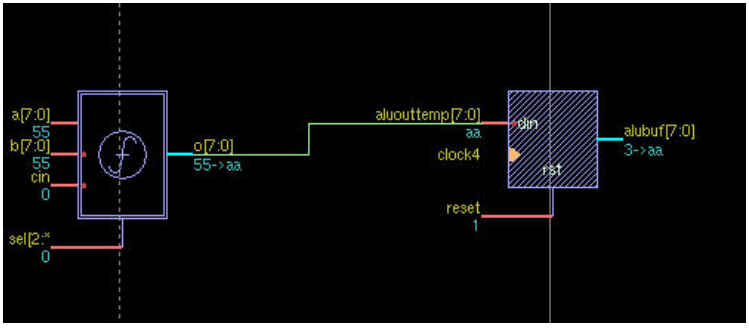


Figure: Temporal Flow View Displaying Active Statements

   Note that another symbol (a function block) is added to the *Temporal Flow View*. You can continue tracing active statements on any input node.

2. Double-click *a[7:0]* to expand and display another function symbol.
3. Double-click *b[7:0]* to expand and display a *mux*.
4. You can use the **View -> Signal -> Active Nodes Only** command to toggle between viewing all nodes or only the active nodes. (Active nodes have a pink color to the port stub on the symbol.)

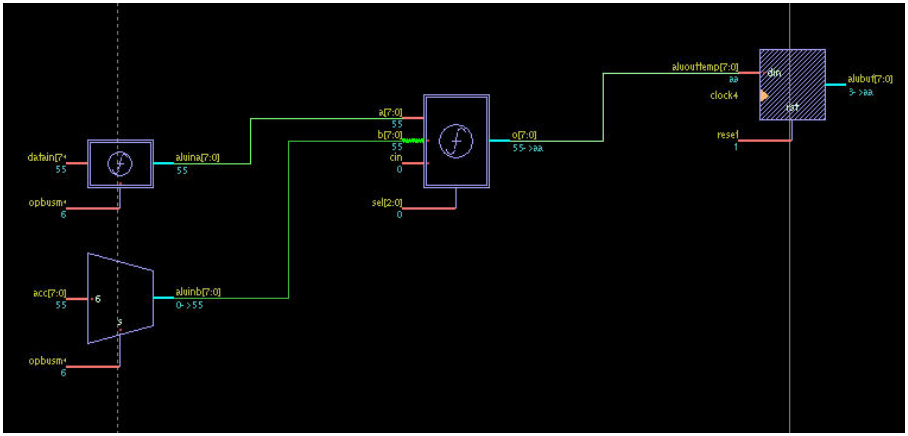


Figure: Temporal Flow View Displaying Active Nodes Only (in pink)

5. Use the zoom icons to manage the display. Click the **Fit Time** icon to perform a full zoom of the symbols.
6. Turn off the **View -> Signal -> Active Nodes Only** toggle command again.

Display Source Code

Select the display area of the source code from the *Temporal Flow View*. You can choose to display the source code in an existing *nTrace* window or a new window.

1. Choose the **Tools -> Preferences** command.
2. Select the **Automatic Command** page under the **Temporal Flow View** folder, and choose the preferred method for **Show Source Code Automatically on**.
3. In this tutorial, select the **New nTrace Window** option.
4. Click **OK**.

5. In the *Temporal Flow View*, click the **Enable Show Source Automatically** icon on the toolbar.
6. Click the *alubuf* node at time 825.

   The corresponding active (driving) statement is located and highlighted in a new tab of the source code frame:

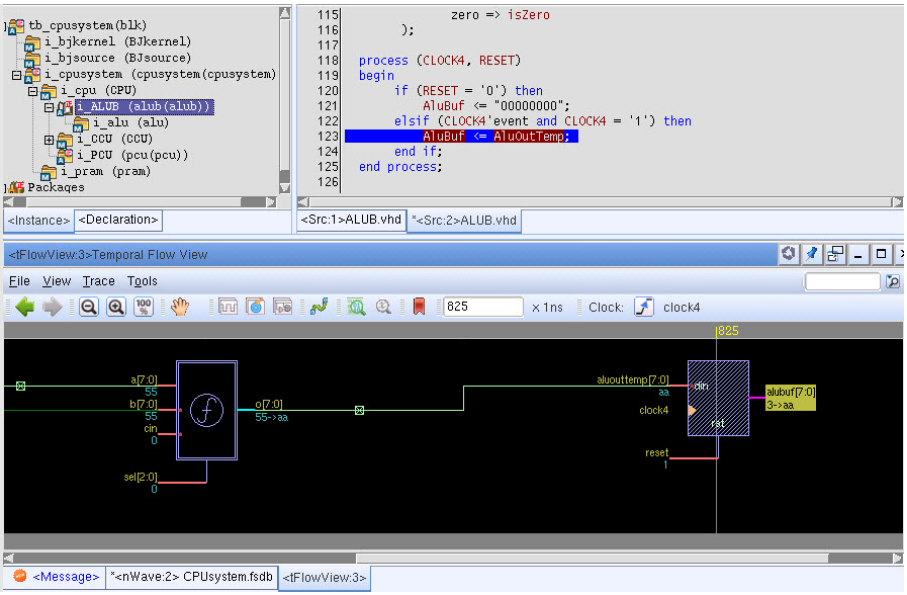


Figure: nTrace Window Displaying Source Code from Temporal Flow View

The source code statement shows how the *AluOutTemp* and *RESET* signals functionally contribute to the contents of the *AluBuf* register. In particular, observe that *AluBuf* is assigned the value from *AluOutTemp* in a VHDL process.

Note that this is a mixed language design so the signal is *alubuf* in the waveform and flow views and *AluBuf* in the source code.

7. In the *Temporal Flow View,* click the output node *o[7:0]* of the function symbol.

   Note that a single line in a Verilog case statement is highlighted.

8. Click the *aluina[7:0]* output node of the second (left-most) function symbol.

   Note that a single line in a VHDL process is highlighted.

9. Click the **Disable Show Source Code Automatically** icon to disable the showing source code.

Add Signals from the Temporal Flow View to nWave

There are several methods for adding signals to *nWave* from the flow view.

1. To simplify things, drag-left over all of the signal names in the signal pane to select the signals and then select the **Signal -> Cut** command to remove the signals from the display.
2. Click *alubuf* in the *Temporal Flow View* to select it, and then select **Ctrl+W** to add the signal's waveform to the *nWave* window.
3. Select **Ctrl+K** to add the clock signal of the register to the *nWave* window.
4. Select the output node *o[7:0]* of the function symbol.
5. Select **Ctrl+A** to add all the fan-in signals to the *nWave* window.
6. Select the output node *aluina[7:0]*.
7. Select **Ctrl+F** to add only the active fan-in signals to *nWave*.

   Note that only the selected signal and the input ports with red colors (*datain*, *opbusmode*) are added.

NOTE: You can change where and how the signals are added through the options in the **nWave** section of the **Automatic Command** page under the **Temporal Flow View** folder on the *Preferences* form.

The *nWave* results are similar to the following figure:

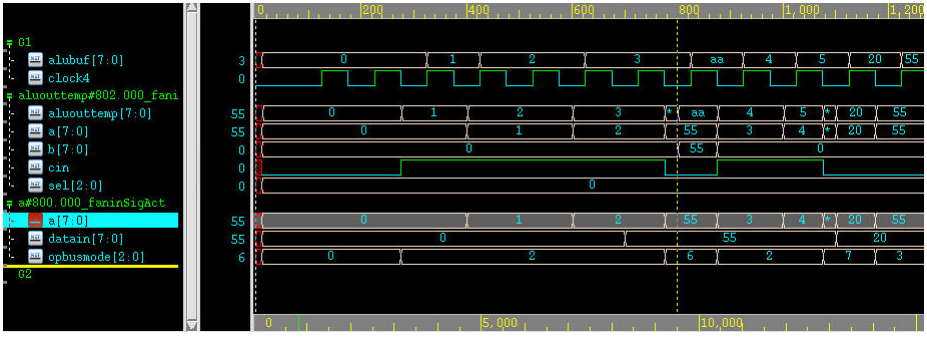


Figure: nWave Window Displaying Results of Adding Signals from Temporal Flow View

Compact Temporal Flow View

The *Compact Temporal Flow View* displays a global view of the usage/definition of design signals and variable elements where the control signal and data signal interactions are presented at the statement and signal level. This representation and view is complete with data/control signals active/inactive identifications, signal dependencies, and timing annotations.

You can open a *Compact Temporal Flow View* from a *Temporal Flow View*:

1. In a *Temporal Flow View* frame, left-click a signal to select it (for example, *alubuf[7:0]*).
2. In the *Temporal Flow View* window, choose the **Tools -> Open Flow View -> Compact Temporal Flow View** command.
3. In the *Compact Temporal Flow View* window, double-click a node to see which statement defines the value for it.

   The Compact Temporal Flow View window updates to reflect your choices, similar to the following example:

Figure: Active Statements - Compact Temporal Flow View

From the figure above, note the following:

- The yellow rectangular box associated with *AluBuf* at time 825 indicates that *AluBuf* is a register, and its clock edge arrives at time 825.
- The yellow octagon associated with *out* at time 800 indicates *out* is an internal signal.
- The gray line linking *AluOut* to *out* indicates that *AluOut* is driven by *out*.

- The red line surrounding the *a*, *b*, *cin*, and *select* signals. This indicates that these signals are inputs to the statement and the output is the *out* signal.
- The small red dot to the upper left of the octagon associated with *AluOut* indicates that the last assignment to this signal has a value change.
- A vertical dashed line at time 800 indicates that *out* signal can be traced back to registers activated at time 800. That is, ignoring delay of the combinational logic, the *out* signal is generated at time 800.

NOTE: The yellow box and yellow octagon associated with *AluBuf* and *out*, respectively, indicate that these are "output nodes." This does not imply that they are primary outputs. However, they are the outputs associated with their respective assignment statements in the source code.

- "D" and "C" characters indicate if these signals are contributing data and/or control values.

NOTE: After enabling the source code display, the source code can be displayed similar to the *Temporal Flow View* by clicking on “output” nodes.

NOTE: Signals in the *Compact Temporal Flow View* can be added to the waveform in a similar manner to the *Temporal Flow View* through drag and drop or bind keys.

Showing Statement Flow in an *nSchema* Frame

In a *Compact Temporal Flow View* window, perform the following steps to show the statement flow of a debug path in an *nSchema* window*:*

1. In the *Compact Temporal Flow View* window, turn on the **View-> Enable Flow Schematic Automatically** toggle command to turn on the schematic display mode.
2. In the *Compact Temporal Flow View* window, click an output node to select it.

   A symbol representing the statement is shown in an *nSchema* window.
3. In the *Compact Temporal Flow View* window, click another output node to add its associated symbol to the *nSchema* window.
4. Turn off the **View-> Disable Flow Schematic Automatically** toggle command to disable the schematic display mode.

Temporal Register View

The *Temporal Register View* displays the design interaction at the register level only. All the intra-cycle combinational interaction is abstracted away. This view is possible only using the cycle-based engine. Similar to the *Temporal Flow View*, this representation and debug view is annotated with timing, activity, and control/data signals functionality.

You can open a *Temporal Register View* window from a *Temporal Flow View* window as follows:

1. In the *Temporal Flow View* frame, select a signal (for example, alubuf) and choose the **Tools -> Open Flow View -> Temporal Register View** command. A *Temporal Register View* window appears.
2. Double-click a fan-in register to see the active registers. You can also select either of the following methods to view the active registers:
   - Use the right mouse button and choose **Show Fan-in Registers**.
   - From the main menu, choose the **Trace -> Show Fan-In Registers** command.

   The *Temporal Register View* window is updated, as shown in the following figure:

Figure: Active Registers in Temporal Register View

Refer to the figure above and note the following:

- In *Temporal Register View* notation, the box on top of the arrow represents the fan-out register, and the boxes below the arrow represent the set of fan-in registers or primary inputs.
- The gray vertical line represents the clock edge of the output register.
- If you click the box associated with any register, a number appears in the time box on the toolbar, which indicates the time the register changed. The active clock edge is also displayed.

NOTE: Selection can be changed choosing the **Mouse Click** or **Automatically Select When Mouse over** options on the **Miscellaneous** page under the **Temporal Flow View** folder of the *Preferences* form (invoked with the **Tools -> Preferences** command). For the value shown in the figure above, each fan-in register represents the value before the active clock edge of *AluBuf*. These are the stable values of the fan-in and primary input at this cycle.

- Note the "C" and "D" characters in the fan-in register boxes. These characters indicate if the fan-in registers are contributing control and/or data paths to *AluBuf*. (An example of a control path is a multiplexer's select input, while an example of a data path is a multiplexer's data signals.) In particular, note that *CWR* has both a "D" and a "C," which indicate that some of this register's bits are acting as data and others are providing control.
- The boxes associated with the *CWR*, *ACC*, and *IDR* registers are pink, which indicate that these fan-in nodes are actively contributing to the value in *AluBuf* at time 825. By comparison, the *IXR* and *PC* registers are gray, which indicates that these fan-in notes are not actively contributing to the value in *AluBuf* at time 825.
- There is a red dot on the left upper corner on *ACC*, *PC*, and *CWR*. That is, these fan-in registers have value transition in its previous clock edge. Since ACC and CWR are also active fain-in registers, they are active fan-in with transition.

NOTE: After enabling the source code display, the source code can be displayed similar to the *Temporal Flow View* by clicking on “output” nodes.

NOTE: Signals in the *Temporal Register View* can be added to the waveform in a similar manner to the *Temporal Flow View* through drag and drop or bind keys.

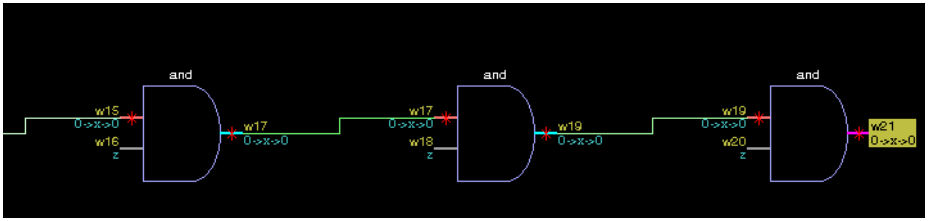
Trace the Root-cause of glitches in the Design

You can debug a glitch in *Temporal Flow View* by identifying annotated glitches and locating the root source of the glitch. The GUI commands are available only when the selected signal contains glitches in its value. This command does not create a report but only helps trace signals with glitches.

Add the following options during simv runtime to dump the glitch information:`+fsdb+glitch=0 +fsdb+sequential` before you begin tracing a signal glitch.

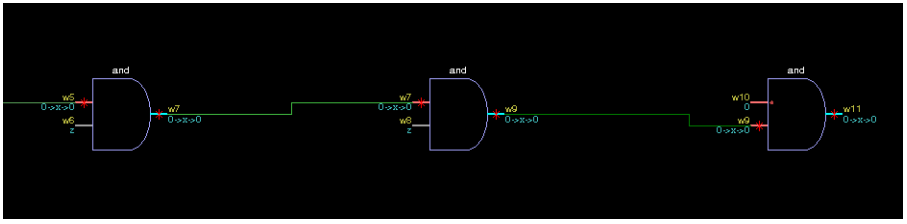
When you invoke the **Trace Glitch** command from the *nWave* menu option, or from the right click command on the *Temporal Flow View*, the **Display Glitch** option is automatically turned on.

If a reference signal in the *Temporal Flow View* has a glitch, it is marked with a * symbol and the complete transition value is displayed, even if it is on the input side of the function block as shown in the following figure:



To trace the glitches in a signal, select **Trace** -> **Trace Glitch** command from the right-click command menu in the *Temporal Flow View*.

When you trigger this command on a reference signal, the path in the *Temporal Flow View* is expanded along all the fan-in signals that contain glitch values.The fan-in does not need to be active or have value change in order to continue tracing along that path as shown in the following figure:



Debug a Design with Simulation Results Tutorial

Before you begin this tutorial, follow the instructions in the *Before You Begin* chapter.

This tutorial shows you how to use the Verdi platform to debug a simulation failure in a RTL design. A Verilog example is used; however, the same debug techniques apply to VHDL or mixed designs.

Find the Active Driver

Assume the transition from 3 to 55 of *ALU[7:0]* at time 951 is wrong, and you have to discover the real cause(s).

1. Change the directory to <*working_dir*>*/demo/verilog/rtl*. Execute the following command to import the design and load the simulation results:

   ```
   % verdi -f run.f -ssf rtl.fsdb -workMode hardwareDebug &
   ```

2. In the **Instance** tab of the *Design Browser* frame, click the "+" of *i_cpu(CPU)* folder to expand the hierarchy.
3. Drag and drop *i_ALUB(ALUB)* into *nWave* frame to display the signals.

NOTE: You can also use **Get Signals** in *nWave* to select the signals of interest.

4. Choose the **Source -> Active Annotation** command to annotate the simulation results into the *Source Code* frame.
5. In *nWave*, double-click the transition from 3 to 55 of *ALU[7:0]* at time 951 ns to locate the source of the transition.

   This action displays the active driver of the signal transition in the *Source Code* frame. For this example, it is signal *out* on line 52 of *i_alu(alu)*, as shown below:

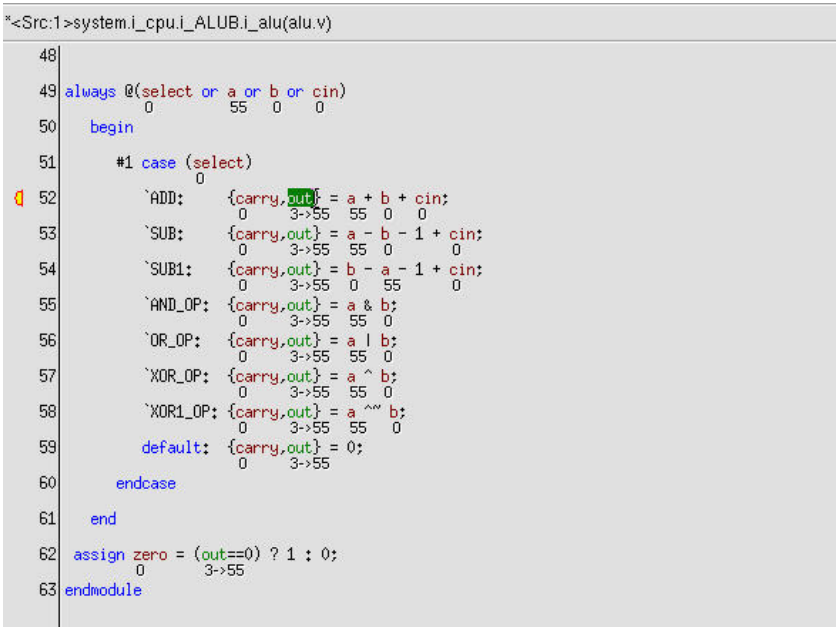


Figure: Active Driver of Signal Transition

NOTE: *ALU* has crossed a hierarchical boundary, and changed names to *out*. Look at the equation for *out*. It is an OR of *a*, *b*. and *cin*. Which signal matches the incorrect value of 55?

With active annotation, it is clear the source of the 55 value is signal *a*.

6. Locate all drivers of *a* by double-clicking *a* on line 52.

   There are five drivers reported in the message frame. It would take time to trace back the logic of all drivers.

   Let's find the real active driver *a* to dramatically reduce the effort.

   Note that the time is 951 ns.

7. Click the **Backward History** icon (see left) on the *nTrace* main window toolbar to return to previous step.
8. With signal *a* selected, click the right mouse button menu, and choose **Active Trace** to find the real driver (or **Ctrl+T**).

   Now, *X0* in line 81 is selected, which means that the real driver is coming from this line.

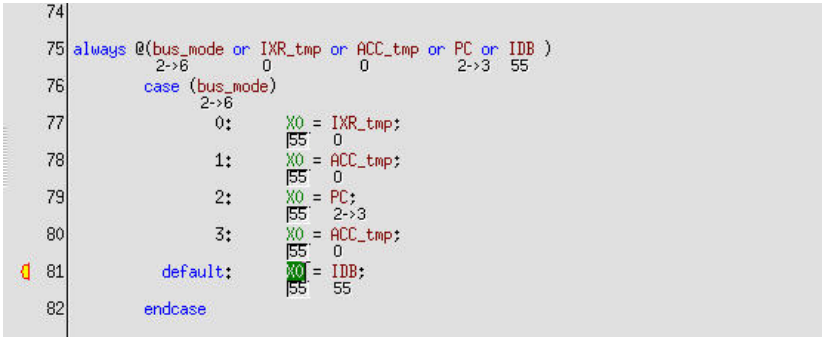


Figure: Partial nTrace Window Displaying Active Trace for Real Driver

NOTE: Signal *a* changes the name to *X0* in *ALUB.v* because of its connectivity.

An *Information* dialog window displays (shown below) to denote a 1ns change in time.

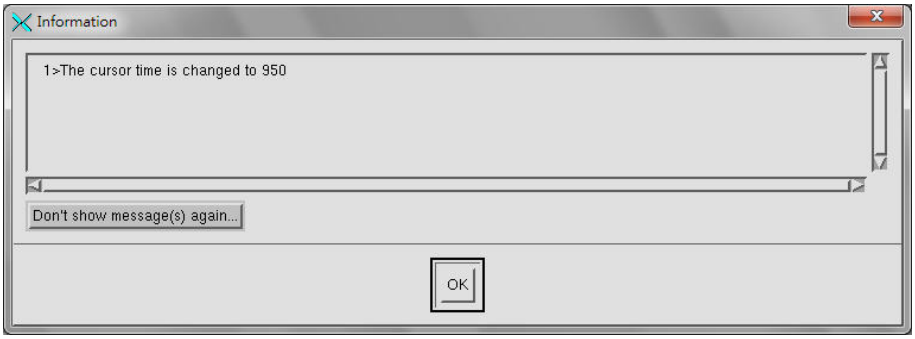


Figure: Information Window Displaying Change in Time

9. Click **OK** on the *Information* dialog.

Generate Fan-in

Active trace can be performed multiple times until the cause is located. However, you may need to display a schematic that shows only the logic driving *IDB* (the active driver of *X0*), independent of hierarchy.

1. To generate the **Fan-In** for *IDB*, select *IDB* in line 81 on source code frame.
2. Choose the **Tools -> New Schematic from Source -> Fan-in** command.

   A flattened *nSchema* frame opens as a new tab in the same area as the source code frame, displaying the logic driving *IDB*. You can select specific blocks, and find they are from different hierarchies.
3. In the *nSchema* frame, choose the **Schematic -> Active Annotation** command to annotate the simulation results in the **Fan-in** window.

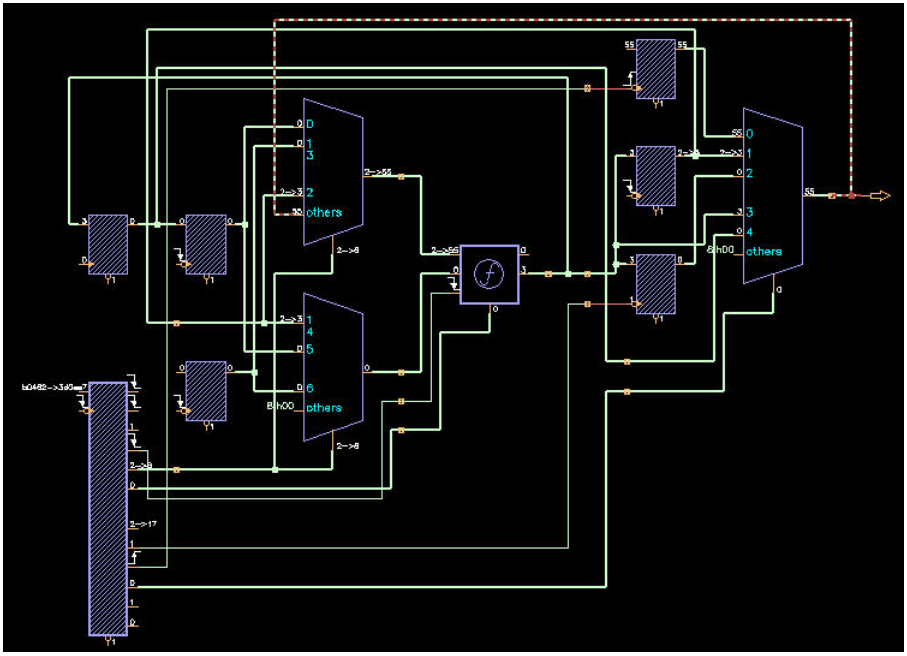


Figure: nSchema Frame Displaying Annotated Fan-in Schematic

4. Use the zoom function (drag-left in the upper right around the three registers and mux) to view the values on the nets in detail.

   Let's analyze the generated **Fan-in** to locate the real cause of the result.
5. Choose the **View -> Net Name** command to display signal names on the schematic.

IDB is driven by a mux, so it is important to know the value of the select line in order to determine which input is active. In this scenario, the current value on the select line is 0, making the topmost input your starting point. The top input of the mux is coming from a storage element.

6. Select *IDR[7:0]* (the output pin of the storage element driving the mux input) and click the **Search Backward** icon on the toolbar to locate its transition to '0->55'.

   The time should now be 850ns.

7. Double-click the input pin of the storage element to trace the logic back. It is a mux.

NOTE: Fan-In stops at storage elements, functional blocks, FSMs, and primary IOs.

8. You want to know when the register input transitions to 55. Select the net (*TDB0[7:0]*) between the register and mux and click the **Search Backward** icon on the toolbar.

   Note the time changes to 800 and the value on the control signal of the mux is 1.

   Continue tracing the 55 value back through the signal by completing the following steps:

9. Double-click the second data input pin of the mux to expand the driving logic. It is driven by a latch.
10. Double-click the data input of the latch, which is driven by a pair of tri-state devices.

    At this point, the schematic is getting a little cluttered, so let's continue on a fresh schematic.

11. Select the output of the tri-state and then choose the **Tools -> New Schematic -> Fan-in** command to generate another **Fan-in.**

    In the new *nSchema* frame, a schematic with its output driven by two tri-states and memory component is displayed, as shown below:

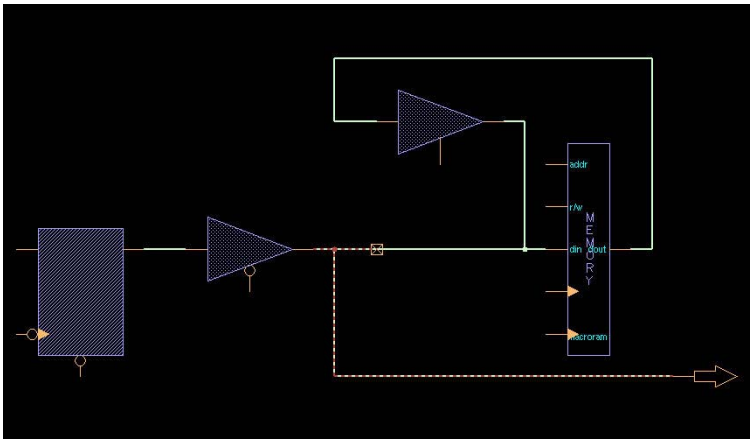


Figure: Example Tri-State and Memory Component

12. Annotate simulation results using the **Schematic -> Active Annotation** command (or use the 'x' bind key).

    The enable pin of the tri-state is active low, and its current value is 1, which indicates that the output is driven by the memory component.

Debug Memory Content

You know the bad value is related to the memory. Now you want to look at the memory. Use the following steps to load memory content.

1. In the *nTrace* main window or the *nWave* frame, choose the **Tools -> Memory -> Memory/MDA** command, which opens an *nMemory* frame in the lower right.
2. In the *nMemory* frame, choose the **File -> Get Memory Variable** command.
3. On the **Dumped by Simulator** tab of the *Get Memory Variable* form, select *system.i_pram.macroram*.
4. In the **Display Range** text fields, specify *0* for start address and *64* for the end address.

   You know this is the only part of the memory that is used during this simulation. The form will be similar to the following:

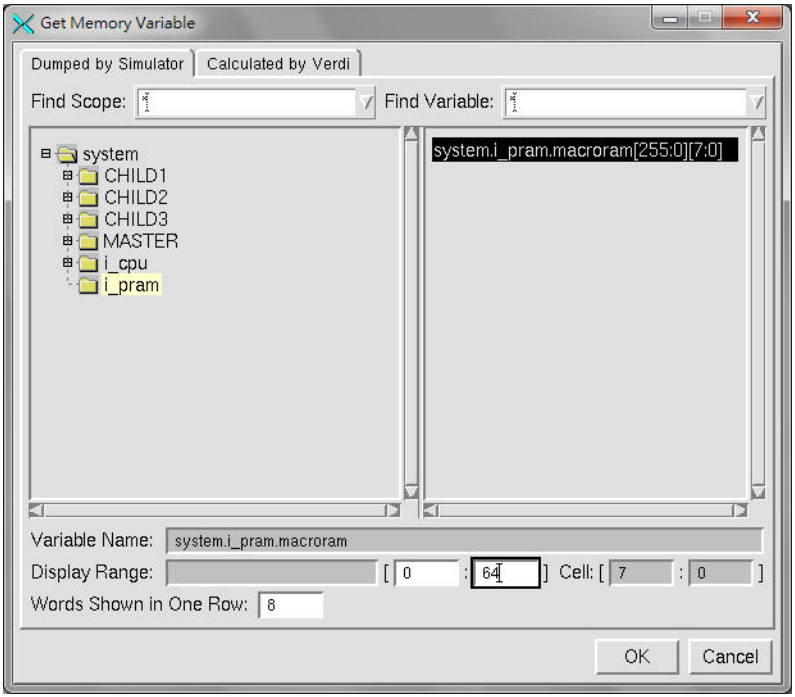
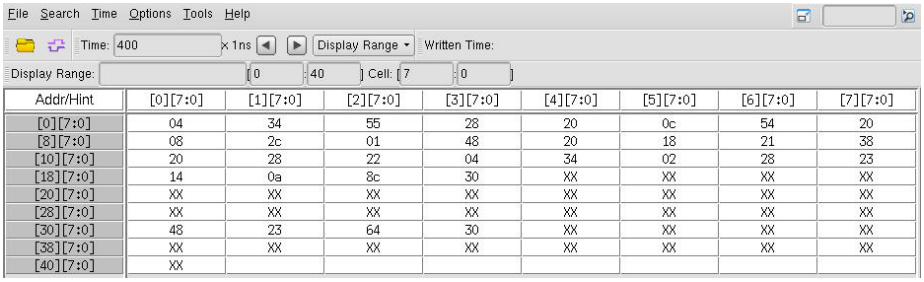


Figure: Get Memory Variable

5. Click the **OK** button. The specified memory and address range will be loaded into *nMemory*, similar to the following:



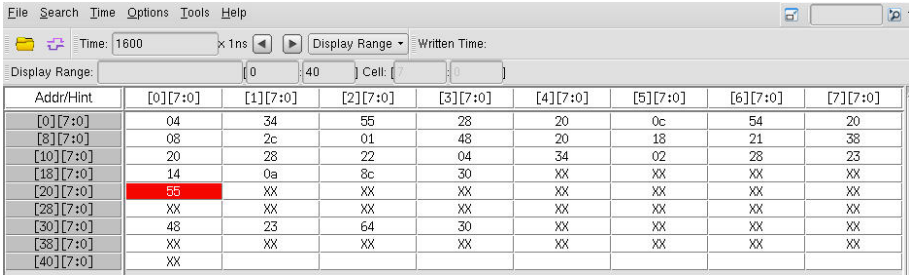
Addr/Hint	[0][7:0]	[1][7:0]	[2][7:0]	[3][7:0]	[4][7:0]	[5][7:0]	[6][7:0]	[7][7:0]
[0][7:0]	04	34	55	28	20	0c	54	20
[8][7:0]	08	2c	01	48	20	18	21	38
[10][7:0]	20	28	22	04	34	02	28	23
[18][7:0]	14	0a	8c	30	XX	XX	XX	XX
[20][7:0]	XX	XX	XX	XX	XX	XX	XX	XX
[28][7:0]	XX	XX	XX	XX	XX	XX	XX	XX
[30][7:0]	48	23	64	30	XX	XX	XX	XX
[38][7:0]	XX	XX	XX	XX	XX	XX	XX	XX
[40][7:0]	XX							

Figure: nMemory Window

Use the following steps to look for the new data.

6. In the *nMemory* frame, turn on the **Time -> Sync Cursor Time** toggle command.

7. Click the **Next Dump** icon (see left) on the memory toolbar until you see one of the addresses highlighted in red.



Addr/Hint	[0][7:0]	[1][7:0]	[2][7:0]	[3][7:0]	[4][7:0]	[5][7:0]	[6][7:0]	[7][7:0]
[0][7:0]	04	34	55	28	20	0c	54	20
[8][7:0]	08	2c	01	48	20	18	21	38
[10][7:0]	20	28	22	04	34	02	28	23
[18][7:0]	14	0a	8c	30	XX	XX	XX	XX
[20][7:0]	55	XX	XX	XX	XX	XX	XX	XX
[28][7:0]	XX	XX	XX	XX	XX	XX	XX	XX
[30][7:0]	48	23	64	30	XX	XX	XX	XX
[38][7:0]	XX	XX	XX	XX	XX	XX	XX	XX
[40][7:0]	XX							

Figure: nMemory frame Displaying a Address Highlighted in Red

In this case, the data value at *address[20]* is *55* at a simulation time of *1600* ns.

Use the following steps to trace memory content in *nWave*.

8. In the *nMemory* window, choose the **Search -> Find** command.
9. In the *Find* form, enter *55,* and use the binocular icons (**Find Next** or **Find Previous**) to search for the other value of 55, which is located at *address[2]*. Another location is *address[20]*.
10. Select the address locations with value *55* by holding the Ctrl key to select multiple locations.
11. Use the middle mouse button to drag and drop the data to *nWave* frame to see the waveform, as shown below:

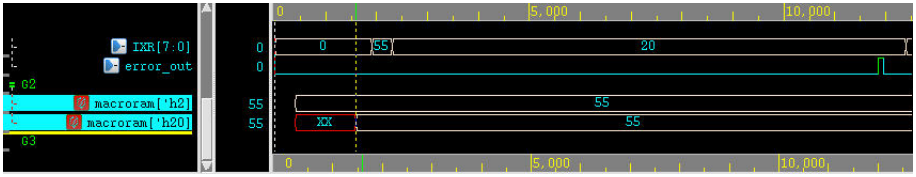


Figure: Memory Waveform

nCompare Tutorial

Overview

The *nCompare* frame compares simulation results stored in FSDB dump files using flexible, user-specified comparison criteria. Optimized for the extremely fast comparison of large data sets, the *nCompare* frame is fully integrated with Verdi platform to intuitively display any differences between runs.

The details about what to compare are described in a rule file that is written in Tcl. The *nCompare* module uses Tcl language and *nCompare*-defined-Tcl-extended comparison commands (refer to the *Appendix D* in the *Verdi and Siloti Command Reference Manual* for details) to describe the comparison rules and specific comparison options.

The following three parts should be included in a basic rule files:

1. Specify golden and secondary simulation files.
2. Specify compared signal pairs.
3. Start time-based comparison.

Follow this tutorial to learn how to use *nCompare*'s basic functionality.

Start nCompare and Compare Waveforms

After the *nCompare* frame is opened in the Verdi platform, the frame can be used to import a rule file in one tab and to compare and view the mismatches in another tab.

Perform the following steps to open an *nCompare* frame and import a rule file:

1. Change to the *nCmp_demo1* directory.

   ```
   % cd <working_dir>/demo/nCompare/nCmp_demo1
   ```

2. Invoke the Verdi platform.

   ```
   % verdi -workMode hardwareDebug &
   ```

3. In the main window, choose **Tools -> New Waveform** command to open the *nWave* window.
4. In the *nWave* window, choose **Tools -> nCompare** command to open the *nCompare* frame. The *nCompare* frame opens as a new tab in the same location as the source code frame, as illustrated in the following figure:

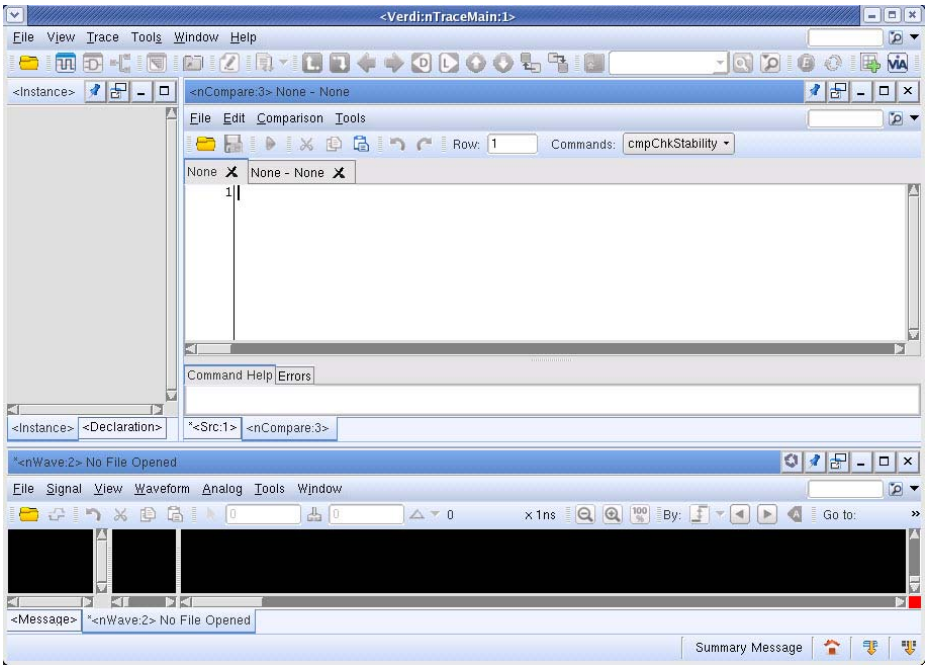


Figure: nCompare Frame

5. In the *nCompare* frame, choose **File -> Open Rule File** command. The *Open Rule File* form opens, as illustrated in the following figure:

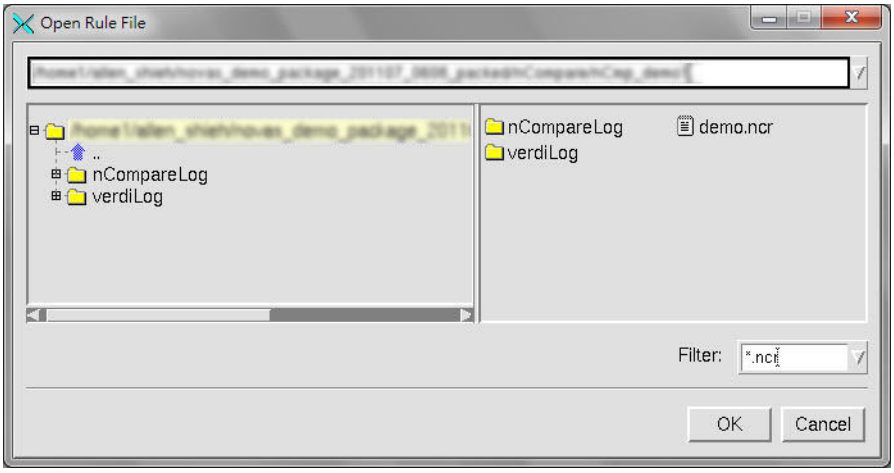


Figure: Open Rule File Form

6. Select the *demo.ncr* file in the *Open Rule File* form. The opened rule file is displayed as another tab in the top half of the *nCompare* frame, as illustrated in the following figure:

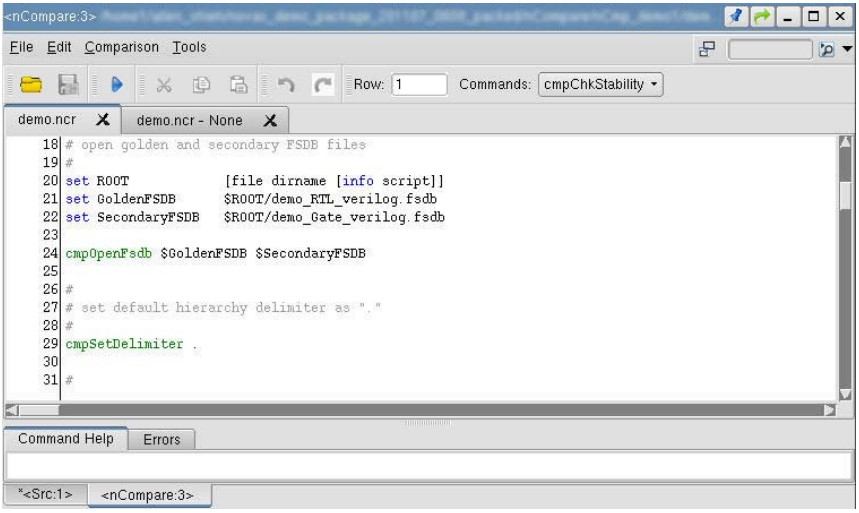


Figure: Current Opened Rule File

7. To start the waveform comparison, choose **Comparison -> Run** command or click the **Run** icon on the toolbar of the *nCompare* frame to start the waveform comparison.

8. To pause or stop the comparison process during the comparing process, choose **Comparison -> Pause** or **Comparison -> Stop** commands. Alternatively, click the **Pause** or **Stop** icons on the toolbar.

View Errors

Following are three main methods for viewing the errors: time view, hierarchy view, or in the waveform.

Sorted by Time

After the waveform comparison is completed, the *nCompare* frame shows the mismatches on the result window and sorts them by design hierarchy or mismatch time sequence depending on the view setting.

To view the mismatches by time sequence, choose **View -> View By Time** command.

The following figure shows the *nCompare* frame with the results sorted by time sequence.

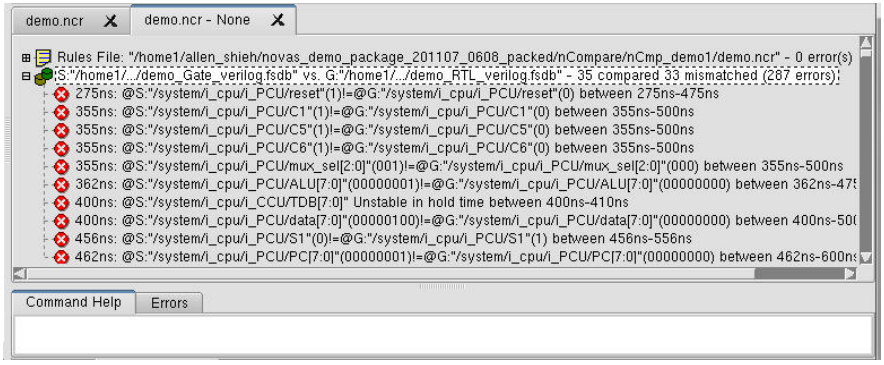


Figure: nCompare Results Sorted by Time Sequence

Sorted by Hierarchy

Alternatively, to view the mismatches by design hierarchy, choose **View -> View By Hierarchy** command. The following figure shows the *nCompare* frame with the results sorted by design hierarchy.

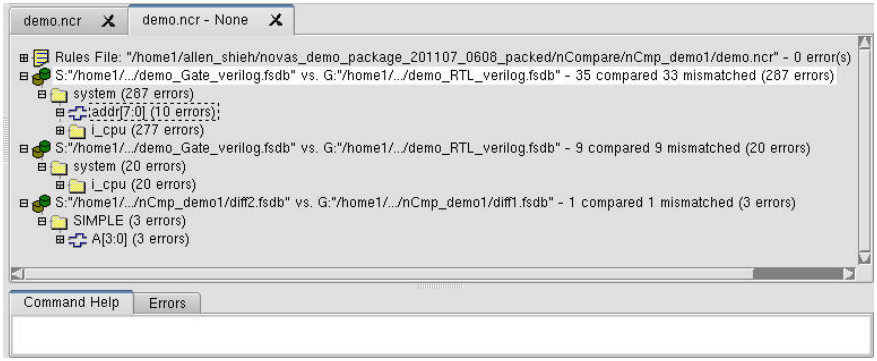


Figure: nCompare Results Sorted by Design Hierarchy

In the Waveform

The *nCompare* frame is tightly integrated with the *nWave* window for viewing mismatches. Perform the following steps to view the errors in *nWave*:

1. Confirm the **View -> View By Hierarchy** command is enabled and then click the + in the hierarchy view of the *nCompare* frame to view the mismatches.
2. Double-click a mismatch error node to add the compared signals into the *nWave* window (the *nWave* window is opened automatically if it is not opened). The cursor time in the *nWave* window is changed to the mismatch time. The result is similar to the following figure.

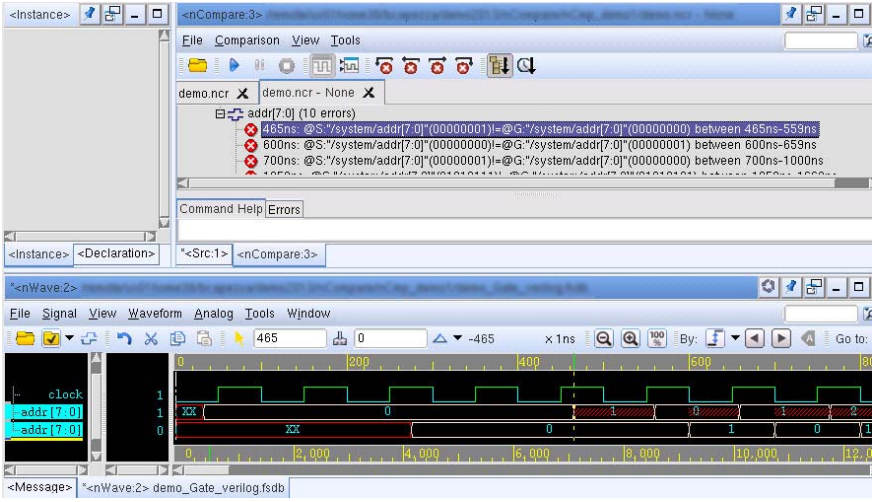


Figure: View Errors With the nWave Window

Error Report File

Save the Current Error File

The error report can be saved to a file for debug at a later time. To save the error file, perform the following steps:

1. In the *nCompare* frame, choose **File -> Save Error** command.
2. Type *error.nce* in the file name text field in the *Save Error As* form as shown in the following figure:

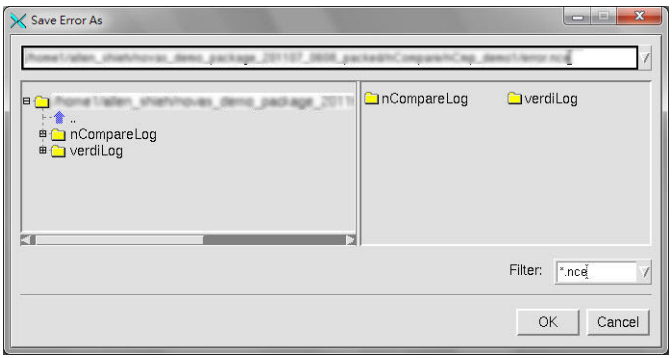


Figure: Saving the Error Report File

3. Click **OK**.

NOTE: The recommendation is to save the error file with the *.nce extension.

Load the Previous Error File

An error report file that is previously saved can be loaded for additional debug. To load the previously saved error report file, perform the following steps:

1. In the *nCompare* frame, choose **File -> Open Error File** command.
2. In the *Open Error File* form, select the *error.nce* error file.
3. Click **OK**.

   The mismatch error report is displayed in the *nCompare* frame by time sequence (the default view). Errors can be viewed in the *nCompare* frame by following the steps in the *View Errors* section.

Application Tutorials

Searching Backward for Value Causes

When you trace the root cause of the signal value by the flow views, the values on the active fan-in signals are compared with the fan-out signal. If there are matching values, then they are taken as new root nodes and the operation continues. This allows you to track the first occurrence of a value of interest using a single command.

Assume that from your testbench, the *alubuf_out* signal in instance *i_ALUB* follows the sequence *0*, *1*, *2*, *3*, *4*, *5* commencing at time 0. However, it actually follows the sequence *0, 1, 2, 3, aa, 4, 5*. You need to find out the reason and the location from where the *aa* value is generated.

Before you begin this application, follow the instructions in the *Before You Begin* chapter.

Open a Temporal Flow View

1. Change your context to the *verdi_mixed* sub-directory, where all the demo source code files are located:

   ```
   % cd <working_dir>/demo/verdi_mixed
   ```

2. Compile the mixed design to create a work.lib++:

   ```
   % ./compile.verdi
   ```

3. Start the Verdi platform by referencing the compiled design and the FSDB file *CPUsystem.fsdb* (contains a set of simulation results) on the command line:

   ```
   % verdi -lib work -top tb_cpusystem -ssf CPUsystem.fsdb
   -workMode -hardwareDebug &
   ```

4. Resize the *nWave* frame to a viewable size and locate it under the source code frame on your screen such that you can see both frames.

The advanced drag-and-drop technology in Verdi platform allows you to quickly locate and display information related to selected objects.

5. In the **Instance** tab of the design browser frame, click the plus symbol to the left of the *i_cpusystem* block instance name to reveal its *i_cpu* and *i_pram* sub-blocks.
6. Double-click the *i_cpu* block instance name in the design browser frame to access the source code of this module, which is displayed in the source code frame and reveals its *i_ALUB*, *i_CCU*, and *i_PCU* sub-blocks.
7. Double-click *i_ALUB* to access the source code of this module.
8. Enter *AluBuf* in the **Find String** text box on the toolbar, and press <Enter> to find the signal.

NOTE: Names are case sensitive.

---

9. Use the middle mouse button to drag *AluBuf* from the source code frame, and drop in the signal pane on the left-hand side of the main *nWave* frame.
10. Right-click *alubuf* in *nWave*'s waveform pane on the transition from *3* to *aa* at time 825, and choose the **Temporal Flow View -> New Temporal Flow View** command.
11. The *Temporal Flow View* frame is created as a new tab in the same location as the *nWave* frame.

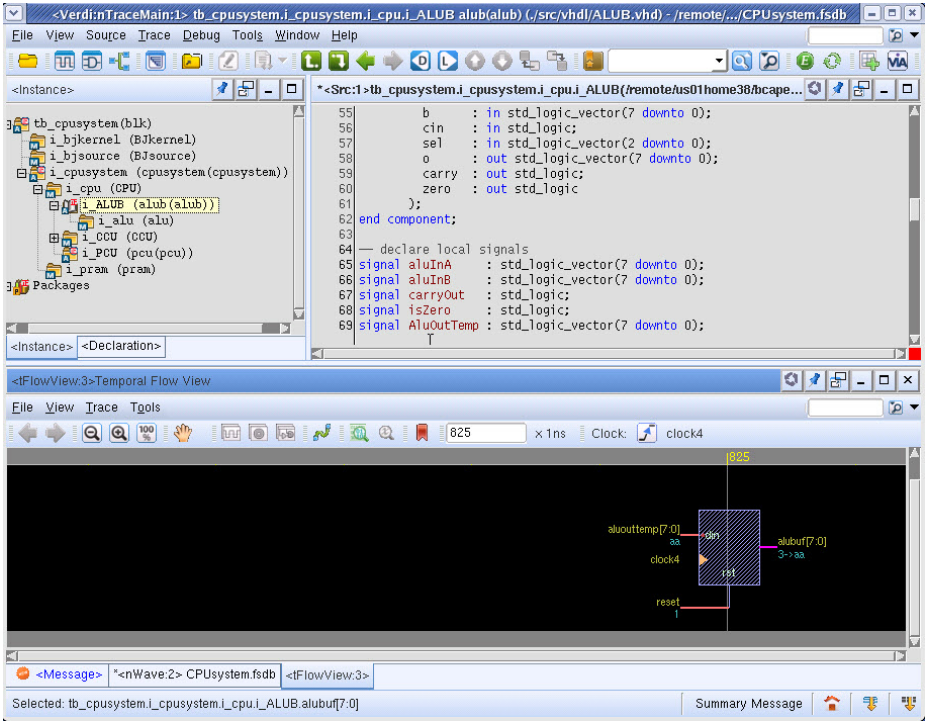


Figure: Temporal Flow View Frame

Show Active Statements

An active statement is the logic statement that generates the value for the driver signals of the current statement.

1. Double-click the *aluouttemp[7:0]* signal in the *Temporal Flow View* frame. You can also right-click the *aluouttemp* node in the *Temporal Flow View* frame, and choose **Show Active Statement**.

   The *Temporal Flow View* frame updates, as shown below:

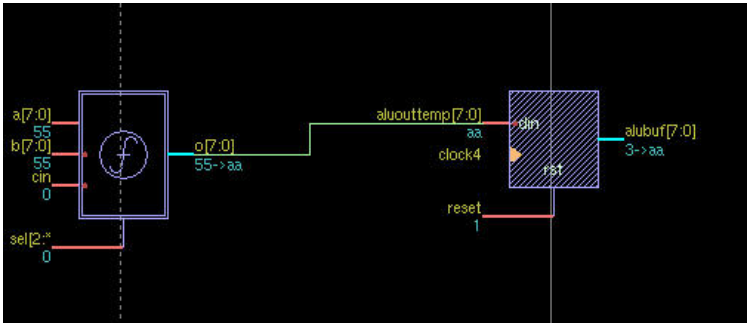


Figure: Temporal Flow View Frame Displaying Active Statements

   Another symbol (a function block) is added to the *Temporal Flow View* window. You can continue tracing active statements on any input node.

Trace This Value Automatically

After you have traced back one statement and seen the function symbol, assume that you want to know where the *55* value on signal *a[7:0]* at time 800 comes from. You can use the **Trace This Value** command, which compares values on the active fan-ins with that of the fan-out signal. If matched values are found, they will be taken as new root nodes and the operation continues. This allows the first occurrence of a value of interest to be tracked down using a single command.

1. Choose the **Tools -> Preferences** command to open the *Preferences* form and set all the options for the **Trace This Value** command.
2. In the *Preferences* form, select the **Cycle Based** page under the **Temporal Flow View -> Trace** folder.

   The form is similar to the following:

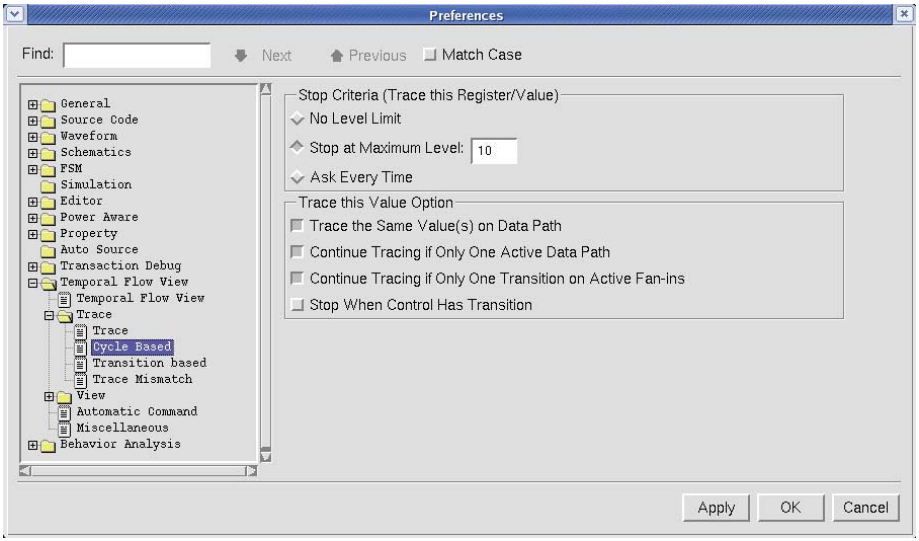


Figure: Preferences Form - Trace This Value Options

You have four options to customize the **Trace This Value** command:

- **Trace the Same Value(s) on Data Path**: If matched values are found among the active data path, they are taken as new root nodes and the trace continues.
- **Continue Tracing if Only One Active Data Path**: If there is only single active data fan-in, it is taken as a new root node and the operation continues.
- **Continue Tracing if Only One Transition on Active Fan-ins:** If there is only one active fan-in with transition, it is taken as a new root node and the operation continues.
- **Stop When Control Has Transition:** If the control signal has a transition, tracing stops.

Confirm that the first three options are enabled and the last option is disabled.

3. Click the **OK** button on the *Preferences* form.
4. Right-click *a[7:0]* in the *Temporal Flow View* window, and choose **Trace This Value** from the menu.
5. Click the **Fit Time** icon (see left) on the toolbar to see the entire results.

   The *Temporal Flow View* frame updates similar to the following figure:

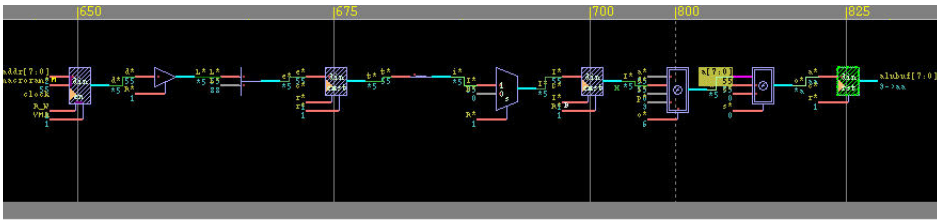


Figure: Trace This Value Results

The **Trace This Value** command traces back several fan-in cones and stops at a memory symbol at time 650.

6. Zoom in around the memory symbol, as shown below:

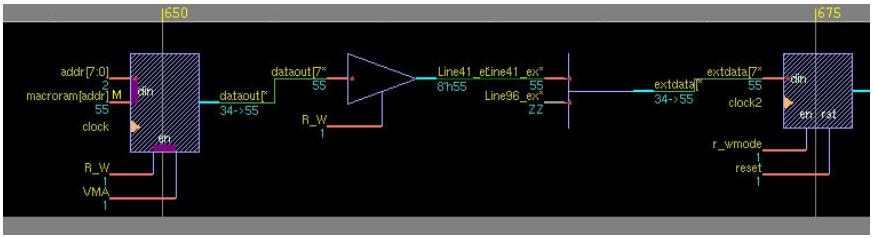


Figure: Memory Symbol

7. Double-click the *macroram[addr]* input node of the memory symbol to find the driving statement.

   The following message displays:

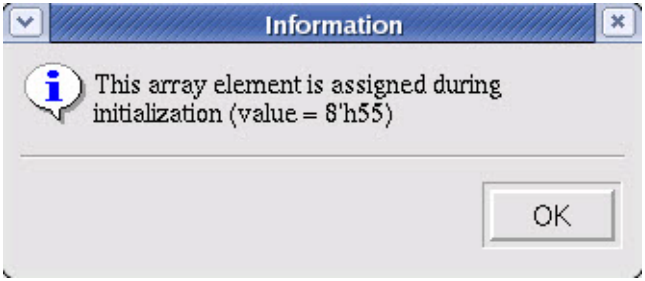


Figure: Information Dialog Window

This message means that the memory content is initialized in the initial block and its value is 55. No further tracing on the *Temporal Flow View* frame is possible. Refer to the *Debug Synthesizeable Memory Models* section for more details.

8. Click the **OK** button on the *Information* dialog window.

You can also complete any of the previous steps using either the *Compact Temporal Flow View* or *Temporal Register View* windows. These views can be opened from the **Tools -> Open Flow View** command, or use the right mouse button and choose **Open Flow View**. The same cause will be located. However, the view will look different.

Trace Another Path

Now that you have traced one path on the *Temporal Flow View* frame, assume you also want to trace the 55 value on *b[7:0]* (where *b[7:0]* is another input node on the function symbol).

1. Right-click *b* in the *Temporal Flow View* window, and select **Trace This Value**.
2. Click the **Fit Time** icon.

   Note that another path will display.
3. Zoom in from time 700-825 to more easily see the details. The frame will be similar to the following:

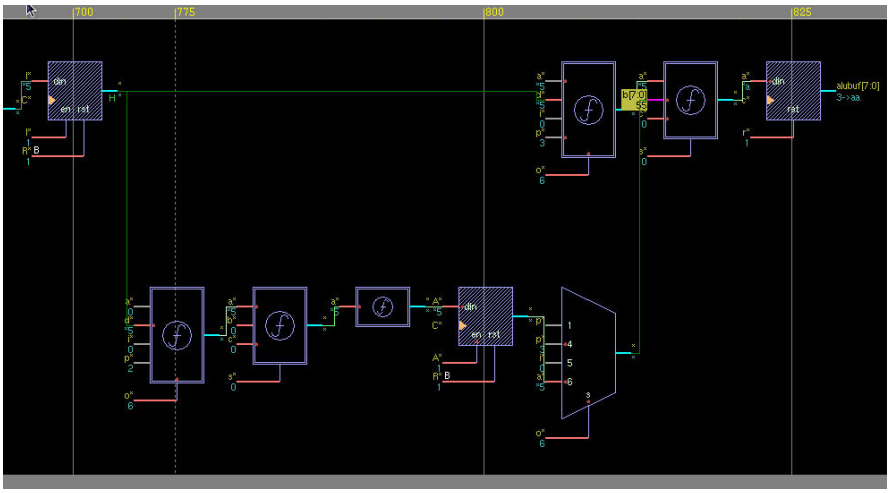


Figure: Trace Multiple Paths

Although *a* and *b* follow different paths initially, at time 700 they come from the same register.

4. Turn on the **View -> Signal -> Active Nodes Only** toggle command to display only the active signals and reduce the clutter.
5. Turn off the **View -> Signal -> Active Nodes Only** toggle command.

Show Signals on *nWave*

At any time, you can select a node and add its waveform to *nWave*. You can also add all fan-in signals for a node.

1. Select the output node *o[7:0]* on the right most function symbol.
2. Use **Ctrl+W** to add its waveform to *nWave*.
3. Use **Ctrl+A** to add all its fan-in signals to *nWave*.

   You can also select a symbol to drag and drop to *nWave*; for example, the register symbol for *AluBuf*.
4. When you are done tracing paths in the *Temporal Flow View* frame and adding signals to *nWave*, choose the **File -> Exit** command in the *nTrace* main window to close the Verdi session.

Debug Memories

The Verdi platform provides a set of features that allow you to debug memories without rewriting your design to note memory contents or adding special tasks to dump memory contents to a file during simulation.

For memory models that can be synthesized, the Verdi platform can calculate memory values dynamically, display memory contents in the waveform display and identify the most recent write time for a specific memory location. All these features are enabled by the Behavior Analysis engine and does not require you to dump the memory contents to a file during simulation.

For memory models that cannot be synthesized, the Verdi platform provides the Memory Definition Table (MDT) template that can be used to define the control signals like write, enable, address and data for a memory. After the memory model is defined through the template, you can view the memory contents, display them in the waveform window, and identify the most recent write time of any address location.

This tutorial illustrates these memory debug features on a simple design.

This section covers the following topics:

- Debug Synthesizeable Memory Models
- Debug Non-synthesizeable Memory Models
- Debug PLI Memory Models

Before you begin this application, follow the instructions in the *Before You Begin* chapter.

Debug Synthesizeable Memory Models

In this example, you will trace the value in an incorrect transition on signal *ACC* back to its source - an instance of a memory model. You will then use the memory debug features to inspect the memory and determine when the value was stored.

Locate the Cause of a Value on a Signal

Enter the following commands to start the tutorial:

1. Change to the demo directory.

   ```
   % cd <working_dir>/demo/verilog/cpu/src
   ```

2. Start the Verdi platform and reference the file *run.f* in the current directory and the FSDB file *CPUsystem.fsdb* in the parent directory:

```
% verdi -f run.f -ssf ../CPUsystem.fsdb
-workMode hardwareDebug &
```

3. Display the waveform for signal *ACC[7:0]* in module instance *i_ALUB* in *nWave*. You can do this by dragging the signal from the source code to the waveform pane or by using the *Get Signals* form in *nWave*.
4. Zoom out in *nWave* until you see *55* at time 801 ns.

In this example, you will locate the cause of a value in the *nWave* frame instead of a *Temporal Flow View* frame. The Behavior Analysis engine is still used -- the results are displayed in a more familiar format. See the *Searching Backward for Value Causes* section for details on using the *Temporal Flow View* window.

5. Click the *ACC* signal in the *nWave* frame somewhere close to the transition from the *00* value to the *55* value and note the following:
   - A vertical cursor appears in the waveform pane.
   - The simulation time of 801 associated with the cursor's current location is displayed in the *nWave* toolbar.

NOTE: By default, the cursor snaps to the closest transition on the selected signal, the transition from *00* to *55* in this case.

---

6. Choose the **Tools -> Preferences** command to set up the preferences for tracing on *nWave*. Then, in the *Preferences* form:
   a. Select the **Automatic Command** page under the **Temporal Flow View** folder.
   b. Turn on the **Highlight Signal Value When Trace on nWave** option.
   c. Click the **OK** button.
7. Right-click *ACC* in *nWave* on the transition from 00 to 55 and choose the **Trace This Value** command to trace the propagation of 55 across multiple cycles.
8. In the *Question* dialog window for Behavior Analysis, click **Yes**.
9. In the *Behavior Analysis* form, click the **OK** button.

   Several signals display in *nWave* in a new group, as shown below:

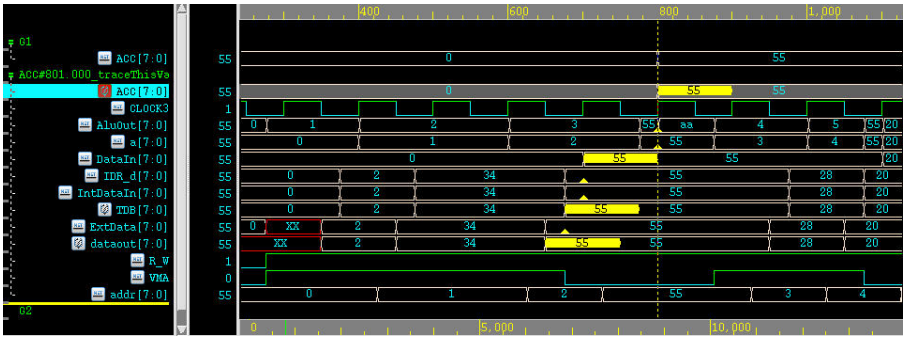


Figure: Display of New Group of Signals in nWave

The Verdi platform traces several cycles back and locates the origin of the *55*, which is signal *dataout*. This signal is the output of a memory. You can confirm this by double-clicking the transition from 34-55 on *dataout* to find the driving statement. In the source code frame, you will see *dataout* assigned by *macroram[addr]*.

In the waveform, you will see that the control signals for the memory were automatically included.

The command stops at the statement that reads memory element *macroram[addr]* into dataout signal. At the time the *addr* signal has a value of 2 and the memory content that is being read out is 55.

The memory content did not get dumped out during simulation. The Verdi Behavior Analysis engine infers the value from the circuit description and simulation dump file.

Locate the Last Write of a Specific Address Location

When a memory output value is wrong, it is usually caused by one of two problems:

- The wrong data was written into the location you are reading.
- You are reading from the wrong address.

First, let's find out when the *55* was written into address *2*:

1. In the *nWave* frame, double-click the transition from 34-55 on *dataout* to find the driving statement.
2. In the source code frame, right-click *macroram[addr*] on line 88 (confirm the cursor is over the *macroram*) and choose the **Debug Memory -> Trace Memory Write** command.

The Verdi platform locates the last write, which was during initialization in this case.

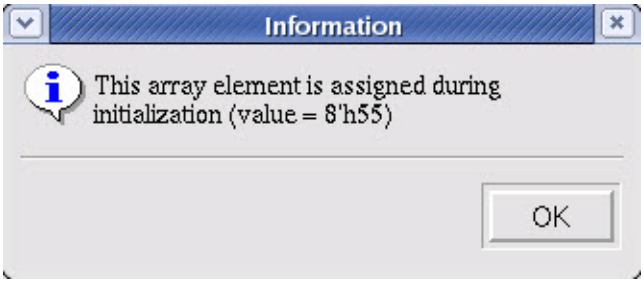


Figure: Question Dialog Window

The Verdi platform analyzes the control logic for the memory, locates the values of these signals in the FSDB file, and determines when the last write to this location occurred.

NOTE: If you had traced the 55 value of *ACC* in the *Temporal Flow View* window, you only need to double-click the memory node to locate the last write.

---

3. Click the **OK** button on the *Information* dialog window.

   The initialization statement for the memory will be displayed in the source code frame.

Now, assume that the initialization file is correct, and you want to know if any other memory locations contain the correct data.

Show Memory Contents

The Behavior Analysis engine can calculate the memory contents at any time you specify:

1. In *nWave*, double-click *dataout* at the 34-55 transition to go to the memory statement again.
2. With *macroram[addr]* selected, right-click and choose the **Debug Memory -> Show Memory Contents** command.

   The *Get Memory Variable* form with the **Calculated by Verdi** tab selected displays, as shown below:

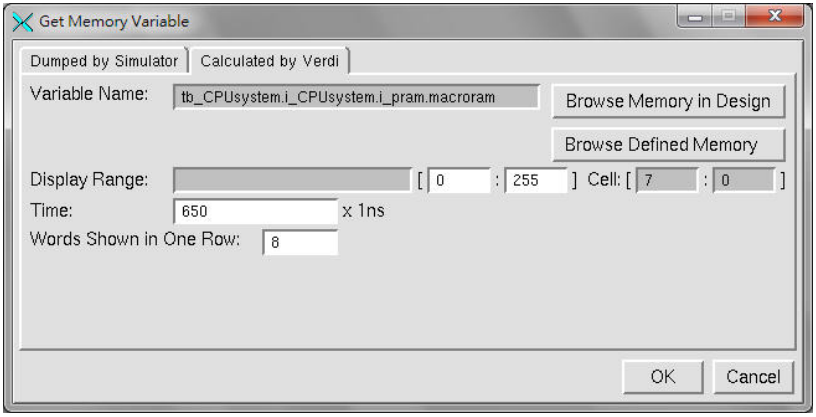
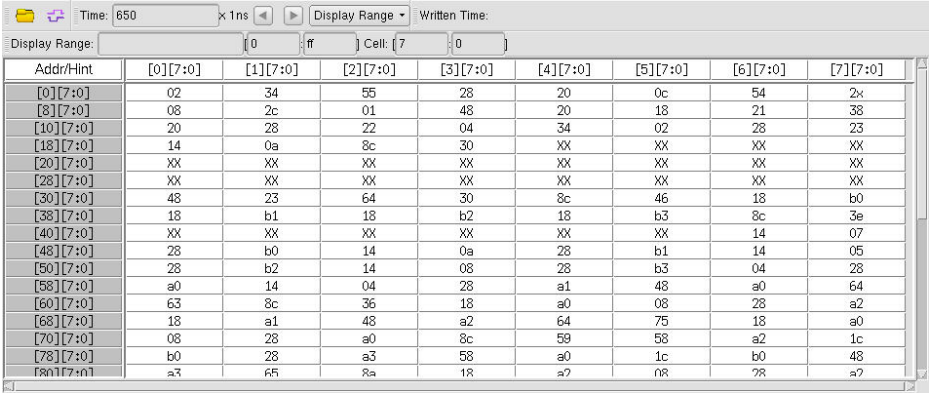


Figure: Get Memory Variable - Calculated by Verdi

The default values of the start and end addresses are the same as the declaration of the memory in the RTL code. The time value is set to the time selected in the waveform (or flow view) by default. In this example, you have specified that you want the Verdi platform to calculate the memory contents from address 0 to address 255 at time 650.

3. Confirm 650 is entered in the **Time** text field and 0:255 is entered in the **Display Range** text fields.
4. Click the **OK** button.

   The memory values are calculated and displayed in the *nMemory* frame (in the lower right of the main window), as shown below:



Addr/Hint	[0][7:0]	[1][7:0]	[2][7:0]	[3][7:0]	[4][7:0]	[5][7:0]	[6][7:0]	[7][7:0]
[0][7:0]	02	34	55	28	20	0c	54	2x
[8][7:0]	08	2c	01	48	20	18	21	38
[10][7:0]	20	28	22	04	34	02	28	23
[18][7:0]	14	0a	8c	30	XX	XX	XX	XX
[20][7:0]	XX	XX	XX	XX	XX	XX	XX	XX
[28][7:0]	XX	XX	XX	XX	XX	XX	XX	XX
[30][7:0]	48	23	64	30	8c	46	18	b0
[38][7:0]	18	b1	18	b2	18	b3	8c	3e
[40][7:0]	XX	XX	XX	XX	XX	XX	14	07
[48][7:0]	28	b0	14	0a	28	b1	14	05
[50][7:0]	28	b2	14	08	28	b3	04	28
[58][7:0]	a0	14	04	28	a1	48	a0	64
[60][7:0]	63	8c	36	18	a0	08	28	a2
[68][7:0]	18	a1	48	a2	64	75	18	a0
[70][7:0]	08	28	a0	8c	59	58	a2	1c
[78][7:0]	b0	28	a3	58	a0	1c	b0	48
[80][7:0]	a3	65	8a	18	a2	08	28	a2

Figure: Show Memory Contents in nMemory

Search Values in the nMemory Frame

1. Choose the **Search -> Find** command to open the *Find* form, as shown below:

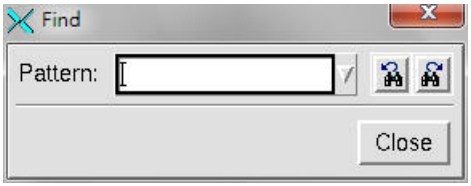
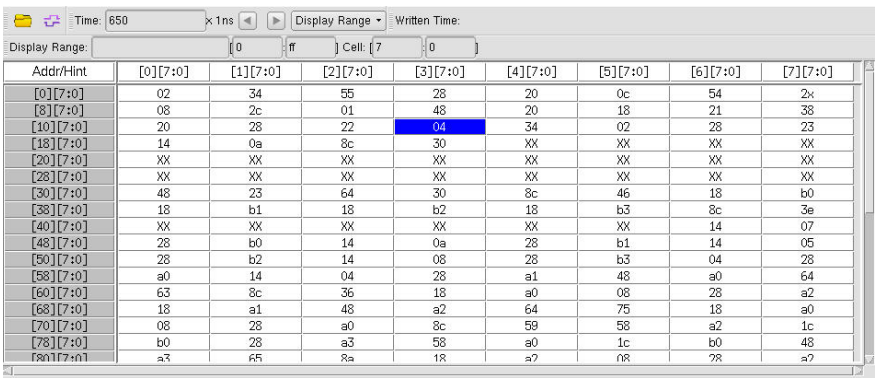


Figure: Find Form

2. Enter a value to search for. For example, 04.
3. Click the **Find Next/Previous** icons to search for the value.

   The desired value is highlighted in the *nMemory* window, as shown below:



Addr/Hint	[0][7:0]	[1][7:0]	[2][7:0]	[3][7:0]	[4][7:0]	[5][7:0]	[6][7:0]	[7][7:0]
[0][7:0]	02	34	55	28	20	0c	54	2x
[8][7:0]	08	2c	01	48	20	18	21	38
[10][7:0]	20	28	22	04	34	02	28	23
[18][7:0]	14	0a	8c	30	XX	XX	XX	XX
[20][7:0]	XX	XX	XX	XX	XX	XX	XX	XX
[28][7:0]	XX	XX	XX	XX	XX	XX	XX	XX
[30][7:0]	48	23	64	30	8c	46	18	b0
[38][7:0]	18	b1	18	b2	18	b3	8c	3e
[40][7:0]	XX	XX	XX	XX	XX	XX	14	07
[48][7:0]	28	b0	14	0a	28	b1	14	05
[50][7:0]	28	b2	14	08	28	b3	04	28
[58][7:0]	a0	14	04	28	a1	48	a0	64
[60][7:0]	63	8c	36	18	a0	08	28	a2
[68][7:0]	18	a1	48	a2	64	75	18	a0
[70][7:0]	08	28	a0	8c	59	58	a2	1c
[78][7:0]	b0	28	a3	58	a0	1c	b0	48
[80][7:0]	a3	65	8a	18	a2	08	28	a2

Figure: nMemory Window with Highlighted Value

4. Click the **Close** button on the *Search Pattern* form.

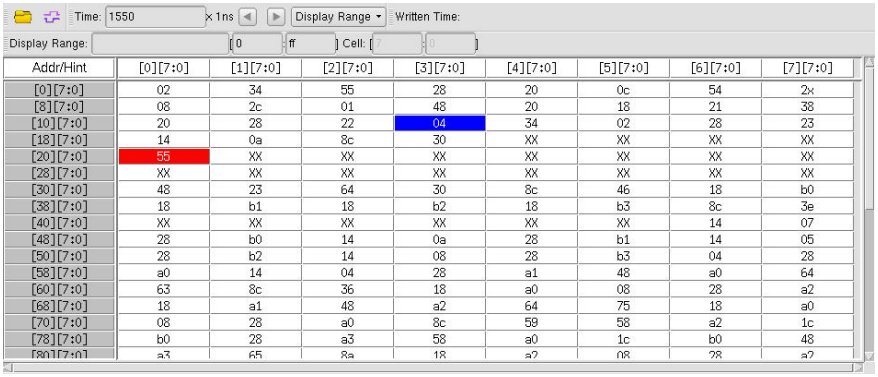
Synchronize the nMemory Frame with nWave

1. Turn on the **Time -> Sync Cursor Time** toggle command to synchronize the time in the *nMemory* frame with *nWave*.

   When you move the cursor in *nWave*, the contents in the *nMemory* frame are updated.

2. Move the cursor in *nWave* to time 1550.

   Note that the value at address *20* changed from *X* to *55*, as shown below:



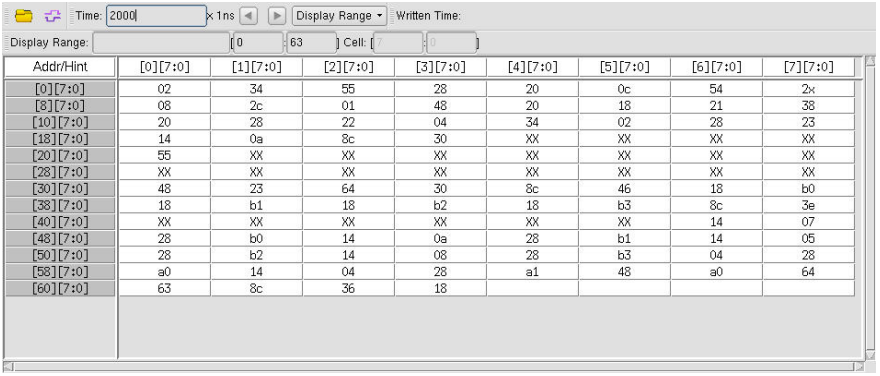
Addr/Hint	[0][7:0]	[1][7:0]	[2][7:0]	[3][7:0]	[4][7:0]	[5][7:0]	[6][7:0]	[7][7:0]
[0][7:0]	02	34	55	28	20	0c	54	2x
[8][7:0]	08	2c	01	48	20	18	21	38
[10][7:0]	20	28	22	04	34	02	28	23
[18][7:0]	14	0a	8c	30	XX	XX	XX	XX
[20][7:0]	55	XX	XX	XX	XX	XX	XX	XX
[28][7:0]	XX	XX	XX	XX	XX	XX	XX	XX
[30][7:0]	48	23	64	30	8c	46	18	b0
[38][7:0]	18	b1	18	b2	18	b3	8c	3e
[40][7:0]	XX	XX	XX	XX	XX	XX	14	07
[48][7:0]	28	b0	14	0a	28	b1	14	05
[50][7:0]	28	b2	14	08	28	b3	04	28
[58][7:0]	a0	14	04	28	a1	48	a0	64
[60][7:0]	63	8c	36	18	a0	08	28	a2
[68][7:0]	18	a1	48	a2	64	75	18	a0
[70][7:0]	08	28	a0	8c	59	58	a2	1c
[78][7:0]	b0	28	a3	58	a0	1c	b0	48
[80][7:0]	a3	65	8a	18	a2	08	28	a2

Figure: nMemory Frame with Synchronized Time

Change Address and Time in the nMemory Frame

1. On the *nMemory* toolbar, change the end address in the **Display Range** field (second text field) to *63* and press Enter on the keyboard.
2. Change **Time** to *2000* and press Enter on the keyboard.

   The *nMemory* frame updates, as shown below:



Addr/Hint	[0][7:0]	[1][7:0]	[2][7:0]	[3][7:0]	[4][7:0]	[5][7:0]	[6][7:0]	[7][7:0]
[0][7:0]	02	34	55	28	20	0c	54	2x
[8][7:0]	08	2c	01	48	20	18	21	38
[10][7:0]	20	28	22	04	34	02	28	23
[18][7:0]	14	0a	8c	30	XX	XX	XX	XX
[20][7:0]	55	XX	XX	XX	XX	XX	XX	XX
[28][7:0]	XX	XX	XX	XX	XX	XX	XX	XX
[30][7:0]	48	23	64	30	8c	46	18	b0
[38][7:0]	18	b1	18	b2	18	b3	8c	3e
[40][7:0]	XX	XX	XX	XX	XX	XX	14	07
[48][7:0]	28	b0	14	0a	28	b1	14	05
[50][7:0]	28	b2	14	08	28	b3	04	28
[58][7:0]	a0	14	04	28	a1	48	a0	64
[60][7:0]	63	8c	36	18				

Figure: Reduced Address Range in nMemory

Customize the nMemory Window

1. Choose the **Options -> Preferences** command to customize the *nMemory* display.

   The *Preferences* form displays, as shown below:

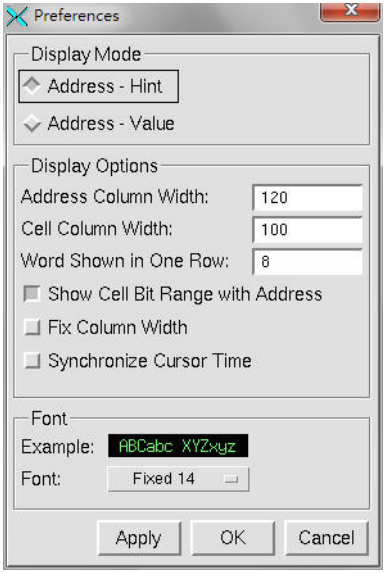


Figure: nMemory Preferences Form

2. Enter *4* in the **Words Shown in One Row** text field.and *30* in the **Cell Column Width** text box and disable the **Show Cell Bit Range with Address** option.
3. Click the **OK** button. The *nMemory* frame re-displays with the new options:

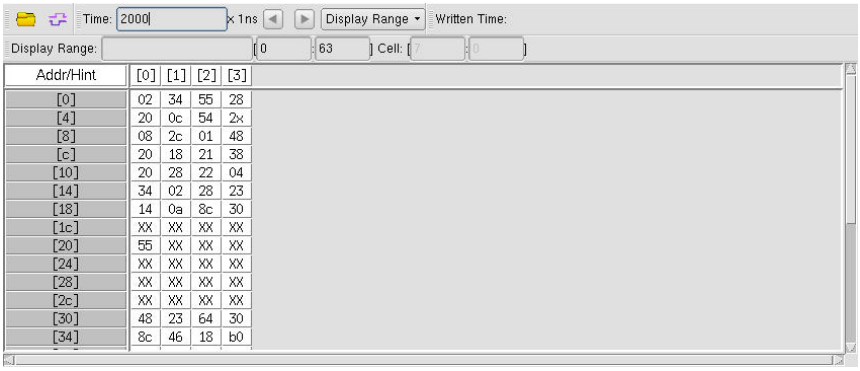


Figure: nMemory Frame with New Display Options

Display Calculated Memory Contents in nWave

1. In the source code frame, right-click the *macroram[addr]* memory variable, and choose the **Debug Memory -> Dump Memory Waveform to FSDB** command to open the *Dump Memory Waveform To FSDB* form.

2. In the *Dump Memory Waveform To FSDB* form, specify the start (0) and end (255) address, the start (0) and end (2501) time, and the name of the FSDB file that you want to write to. Use the values shown in the following figure:

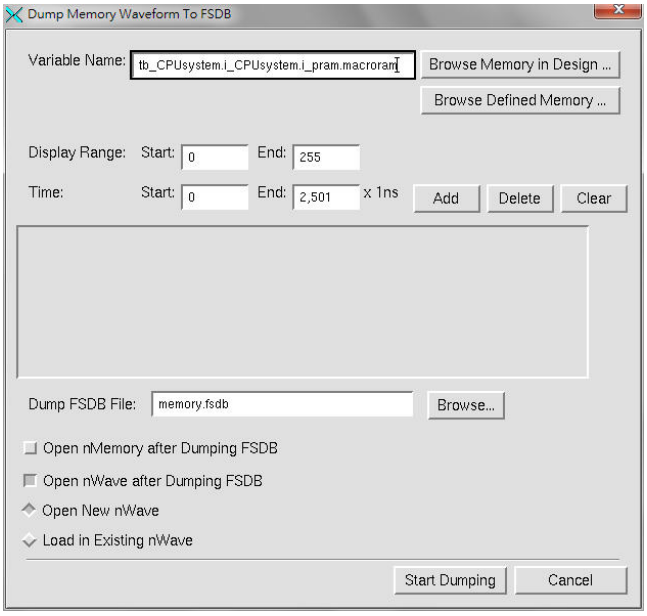


Figure: Dump Memory Waveform to FSDB Form

3. Click the **Add** button.
4. Click the **Start Dumping** button to create the FSDB file and display a new *nWave* frame with the FSDB file loaded

   [Optional] You can also load the new FSDB file into the original *nWave* frame. Choose the **File -> Open** command and select the appropriate FSDB file.

5. In *nWave*, choose **Get Signals** to select the memory to display in the waveform.

NOTE: In the **Find Signal** box in the *Get Signals* form, you can specify which memory range to add. For example, entering *macroram[255:250]* will only list those address elements for that selection.

---

6. In the *nTrace* main window, choose the **File -> Exit** command to close the Verdi session.

Debug Non-synthesizeable Memory Models

Non-synthesizeable memory models are those HDL models described using non-synthesizeable constructs. Since the Verdi platform cannot infer the read/write operation from these HDL models, the memory content cannot be extracted automatically. The Verdi platform provides a way for you to enter the read/write operation manually to extract memory content.

In this tutorial, two static RAM models are used to illustrate the usage:

- 1-port Static RAM
- Multiple-port Static RAM

These examples will show you how to describe the Verdi templates for memory model for non-synthesizeable one or multiple-port memories. After you have described the templates, you can use the Verdi platform to trace memory contents and dump memory content to FSDB.

1-port Static RAM

Enter the following commands to start the tutorial:

1. Change to the demo directory:

   ```
   % cd <working_dir>/demo/verilog/memory_demo/1port
   ```

2. Invoke the Verdi platform using a command file:

   ```
   % verdi –play demo.cmd
   ```

   This loads the design and FSDB, and opens the GUI.

Create a Memory Model Definition for the 1-port Static RAM

1. In the source code frame, select *mem[addr]* in hierarchy *tb.s0*.

NOTE: This should automatically be highlighted by the command file.

The following figure shows the concurrent block that describes the write operation to the 2-D array.

```
 98 always @( posedge clk )
 99   if ( strb ) begin
100     if (~rw_) begin
101       temp_data = mem[addr];
102       if ( !be_[3] )  temp_data[ 31: 24] = din[ 31: 24];
103       if ( !be_[2] )  temp_data[ 23: 16] = din[ 23: 16];
104       if ( !be_[1] )  temp_data[ 15:  8] = din[ 15:  8];
105       if ( !be_[0] )  temp_data[  7:  0] = din[  7:  0];
106       mem[addr] = temp_data;
107     end
108     else begin
109       dout <= mem[addr];
110       @( posedge clk )
111       dout <= 32'hz;
112     end
113   end
```

Figure: Source Code Frame

2. Right-click *mem[addr]*, and choose the **Debug Memory -> Memory Definition Table** command.

   The *Memory Definition Table* form displays. Confirm the default **Module** name is *sram* and the default **Array** is *mem*.

3. In the *Memory Definition Table* form, click the **Add** button to add the memory module name to the list.
4. In the **Operation List** section, toggle to select **Write**, and click the **Add** button. The *Memory Definition Table Editing Window* displays as below.

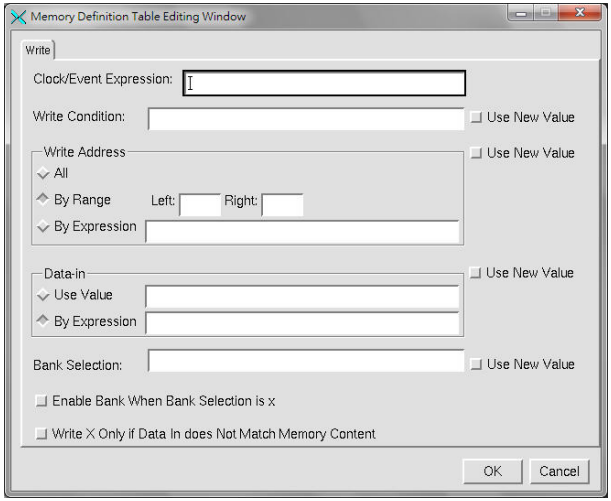


Figure: Memory Definition Table Editing Window - Write Operation

5. Enter the following, or drag and drop from *nTrace* to this table:
   - Clock/Event Expression: *@(posedge clk)*
   - Write Condition: *strb && ~rw_*

- Write Address: select the **By Expression** option and enter *addr*
- Data In: select the **By Expression** option and enter *temp_data*

The **Use New Value** option specifies that the new value should be used for the calculation if there is a value change on *temp_data* at the clock expression; for example, the positive edge of *clk*. Because *temp_data* is assigned and used within the same "*always*" statement, you need to make sure the Verdi platform uses the new values if there are any value changes on *temp_data* at the clock positive edge.

6. Turn on the **Use New Value** option for the **Data In** field.
7. Click the **OK** button in the *Memory Definition Table Editing Window* form.

   This definition is saved, and the list is shown in the **Memory Definition Table - Operation List**.

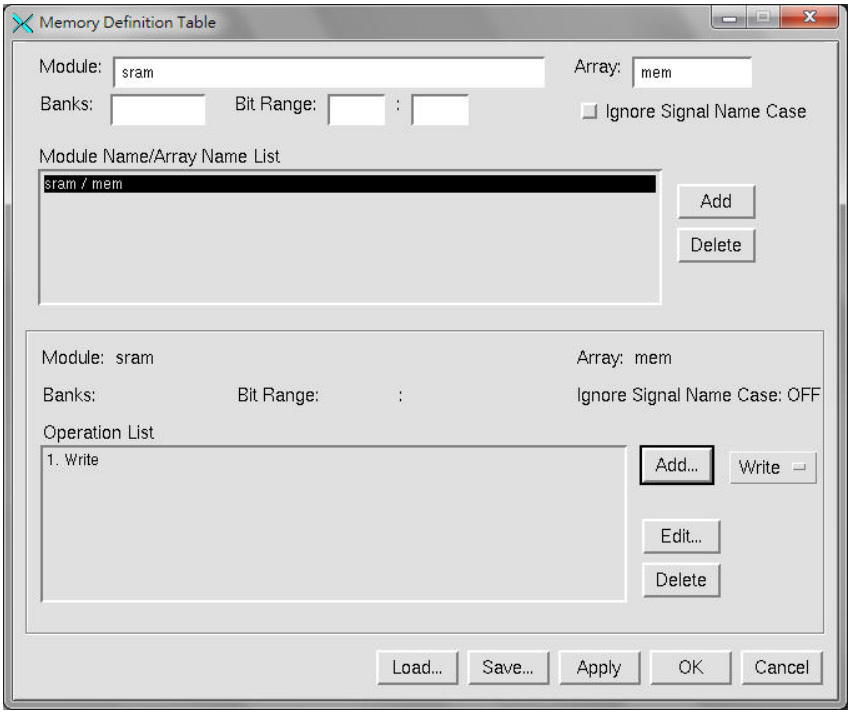


Figure: Memory Definition Table Form

8. [Optional] click the **Save** button to save the definition to a file for future use.
9. Click the **OK** button to close the window and load the memory definition in the Verdi platform.

   You can use the Verdi memory debug features on this one-port static RAM.

Trace the Memory Contents

1. In the source code frame, select the 2D-array signal *mem[addr]*.
2. Right-click *mem[addr]* and choose the **Debug Memory -> Show Memory Contents** command and click **Yes** the on the *Question* dialog window to perform Behavior Analysis.
3. In the *Behavior Analysis* form, click **OK**.
4. The *Get Memory Variable* form opens.

   Suppose you want to see the memory content for element 0 to 63 at time 80000.

5. On the **Calculated by Verdi** tab of the *Get Memory Variable* form, enter 80000 in the **Time** text field, and click the **OK** button.

   An *nMemory* frame displays the specified memory, as shown below:

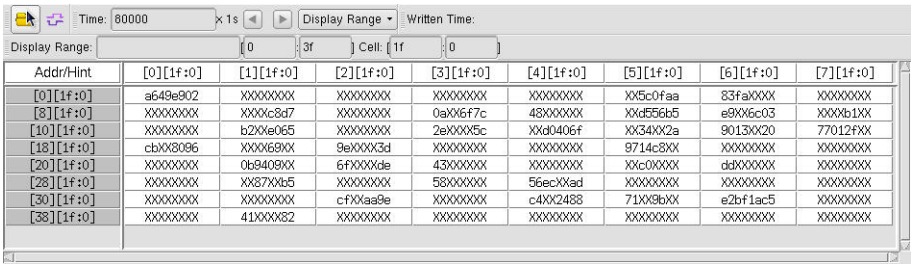


Figure: nMemory Frame Displaying Memory Content

6. Click the value for any address element.

   The Verdi platform will report when the memory element was written on the toolbar.

Display the Memory Contents in *nWave*

1. In the source code frame, select the 2D-array signal *mem[addr]*.
2. Right-click *mem[addr],* and choose the **Debug Memory -> Dump Memory Waveform to FSDB** command.
3. Set the following in the table:
   - Start Display Range: *0*
   - End Display Range: *63*
   - Start Time: *0*
   - End Time: *80000*
4. Click the **Add** button and then the **Start Dumping** button.

The Verdi platform will display a new *nWave* window loaded with *memory.fsdb*, which contains the calculated memory contents.

5. In the new *nWave* window, choose the **Signal -> Get All Signals** command to display the memory, click **OK** on the *Confirmation* question box.
6. In *nWave*, double-click the memory signal to expand and show every address of the memory.
7. Zoom out until you can see values on *mem[0]*. Note that *mem[0]* changes to "*a649_e902*" at time 44450, as shown in the figure below:

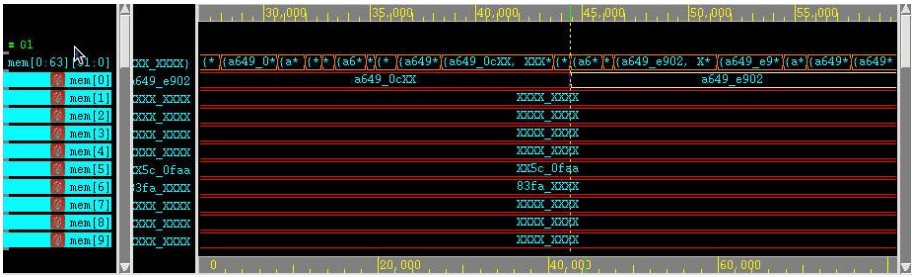


Figure: nWave Frame Displaying Memory Contents

To verify that the memory location *0* has a write operation, check the memory control logic.

8. Move the new *nWave* frame to a docking position above the original *nWave* frame and arrange the two waveform frames so that you can see both. You can hide the *nMemory* frame or other frames if necessary.

   For example rearrange the frame as shown in the following figure.

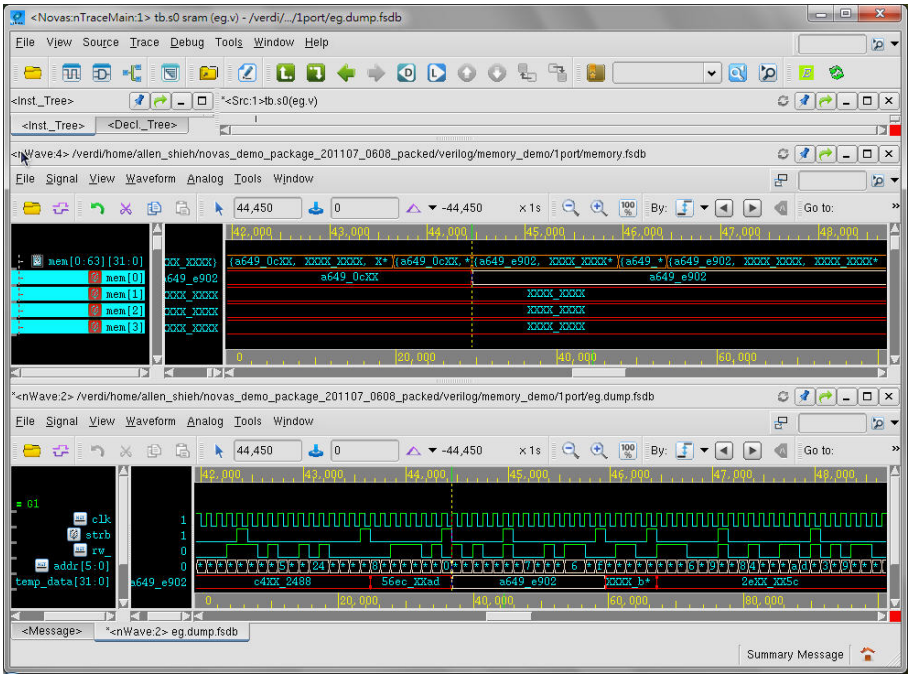


Figure: Rearrange to See Both nWave Frames

9. In the *nWave* frame that is loaded with *memory.fsdb*, choose the **Window -> Sync Waveform View** command to synchronize both waveform frames.
10. Click *mem[0]* at the transition "*a649_e902*" at time 44450. This will also change the cursor time in the other frame to 44450.
11. Locate the control signals *strb* and *rw_* in the waveform. You can see that they are performing a "write" operation at this time and the address is *0*.

[Optional] You can also load the new FSDB file into the original *nWave* window. Choose the **File -> Open** command, and select the appropriate FSDB file.

12. In the *nTrace* main window, choose the **File -> Exit** command to close the Verdi session.

Multiple-port Static RAM

Enter the following commands to start the tutorial:

1. Change to the demo directory:

   ```
   % cd <working_dir>/demo/verilog/memory_demo/mport
   ```

2. Start the Verdi platform:

   ```
   % verdi -play demo.cmd
   ```

Create a Memory Model Definition for the Multiple-port Static RAM

1. In the **Instance** tab of the design browser frame, select hierarchy *tb.s0*.

   The following figure shows the concurrent block that describes the write operation to the 2-D array.

```
62
63  reg [7:0] mem[63:0];
64
65  assign #5 dout = (~ce_ & ~pe3_)? mem[addr_o]: 8'bzzzzzzzz;
66
67  always @(posedge clk)
68  begin
69    if(~ce_ && ~pe1_) begin
70       #3 mem[addr1] = din1;
71    end
72    else if (~ce_ && ~pe2_) begin
73       #3 mem[addr2] = din2;
74    end
75  end
76
```

Figure: nTrace Code Excerpt

2. Select *mem[addr1],* and then choose the **Tools -> Memory Definition Table** command to create the model.
3. In the *Memory Definition Table* form, enter *sram* in the **Module** text box, *mem* in the **Array** text box.
4. Click the **Add** button to add the memory to the list.
5. In the **Operation List** section, toggle to the **Write** condition, and click the **Add** button.
6. In the *Memory Definition Table Editing Window* form, enter the following, or drag and drop from *nTrace* to this table:
   - Clock/Event Expression: *@(posedge clk)*
   - Write Condition: *~ce_&&~pe1_*
   - Write Address: select the **By Expression** option and enter *addr1*
   - Data In: select the **By Expression** option and enter *din1*
7. Click the **OK** button.
8. In the **Operation List** section, click the **Add** button again to create a second write operation.
9. In the *Memory Definition Table Editing* form, enter the following:
   - Clock/Event Expression: *@(posedge clk)*
   - Write Condition: *~ce_ && ~pe2_*
   - Write Address: select the **By Expression** option and enter *addr2*

- Data In: select the **By Expression** option and enter *din2*

10. Click the **OK** button in the *Memory Definition Editing* form.
11. Click the **OK** button on the *Memory Definition Table* form.

    The template is created, and you can calculate and display the memory contents as previously described.

12. In the *nTrace* main window, choose the **File -> Exit** command to close the Verdi session.

Debug PLI Memory Models

The Verdi platform can help to trace the content of PLI memory models. The following shows an example of how it works in the Verdi platform.

Create a PLI Memory Definition File

Enter the following commands to start the tutorial.

1. Change to the demo directory:

   ```
   % cd <working_dir>/demo/verilog/memory_demo/PLI_memory
   ```

2. Invoke the Verdi platform:

   ```
   % verdi -play demo.cmd
   ```

   This loads the design, and the FSDB file.

3. Double-click the instance *tb.s0* in the design hierarchy frame to display the associated source code.

   The module defines the PLI tasks for defining memory and the timing for read/write into the memory.

```
73 initial
74 begin
75     $damem_declare("mem",0,63,0,31);
76 end
```

Figure: Code Excerpt for Defining a PLI Memory

```
124 always @( strb or rw_ or addr )
125       if( strb && rw_ )
126         $damem_read("mem", addr, dout);
127
```

Figure: Code Excerpt for Reading a PLI Memory

```
108 always @( posedge clk )
109   if ( strb ) begin
110     if (~rw_) begin
111        // temp_data = mem[addr];
112        $damem_read("mem", addr, temp_data);
113        if ( !be_[3] )  temp_data[ 31: 24] = din[ 31: 24];
114        if ( !be_[2] )  temp_data[ 23: 16] = din[ 23: 16];
115        if ( !be_[1] )  temp_data[ 15:  8] = din[ 15:  8];
116        if ( !be_[0] )  temp_data[  7:  0] = din[  7:  0];
117        // mem[addr] = temp_data;
118        $damem_write("mem", addr, temp_data);
119     end
120   end
```

Figure: Code Excerpt for Writing a PLI Memory

The first step is to prepare a PLI memory function definition file. This file tells the Verdi platform what the function is for the PLI declaration and what the function is for memory write.

4. Create a new file that contains the following three lines:

```
API_MEM_DECL $damem_declare(MNAME, ADDR_LEFT, ADDR_RIGHT,
RANGE_LEFT, RANGE_RIGHT);
API_MEM_WRITE $damem_write(MNAME, ADDR, DATAIN);
API_MEM_READ $damem_read(MNAME, ADDR, DATAOUT);
```

5. Save the file as *pli_memory.def*.

   The first line indicates the PLI memory is defined using the *$damem_declare* function. MNAME is a keyword. It means the first parameter of the function is the memory name in the HDL design.

   The second line indicates data is written into the PLI memory using the *$damem_write* function. ADDR is a keyword, which means the second field represents the address. *DATAIN* is a keyword, which means the third field represents the data to be written into the PLI memory.

   The third line indicates data is read from PLI memory using the *$damem_read* function. ADDR is a keyword, which means the second field represents the address. *DATAOUT* is a keyword, which means the third field represents the data to be read from the PLI memory.

Load the PLI Memory Definition File

Perform Behavior Analysis, and load the PLI memory definition file:

1. Choose the **Tools -> Preferences** command, and then select the **Memory Definition** page under the **Behavior Analysis** folder.

2. Enter the PLI memory definition file (pli_memory.def) in the **PLI Memory Definition File** field, as shown in the figure below:

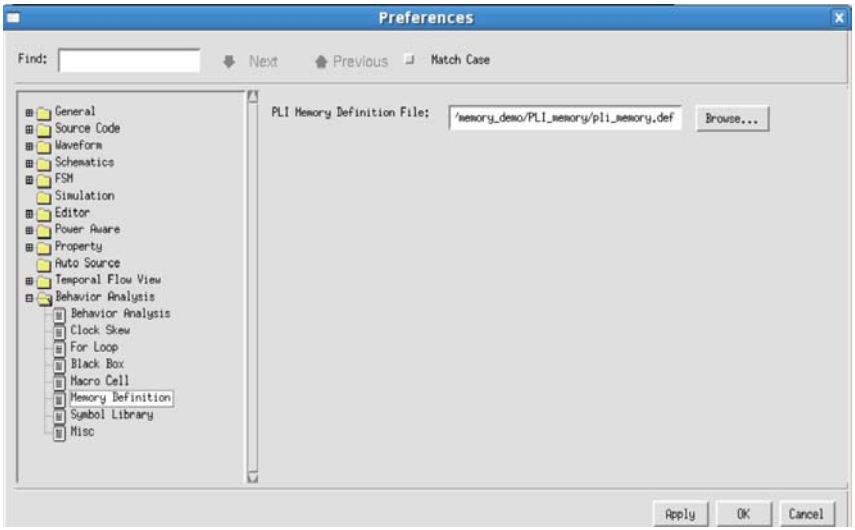


Figure: Load PLI Memory Definition File

3. Click the **OK** button to perform Behavior Analysis.

   The Verdi platform performs analysis on the design and extracts the write conditions for the PLI memory that will be used to trace memory content.

Trace the Content of the PLI Memory

After loading the PLI definition file, you are ready to trace the content of the PLI memory.

1. In the source code frame, locate *mem* in $*damem_declare("mem",0,63,0,31)*; on line 75.
2. Right-click *mem*, and choose the **Debug Memory -> Show Memory Contents** command.

   Although it may look unselectable because the text is black, double-clicking on *mem* will select it.

3. On the **Calculated by Verdi** tab of the *Get Memory Variable* form, enter the address range and the simulation time.

   For example, enter 0 and 63 in the **Display Range** text fields and 50000 in the **Time** text field, as shown below.

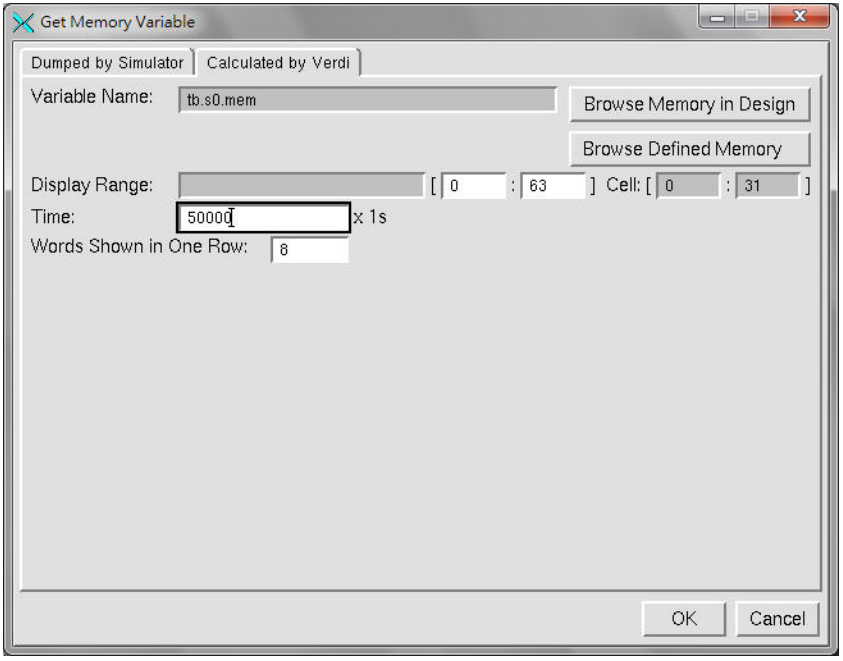


Figure: Get Memory Variable - Calculated by Verdi

4. Click the **OK** button.

   The Verdi platform will analyze the content of the PLI memory from 0 to 63 at time 50000 and display the results in the *nMemory* frame:

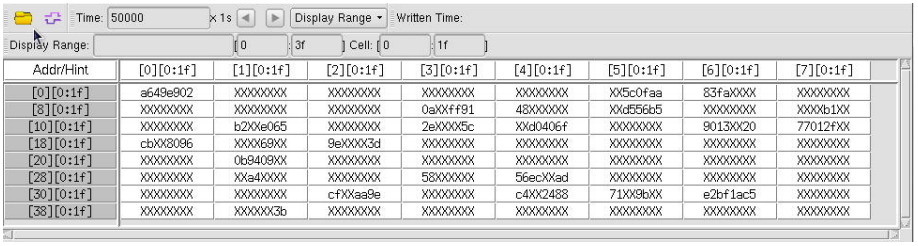


Figure: nMemory Frame for PLI Memory

5. Click any memory element whose value is not X.

   The table will report when the content has been written into the address.

You can also dump PLI memory waveform to FSDB as previously described in 1-port Static RAM.

6. In the *nTrace* main window, choose the **File -> Exit** command to close the Verdi session.

Debug Gate vs. RTL Simulation Mismatch

Before you begin this application, follow the instructions in the *Before You Begin* chapter.

This tutorial will guide you through a scenario explaining how to debug a design if discrepancies occur between RTL and gate-level simulations. In this example, you will debug a mismatch on *carry_flag*.

nWave provides a comprehensive comparison capability to compare simulation results from different simulation runs automatically. *nWave* graphically displays the mismatches in the waveform window after comparison. You can step through each mismatch to analyze the differences.

Locate the Signal to Compare

Typically you must build a gate-level symbol library for your design using the following command:

```
% syn2SymDB synopsys.lib
```

NOTE: *synopsys.lib* is not included in the Verdi package. It is available from other Synopsys tool packages.

For this example, the symbol library is already built and is included in the installation.

1. Set the environment variables for the symbol library using the following commands:

   ```
   % setenv NOVAS_LIBPATHS <VERDI_INST>/share/symlib/32
   % setenv NOVAS_LIBS lsi10k_u
   ```

2. cd to <*working_dir*>*/demo/verilog/gate*.
3. Compile the gate-level design using the following command:

   ```
   % vericom -f run.f
   ```

4. Load the compiled design using the following command:

   ```
   % verdi -lib work -top system -workMode hardwareDebug &
   ```

5. In the main window, choose the **Source -> Find String** command to find *carry_flag* through a string search.

   A *Find String* form displays, as shown below.

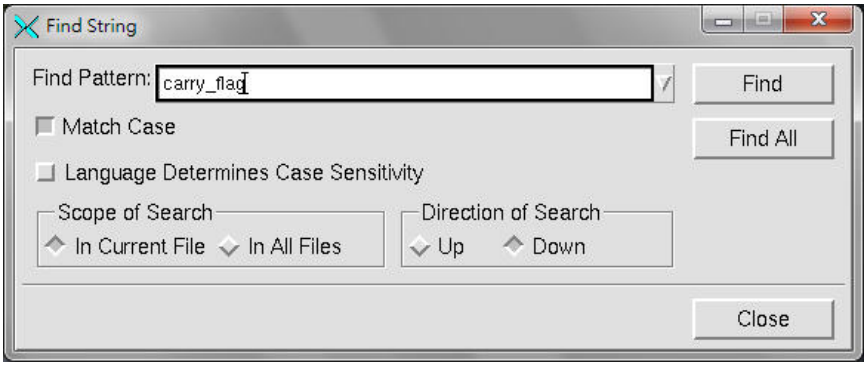


Figure: Find String Form

6. Enter *carry_flag* in the **Find Pattern** text box.
7. Select **In All Files**.
8. Click the **Find All** button.

   In the **Search** tab of the message frame, you will see that *carry_flag* is the output of *carry_flag_reg*.
9. Click the **Close** button on the *Find String* form.
10. In the **Search** tab of the message frame, double-click the driver, *FD2*, which takes you to the corresponding line in the source code.

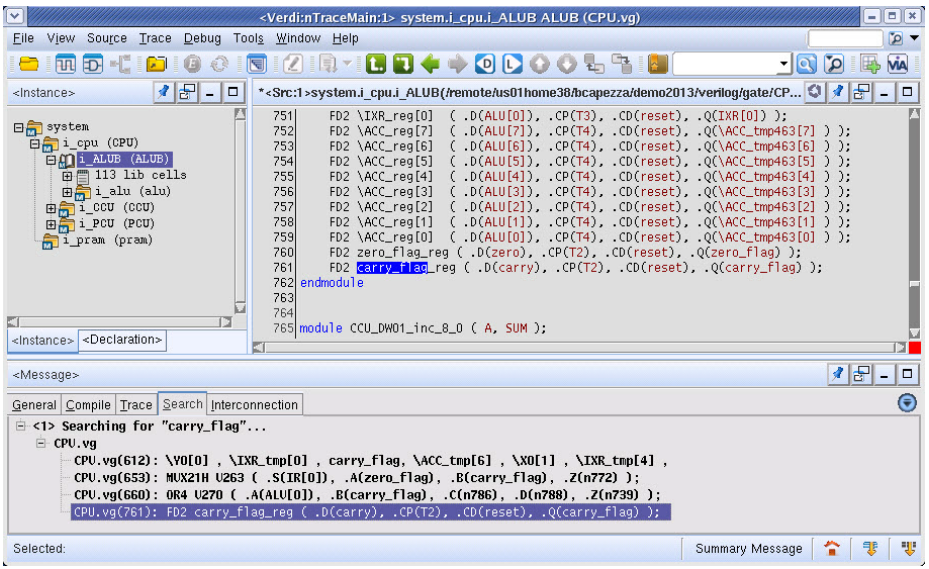


Figure: Find String Results

Load Simulation Results and Display Waveforms

1. From the main window, choose the **Tools -> New Waveform** command or click the **New Waveform** icon on the toolbar open the *nWave* frame.
2. In the *nWave* frame, choose the **File -> Open** command to open the *Open Dump File* form and load the gate-level simulation results.
3. Select *gate.fsdb*.
4. Click the **Add** button and then the **OK** button.
5. In the opened *nWave* frame, choose the **Tools -> New Waveform** command to open another *nWave* frame.
6. From this new *nWave* frame, choose the **File -> Open** command to load the RTL simulation result.
7. In the *Open Dump File* form, select *../rtl/rtl.fsdb*.
8. Click the **Add** button and then the **OK** button.
9. Move the new *nWave* frame to a docking position above the original *nWave* frame and arrange the two waveform frames so that you can see both. For example rearrange the frames as shown in the following figure.

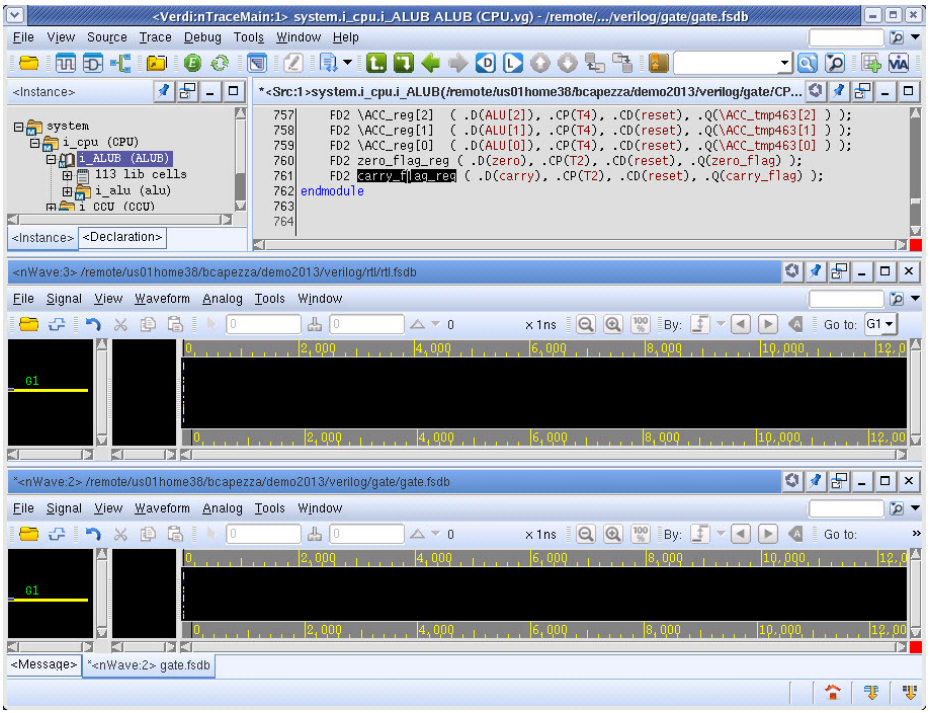


Figure: Compare Two nWave Frames

10. Drag and drop the instance *carry_flag_reg* to both *nWave* frames.

NOTE: You can determine the gate waveform by looking at *nWave*'s title bar for *gate.fsdb*. In the gate-level *nWave* frame, choose the **Window -> Sync Waveform View** command. However, you can tile windows in any *nWave* window.

11. In the both *nWave* frames, choose the **Window -> Sync Waveform View** command to synchronize the viewing based on the simulation time.
12. Use the Pan Left and Pan Right keys (arrow keys on keyboard) to see the effect with synchronized viewing.

    All the viewing operations and cursor and marker positions under one frame are reflected to the other frame except for the Pan Up and Pan Down scrolling.

Compare the Simulation Results

1. Select *carry_flag* in both *nWave* frames.
2. In the gate-level *nWave* frame, choose the **Tools -> Waveform Compare -> Compare Selected Signals** command to compare the simulation results.

   After the comparison is complete, *nWave* displays a dialog window that shows that there is 1 mismatch and the **Search By** toolbar would be changed to **Search By Mismatches**.

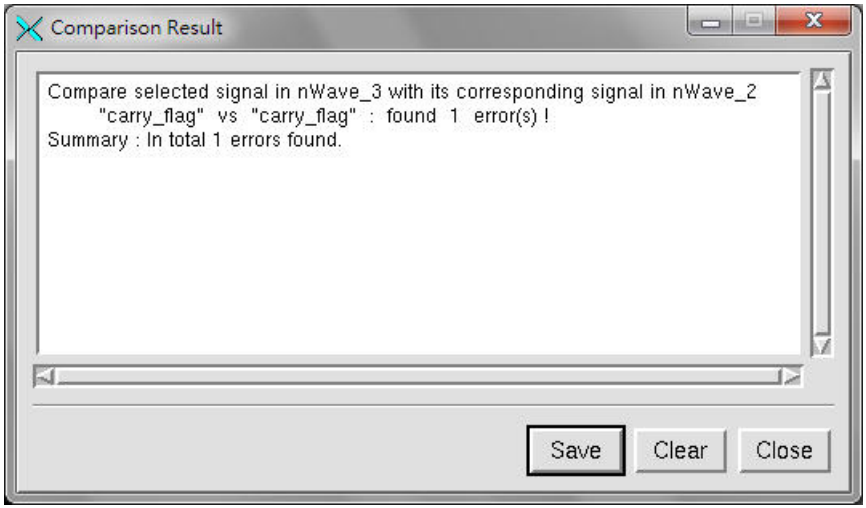


Figure: Comparison Result Message Window

3. On the *Comparison Results* window, click the **Close** button.

4. In the gate-level *nWave* frame, locate the mismatch (indicated by red hatch marks) by clicking the right arrow (**Search Forward**) icon on the toolbar.

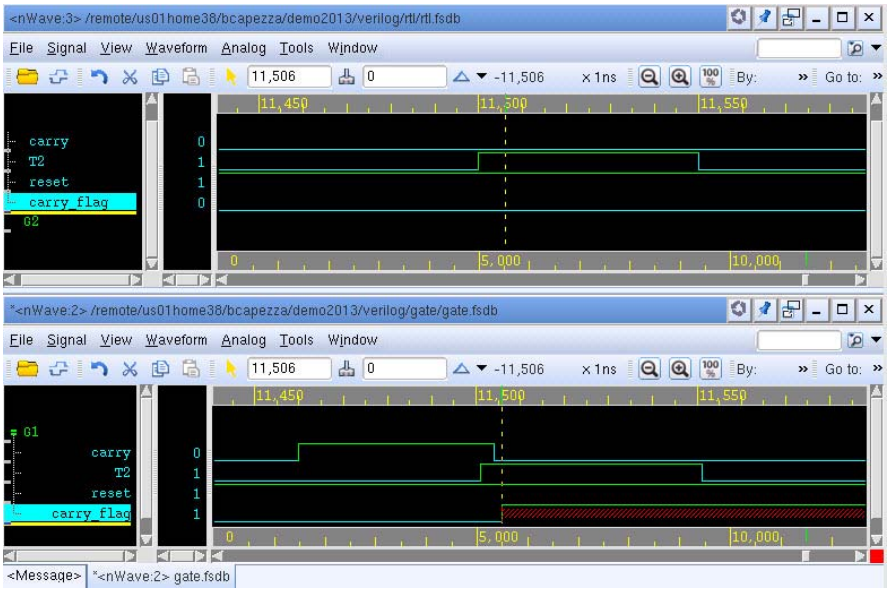


Figure: Mismatch Results in nWave frames

The input (*carry*) to the register in the gate-level design changes too close to the clock edge, thereby causing the mismatch.

Isolate the Problem

1. To find the active driver in the source code frame, double-click the rising edge of *carry* in the gate *nWave* frame.
2. In the *nTrace* main window, choose the **Tools -> New Schematic from Source -> Fan-In** command to generate the fan-in cone for *carry*.

NOTE: It will take a couple of seconds for this to occur due to the fact that the fan-in cone is quite large which makes it difficult to debug.

3. In the gate-level *nWave* frame, select *carry* and put the cursor on the rising edge of *carry* at time 11460.
4. Choose the **Tools -> Active Fan-In** command and specify *10* in the **Back Trace Time Period** field.

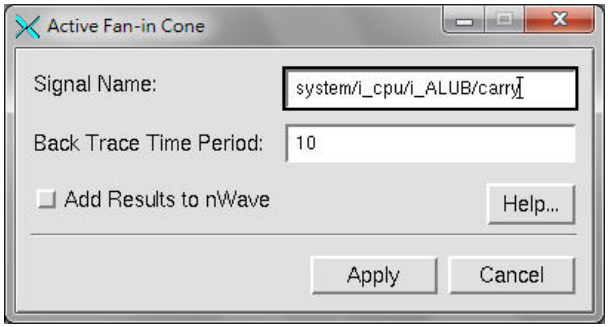


Figure: Active Fan-in Cone Window

5. Click the **Apply** button. The **Fan-in** logic has been reduced to four gates and is now ready for further analysis.

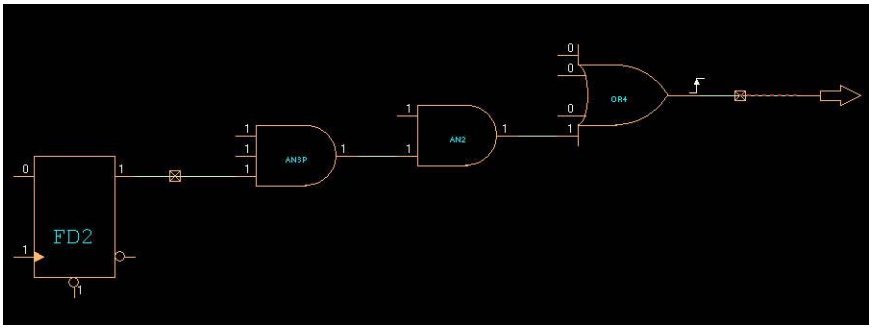


Figure: Active Fan-in Results

6. Exit the Verdi session.

Behavior Trace for Root Cause of Simulation Mismatches

Simulation mismatches between two simulation runs for the same design occur due to any of the following reasons:

- Different optimization levels applied in a simulator.
- Different versions of a simulator or libraries.
- Different compile-time options.
- Different simulators.

Using the Verdi platform, you can locate the cause of this simulation mismatch by tracing back the behavior and comparing the results of two simulation files to identify the root cause of the mismatches.

This feature is different from the *nCompare* module or waveform compare in the Verdi platform in which the goal is to find all the mismatches between two FSDB files.

Before you begin this application, follow the instructions in the *Before You Begin* chapter.

Locate the Simulation Mismatch

Enter the following commands to start the tutorial and then use waveform compare to locate the signal and time that mismatches.

1. Change to the demo directory:

   ```
   % cd <working_dir>/demo/verilog/cpu/src
   ```

2. Start the Verdi platform and reference the file *run.f* in the current directory and the FSDB file *CPUsystem.fsdb* in the parent directory:

   ```
   % verdi -f run.f -ssf ../CPUsystem.fsdb
   -workMode -hardwareDebug &
   ```

3. Display the waveform for *AluBuf* signal from the *ALUB* module in the *nWave* frame by dragging from the source code frame or using **Get Signals**.
4. Open another *nWave* frame by clicking the **New Waveform** icon on the tool bar of the main window.
5. In the new *nWave* frame, choose the **File -> Open** command to load the *../CPUsystem_fix.fsdb* file.
6. In the *Open Dump File* form, click **Add** and then **OK** to complete the load.

7. Move the new *nWave* frame to a docking position above the original *nWave* frame and arrange the two waveform frames so that you can see both. For example, rearrange the frames as shown in the following figure.

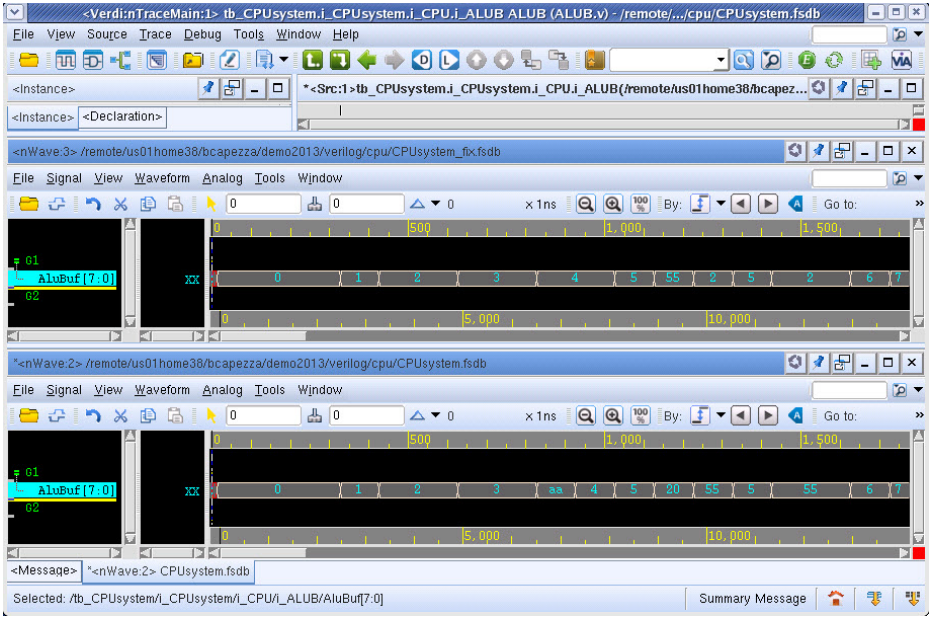


Figure: Compare Two Waveforms

8. Drag and drop the *AluBuf* signal from the first *nWave* frame to the second *nWave* frame.
9. Choose the **Window -> Sync Waveform View** command in both *nWave* frames to synchronize the two *nWave* frames.
10. Select the *AluBuf* signal on both *nWave* frames.
11. In the original *nWave* frame (the one that displays *CPUsystem.fsdb* in the toolbar), choose the **Tools -> Waveform Compare -> Compare 2 signals** command.

    The *Comparison Results* dialog window opens, indicating 11 mismatches, as shown below:

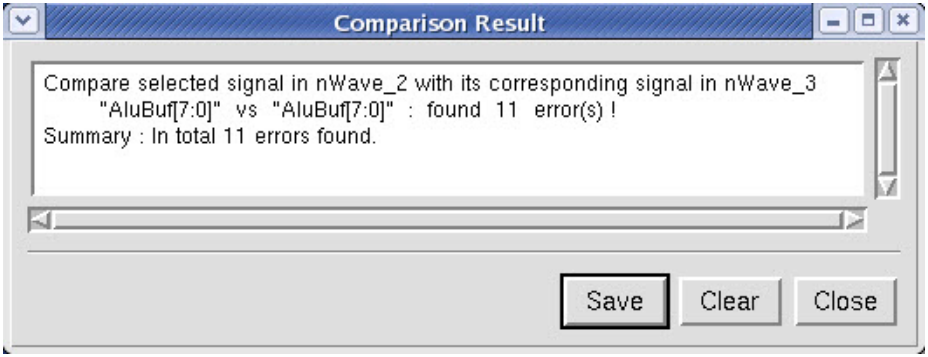


Figure: Comparison Result Window

12. Click the **Close** button on the *Comparison Result* dialog window.
13. Use the **Search Forward** icon in the original *nWave* frame to locate the first mismatch at time 826ns. One has value *aa* and the other is *4*.

    Now that a mismatch point is located, let's find out the cause.
14. Continue with the next section, Behavior Trace for the Root Cause of Mismatch.

Behavior Trace for the Root Cause of Mismatch

1. Click the *AluBuf* signal in the first *nWave* frame at time 826.
2. Right-click, and choose the **Temporal Flow View -> New Temporal Flow View** command to create the *Temporal Flow View* frame.
3. In the *Temporal Flow View* frame, select the **File -> Load 2nd Waveform for Trace Mismatch** command to load the *CPUsystem_fix.fsdb* file.
4. Before tracing the mismatch, choose the **Tools -> Preferences** command to open the *Preferences* form and customize the options.

   In the *Preferences* form, select the **Temporal Flow View** folder **-> Trace** folder **-> Trace Mismatch** page, similar to the following form:

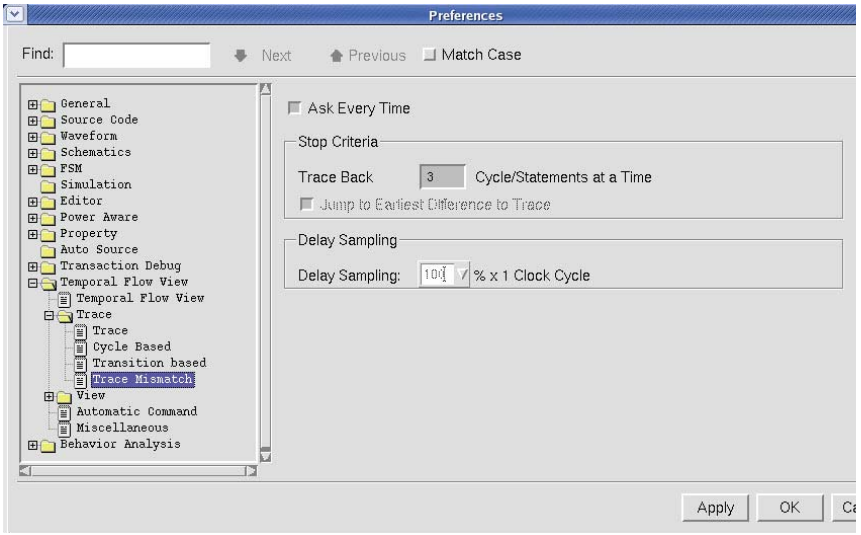


Figure: Preferences Form - Trace Mismatch Options

- **Ask Every Time**: the option form will open every time you start the behavior trace.
- **Trace Back N Cycles/Statements at a Time**: specify how many cycles or statements to trace back and compare.
- **Jump to Earliest Difference to Trace:** for the fan-in signal that has a different value, instead of continuing the same Behavior Analysis process at the time the value is different, jump to the earliest time there is a difference in the FSDB and continue the trace process.

5. In the *Preferences* form, turn off the **Ask Every Time** and **Jump to Earliest Difference to Trace** options and change the **Trace Back** value to *20*.
6. Click the **OK** button to close the form.
7. Right-click *AluBuf* in the *Temporal Flow View* frame, and choose the **Behavior Trace for FSDB Mismatch** command.
8. After the results are displayed, zoom in around time 800-825 by dragging-left on the time display.

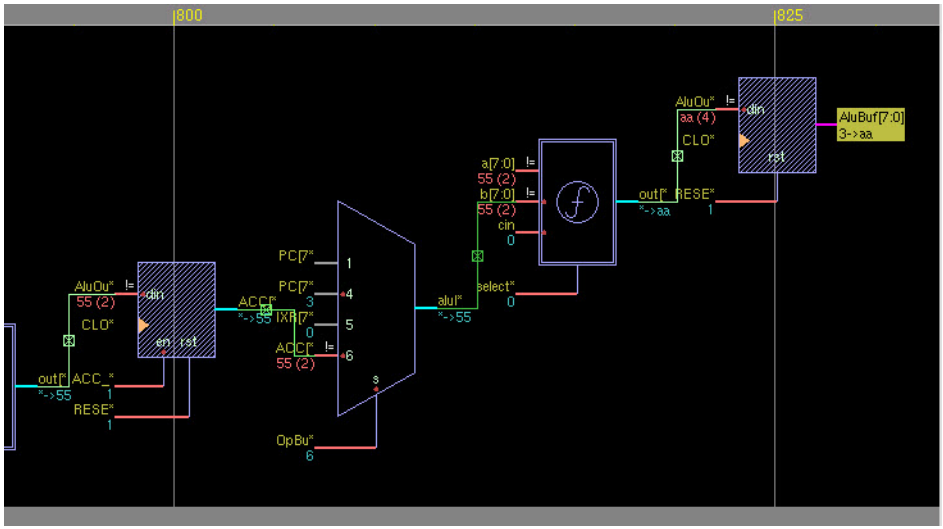


Figure: Behavior Trace Mismatch Results

The value of *AluBuf* in the two FSDB files is different at time 800 as shown in the right-most function symbol. The *b* path is automatically traced. The **Behavior Trace** command works in the following way:

- First the fan-in cone of the selected signal is expanded.
- Then, the Behavior Analysis engine is used to determine what the value is and when it happens for the fan-in signals to produce the value of the selected signal.
- The value and time of these fan-in signals is then used to compare them with those in the second FSDB. A mismatch mark is set on the fan-in signals that have different values.

9. Scroll to the left end and note that there is still a mismatch on the *b* path at time 575. The next element to the left is a memory - it does not have a mismatch.

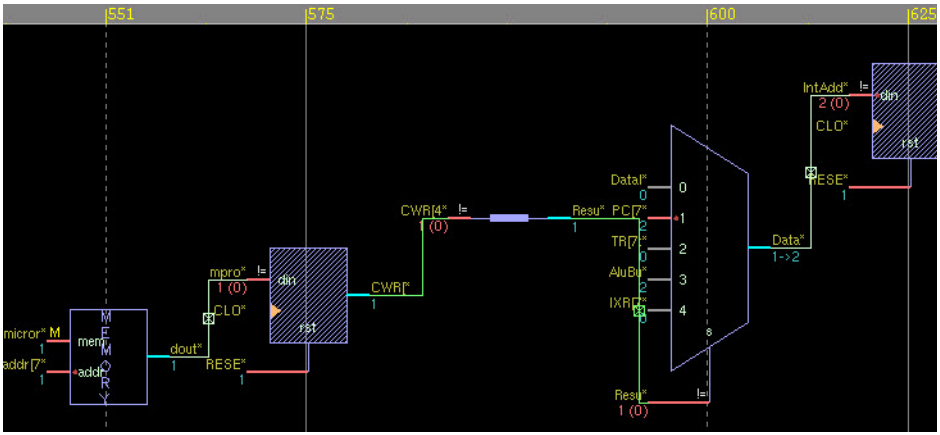


Figure: Memory without Mismatch

10. Double-click the memory input node to find out when it was written.

    It was initialized to the value.

    In this example, a different microrom initialization file was used for each simulation run. This is what produced the different values for *AluBuf* at time 826.

11. In the *nTrace* main window, choose the **File -> Exit** command to close the Verdi session.

Debug Unknown (X) Values

If a signal has an unknown value X, the root cause of the X needs to be found. Using the Verdi platform, you can locate the first X with one command. The Verdi platform will analyze the design and report the possible causes of the X or the path causing the X can be visualized over multiple cycles.

Assume that the signal *ZFout* has an unknown value X at time 2777ns, and you need to find out the root cause of the X. A typical way to debug this problem is to display the signal in the waveform, refer to the source code to find the driving signals, display those signals in the waveform to identify those that are X, and so on. Eventually, and if you are persistent, you will locate the first occurrence of an X.

Before you begin this application, follow the instructions in the *Before You Begin* chapter.

Locate the Root Cause of the “X” Value on ZFout

Enter the following commands to start the tutorial and then use the Behavior Analysis engine to automatically locate the cause of an unknown value with one command.

1. Change to the demo directory:

   ```
   % cd <working_dir>/demo/verilog/cpu/src
   ```

2. Start the Verdi platform, and reference the file *run.f* in the current directory and the FSDB file *CPUsystem.fsdb* in the parent directory:

   ```
   % verdi -f run.f -ssf ../CPUsystem.fsdb
   -workMode hardwareDebug &
   ```

3. In the main window, turn on the **Source -> Active Annotation** toggle command.
4. Display the waveform for *ZFout* from the *ALUB* module in the *nWave* frame.
5. Zoom out in the *nWave* frame until you see “*X*” at time 2777ns. Your task is to locate the cause of this unknown value.

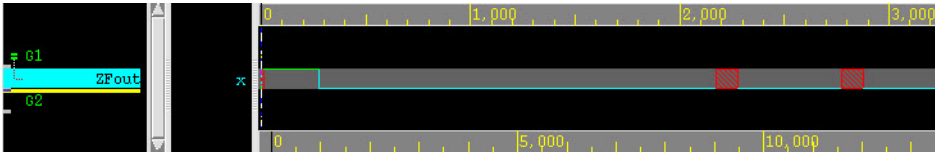


Figure: ZFout with Unknown Value

6. Click the *ZFout* signal in *nWave* somewhere close to the transition from the *0* value to the *X* value, and note the following:
   - A vertical cursor appears in the waveform pane.
   - The simulation time of 2777 associated with the cursor's current location is displayed in *nWave* frame toolbar.

NOTE: By default, the cursor snaps to the closest transition on the selected signal, the transition from *0* to *X* in this case.

7. Right-click the *ZFout* in the *nWave* frame at the transition from *0* to *X*, and select **Trace X**.

   A *Question* dialog window displays, indicating that you need to perform Behavior Analysis.

8. Click the **Yes** button on the *Question* dialog window.
9. In the *Behavior Analysis* form, click the **OK** button.

   The *Trace X Settings* form displays, as shown below:

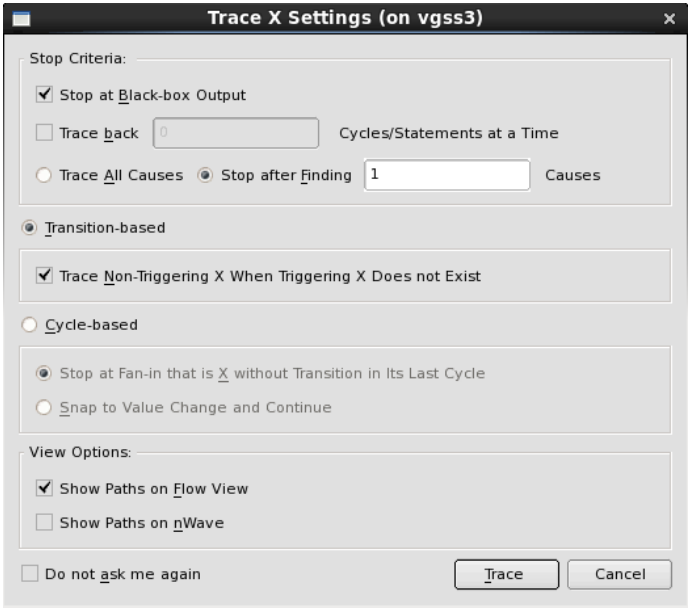


Figure: Trace X Setting Form

The *Trace X Setting* form has several options, including:

- If you select **Stop at Black Box Output**, tracing will stop at these outputs instead of finding the inputs to the black box that are X and continuing the trace from those points.
- The **Stop at Fan-in that is X but No Transition in Its Last Cycle** option instructs the tool to trace back only those fan-in signals that make signal transition.
- The **Snap to Value Change and Continue** option tells the tool to snap to the closest value change point and continue the tracing process.

10. Click the **Trace** button.

    The results will display as a new tab in the same position as the source code frame, as shown in following figure.

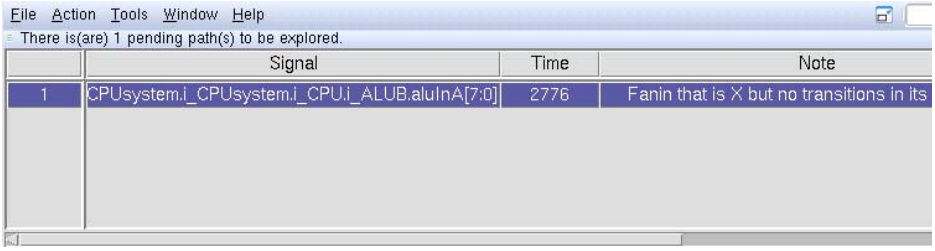


Figure: Trace Triggering X Results Frame

The Verdi platform stops at the signal *aluInA* at time 2776 with the reason that the **Fanin that is X but has no transitions in its last cycle**. The fan-in signal that is X in this case is *DataIn*. This means there is an assignment of X to *DataIn* but this assignment is not the one that causes the value transition in the waveform. This will be investigated later.

The following table describes the possible causes of an unknown value:

Possible X cause	Comment
Fan-in that is X but no transition in its last cycle	Trace-X algorithm stops because all the fan-ins that are X do not change value in the cycle being evaluated. The algorithm stops because the bug may come from the fan-in signals that are not X but have value changes. The cause will only appear when user turns on the option “Stop at fan-in that is X but no transition in its last cycle.”
No fan-in that is X	None of the fan-in signals is X, or the X signal has no fan-in.
X from primary input of the work scope	The trace-X algorithm stops at the input signal of the work scope
Load constant ‘X’	Loads a ‘X’ value into a signal, e.g. s1 = 1’bx

Possible X cause	Comment
Memory net (assigned value ‘x’ during initialization)	The memory element has been assigned a ‘X’ value during initialization
Memory net (not initialized)	The memory element has not been initialized
Black box output	Trace-X algorithm stops at the output of a blackbox. This message appears only when the option “Stop at blackbox output” is turned on.

11. To see the source code for the cause, right-click the *tb_CPUsystem.i_CPUsystem.i_CPU.i_ALUB.aluInA* signal in the *Trace Active X Results* frame, and select the **Show Source Code on nTrace** command. The RTL statement that drives the signal is highlighted on the source code frame, as shown below:

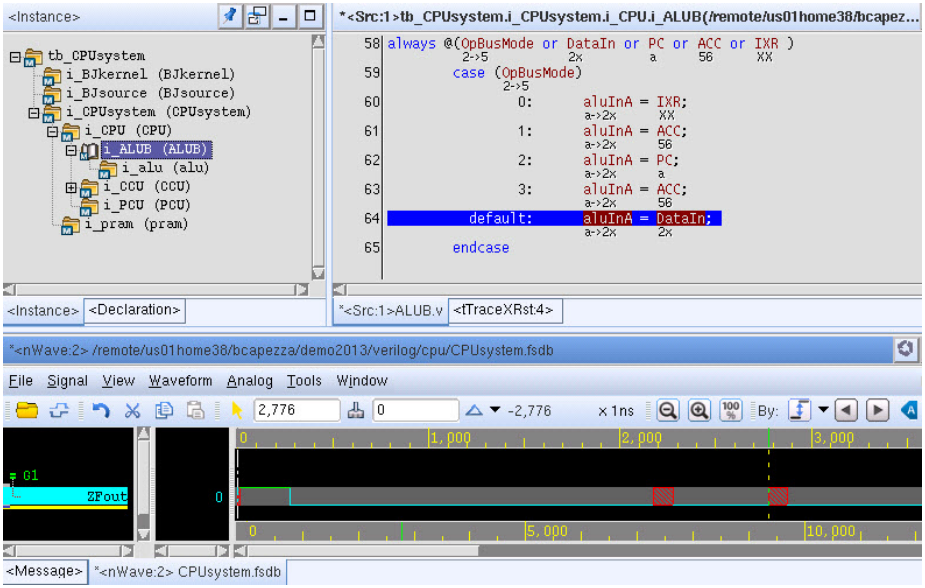


Figure: Source Code for Unknown

12. Right-click the signal in the *Trace Active X Results* frame again, and select the option **Add Active Fan-in Signals to nWave**.

    The active fan-in signals of *aluInA* displays on the *nWave* frame, similar to the figure below:

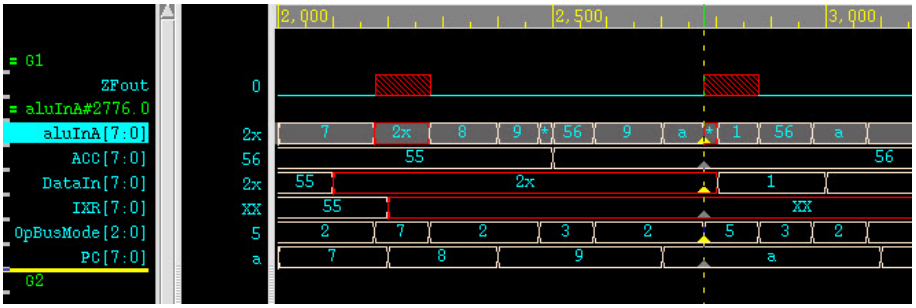


Figure: Active Fan-ins for aluInA

13. To determine if there has been a value change in the last cycle or not, you need to know the clock cycle. In the waveform pane, right-click *DataIn* near the *2x* value, and select **Show Clock (Domain)**.

    The waveform frame displays similar to the following figure:

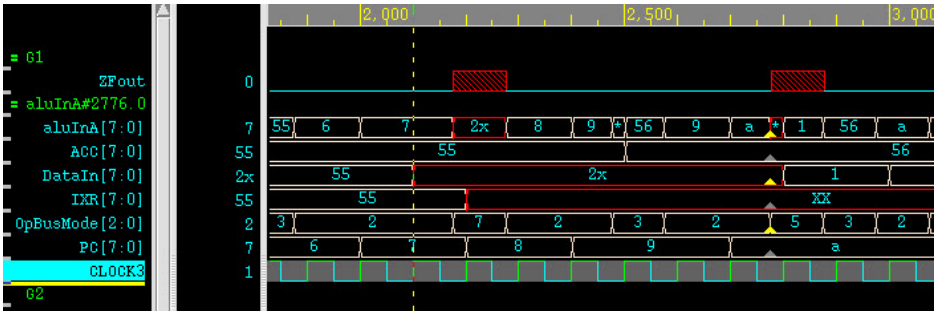


Figure: Clock for DataIn

    You can see the clock for *DataIn* is *CLOCK3*. The value change for the unknown of *DataIn* comes from several cycles before the *ZFout* signal goes unknown. If this is not the cause of unknown, you may continue the trace back process.

14. In the *Trace Active X Results* frame, select the **Action** -> **Continue to Trace Selected Signal** command.
15. Click **Trace** on the *Trace X Setting* form that displays.

    Trace active X will continue on the selected net and will stop at a memory output. The cause for this X is that the memory net has been assigned an unknown value during initialization.

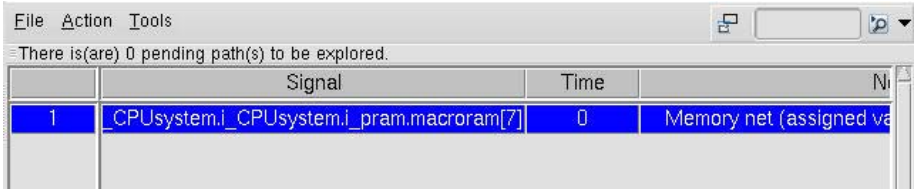


Figure: Trace Active X Results

16. Close the *Trace Active X Results* frame.

Visualize the Active Paths in the *Temporal Flow View*

Another way of tracing active X is to visualize the propagation path on the flow view.

1. Click the *ZFout* signal in the waveform pane of the *nWave* frame somewhere close to the transition to the *X* value at time 2777.
2. Right-click the *ZFout* signal on the transition from *0* to *X*, and choose the **Temporal Flow View** -> **New Temporal Flow View** command. A *Temporal Flow View* frame opens.
3. In the *Temporal Flow View* frame, right-click the *ZFout* signal at time 2777, and select the **Trace X** command.The *Trace Triggering X Setting* form appears.
4. Turn off the **Stop at Fan-in that is X but No Transition in Its Last Cycle** option.
5. Turn on the **Show Paths on Flow View** option.
6. Change the value for **Trace Back N Cycles/Statements at a Time** to 20.
7. Click the **Trace** button.

   An *Information* dialog window opens indicating the memory was assigned during initialization.
8. Click the **OK** button on this dialog window.
9. The *Trace Triggering X Results* frame opens, and the path is traced in the *Temporal Flow View* frame.
10. Click the **Fit Time** icon in the *Temporal Flow View* frame to see the entire path.

    The *Temporal Flow View* frame display updates, as shown below:

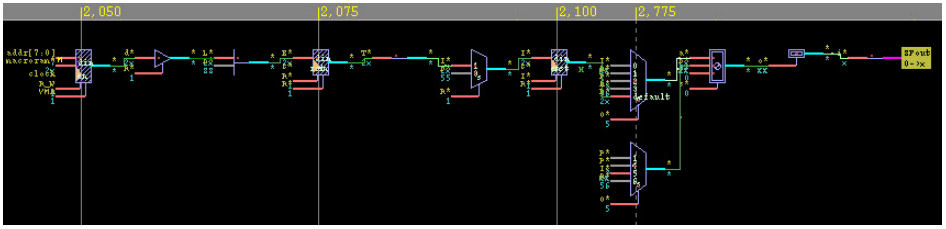


Figure: Trace X Results in Temporal Flow View Window

By default all nodes are displayed. The tracing stops at the memory output *macroram[7]*.

11. Choose the **View -> Signal -> Nodes with Value 'X' Only** toggle command to remove the known nodes from the display.

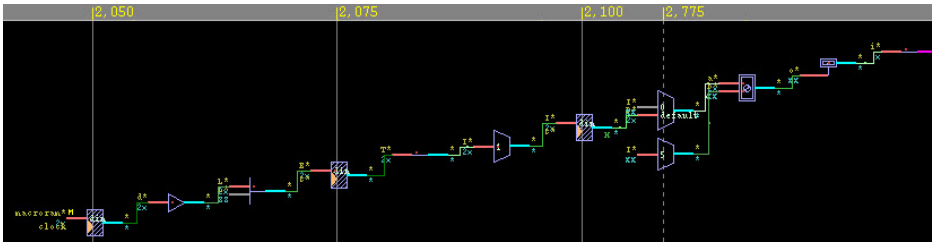


Figure: Unknown Nodes in Temporal Flow View Frame

12. Double-click any node to see its driver in the source code frame.
13. In the *nTrace* main window, choose the **File -> Exit** command to close the Verdi session.

Debug Forced Signals

In Verilog, a "force" statement applied to a variable overrides a procedural assignment, a continuous assignment, or an assign procedural continuous assignment to the variable. The value of the variable is 'forced' until a release procedural statement is executed on the same variable. In addition, the **force** and **release** PLI or UCLI commands can be used to impact variables in interactive simulation debug mode. This makes it difficult for you to know the real driver of the variable just by looking at the source code or waveform.

The Force Debug features provide visibility into the force and release actions. When an FSDB file containing dumped forced information is loaded, you are able to view and debug the forced signal through the annotated source code, schematic and waveform. The force/release statement is also be considered as the root cause during tracing. In addition, the forced, deposited, and released events of the specified signals can be extracted to a report file.

Enable Force Debug

Perform the following steps to enable the Force Debug features.

1. Specify these environment variables to setup your environment and simplify scripting:

   ```
   > setenv VERDI_HOME <VERDI_INST_DIR>
   ```

2. To enable the capability of force debug dumping, the **-debug_access** compile options are required during compile time for the VCS compiler. For example:

   ```
   > vcs -full64 -sverilog -debug_access+all ./design.sv -lca
   ```

NOTE: The current available simulators are VCS version I-2014.03 and later.

NOTE: The **-debug_access** compile options are required to enable force features with VCS for force/release statements coming from an external source (e.g. UCLI commands), not in the source code. The **-debug_access** option is an LCA option in VCS, refer to the VCS documentation for details.

3. There are two methods to enable dumping force/release information into the FSDB file. You can set the environment variable:

   ```
   > setenv FSDB_FORCE 1
   ```

Alternatively, you can use the **+fsdb+force** runtime option during simulation. For example:

```
> ./simv +fsdb+force
```

The dumped force/release information includes the following:

- Signal name
- Time point to be forced
- Force/Release type: deposit or freeze type

Debug in Source Code

Trace Signal

In the source code frame, the results of trace driver (invoked by the **Driver** command in the right-click command menu) for a selected signal include force/release statements. After the FSDB file containing dumped force information is loaded into the Verdi platform, you can use the **Active Trace** command to only show the tracing results with those forced/released events that impact the range of the variable at a specified time. As shown in the following figure, a forced event will be treated as a real driver even when the force event does not cause a value change.

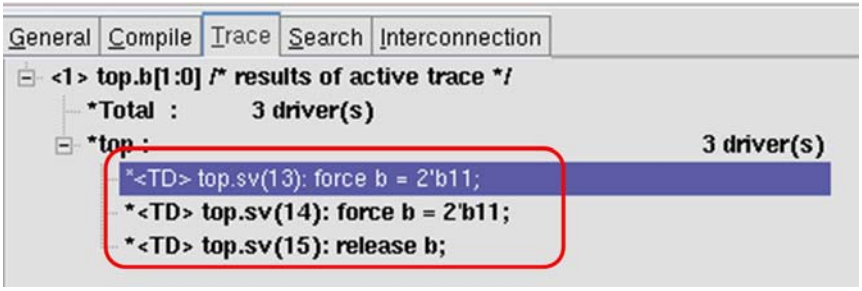


Figure: Force/Release as the Active Trace Result

When the force or release event is caused by UCLI (e.g. use the **force** command in UCLI mode) or PLI commands, the external command will be listed at the end of the original result as another category of the active tracing result.

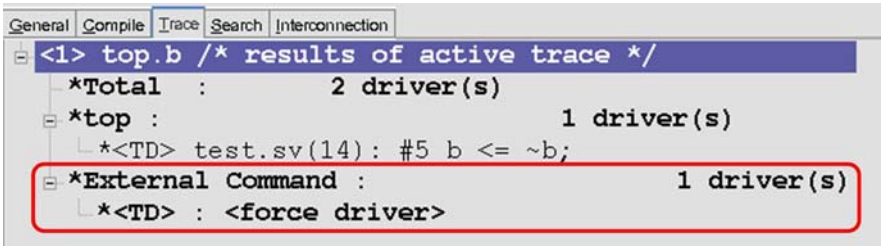


Figure: External Force Command as the Active Trace Result

If the signal is frozen forced, <*force driver*> is shown as the result for the selected signal (refer to the figure above). If the signal is deposit forced, <*deposit driver*> is shown (refer to the figure below). If all bits of the selected signal are released, <*release driver*> will be shown. If partial bits are still forced and partial bits are released for the selected signal, <*force driver*> will be shown.

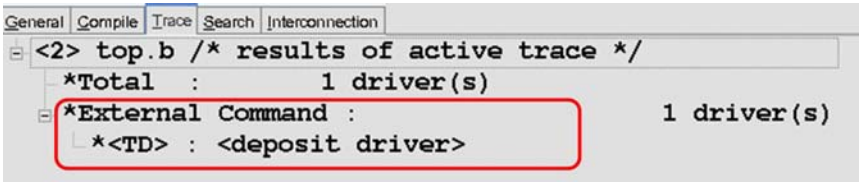


Figure: External Deposit Command as the Active Trace Result

Active Annotation

After the FSDB file containing dumped force information is loaded into the Verdi platform, the annotation value will be displayed with a force mark. In the source code and *Signal List* frames, use the **Source** -> **Active Annotation** command to see the current forced value annotated below the signal with the caret "^" character. As shown in following figures, the caret "^" character is annotated to highlight the force/release event.

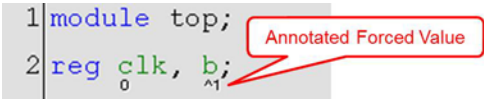


Figure: Active Annotation for Forced Value

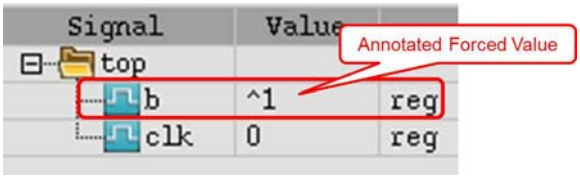


Figure: Forced Value in Signal List

If a value change is caused by a force/release command, the transition will be annotated with values. For example, if the value change from '1' to '0' is caused by a force command, the annotate value will be '*1->^0*' to indicate the value change is caused by a force command, as shown in the following figure.

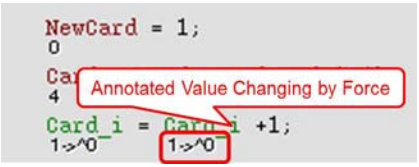


Figure: Value Change with Forced Value

In the *Watch Expressions* frame (invoked by the **Tools** -> **Watch Expressions** command), the forced values are also annotated with the caret "^" character for the variables, as shown in the following figure.

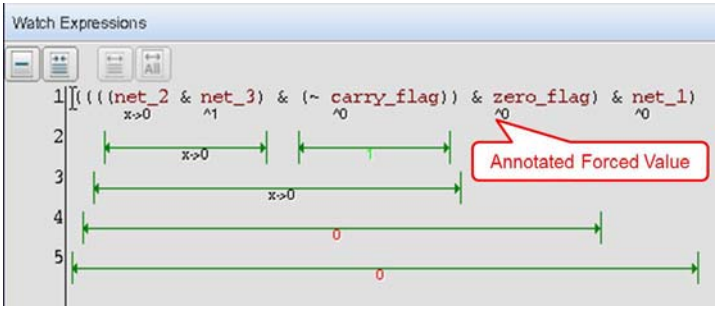


Figure: Forced Value in Watch Expression

NOTE: The expression value in the *Watch Expressions* frame will not show the force information.

Debug in the Waveform View

When an FSDB file containing dumped forced information is loaded into the Verdi platform, the force/release information is also automatically shown in the *nWave* frame with special icons to highlight the force/release events.

Icons in the Waveform Pane

As shown in the following figure, the force, release, and deposit values, and the forced status are shown with corresponding icons in the waveform pane. The value will be annotated with the caret "^" character in the value pane if the current value is forced.

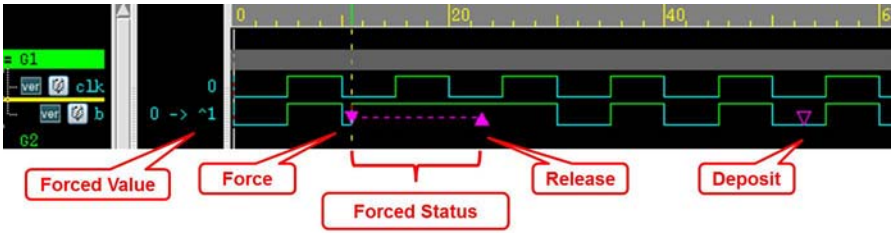


Figure: Forced Value in Watch Expression

The following table shows the attributes of the forced icons in the *nWave* frame.

Icons	Attributions
	The value is forced.
	The value is released.
	The value is deposited.
	The status of the value is forced.

If there is a value change with an icon in the *nWave* frame, you can double-click the icon to trace the force, deposit, or release statement in the main window.

Show Tips

In the waveform pane, you can turn on the **Enable Tip** option in the **Waveform -> View Options -> Waveform Pane -> General** page of the *Preferences* form (invoked with the **Tools -> Preferences** command) to show a tip window when

the cursor is placed over an force/release event. The following figure shows the examples for the tips.

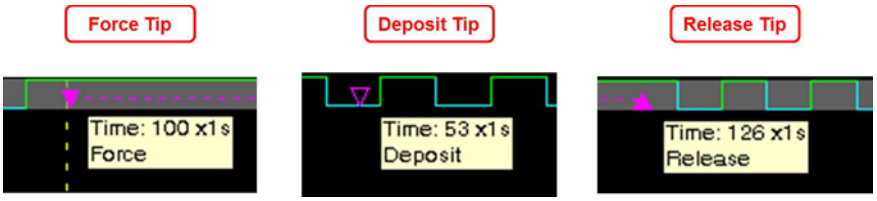


Figure: Tip in Waveform Pane

Search by Force/Deposit/Release

As shown in the following figures, you can select the **Force/Deposit/Release** option in the **Search By** list on the toolbar and click the ◀ or ▶ buttons to search backward or forward for values of the selected signal that are the result of a force, deposit, or release event.

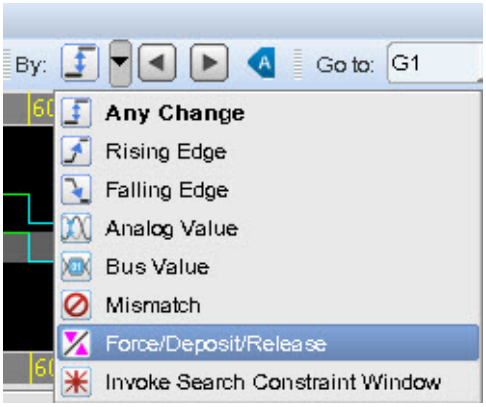


Figure: Force/Deposit/Release Option in the Search By List

Figure: Tip in Waveform Pane

Debug in the Schematic View

After an FSDB file containing dumped force information is loaded into the Verdi platform, you can also see the force values in the *nSchema* frame. In the *nSchema* frame, use the **Schematic -> Active Annotation** command to annotate the current forced value below the signal with the caret "^" character, as shown in the following figure.

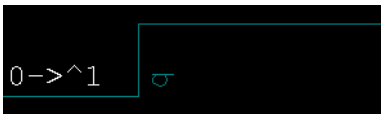


Figure: Forced Value in nSchema

When the T**race -> Active Fan-in** command is invoked for the selected signal, and if a force, deposit, or release event causes a value change of the signal, the event will be regarded as the fan-in tracing result. Active fan-in tracing will stop at the cell if its output signal is in one of the following time durations or time points:

- Forced state
- The time point of the forced state changed to release
- The time point of a deposit event

Debug in the Temporal Flow View

In the *Temporal Flow View* frame, the force, deposit, or release events will be considered as the cause when tracing the selected signal. As shown in Figure 13, you can see the forced value is shown with the caret "^" character and the *(forced)*, *(deposited)* or *(released)* strings suffix. If a value is forced with unknown value, the "^x" will be added as the root cause for the forced value, as shown in the following figure.

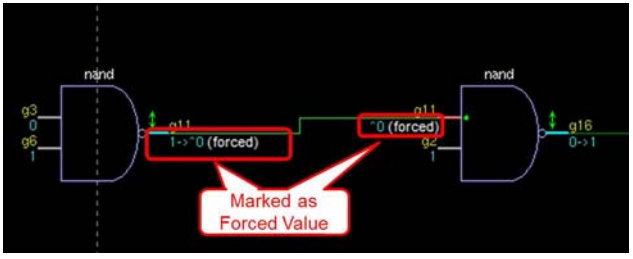


Figure: Forced Value in Temporal Flow View

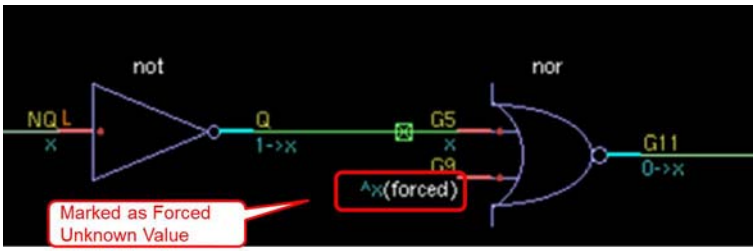


Figure: Forced with Unknown Value in Temporal Flow View

NOTE: If the force or release command is in the design, the command and the selected tracing signal should be in the same scope.

In the *Compact Temporal Flow View frame*, the forced value is also shown with the *(forced)*, *(deposited)* or *(released)* strings suffix.

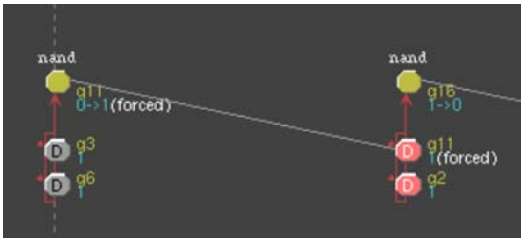


Figure: Forced Value in Compact Temporal Flow View

Practice

Complete the following steps to dump the force/release information into the FSDB file and then debug the design with forced values in the Verdi platform.

1. Set following variable for facilitating the script:

   ```
   > setenv VERDI_HOME <VERDI_INST_DIR>
   ```

2. Change to your local directory and copy the demo case to your local directory:

   ```
   > cd
   > cp $VERDI_HOME/demo/verilog/cpu/src .
   ```

3. To illustrate the Verdi Force Debug features with this demo case, modify the design to use a force statement to postpone the appearance of the 'ACE' signal. Add the following code in the `BJsource.v` file:

```
//to force Card_i at 600ns and release it after 100ns
initial
begin
      $fsdbDumpvars("+all");
      #600 force   Card_i = 32'h0;
      #100 release Card_i;
end
```

4. Set the environment variable:

```
> setenv FSDB_FORCE 1
```

5. Compile the design, enable force debug dumping features, and run the simulation:

```
> vcs -full64 -sverilog -line -debug_access+all -f run.f -lca
> ./simv
```

6. Import the design and load the generated FSDB file into the Verdi platform:

```
> verdi -f run.f -ssf novas.fsdb -nologo &
```

7. In the nWave frame, click the  icon to add signals.
Double-click the *ACE* signal of the t*b_CPUsystem* scope to add it to *nWave*. Refer to the following figure, select the *i_BJsource* scope and the following highlighted signals, and click the **OK** button in the *Get Signals* form.

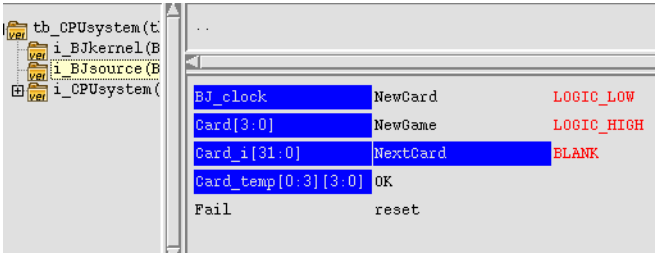


Figure: Select Signals in the Get Signals Form

As shown in the following figure, you can see the waveform of the raising edge of the *ACE* signal and the related signals. Also, the forced event and duration of the *Card_i* signal has been marked.

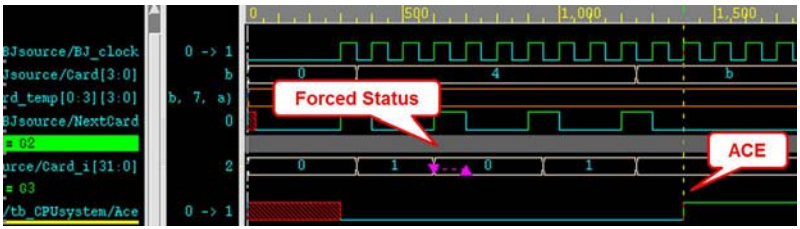


Figure: ACE and related signals waveform

8. Drag and drop the *Card_i* signal from the *nWave* frame to the source code frame. The results of trace connectivity are shown in the **OnesTrace** tab of the Message frame. The force/release statements are also regarded as results of drivers.

```
<1> tb_CPUsystem.i_BJsource.Card_i[31:0] /* results of trace connectivity */
  *Total   :    4 driver(s),          2 load(s)
  *tb_CPUsystem.i_BJsource :                  4 driver(s),          2 load(s)
    *<D> BJsource.v(60): Card_i = 0;
    *<L> BJsource.v(117): Card = Card_temp[Card_i%4];
    *<DL> BJsource.v(118): Card_i = Card_i +1;
    *<D> BJsource.v(126): #600 force   Card_i = 32'h0;
    *<D> BJsource.v(127): #100 release Card_i;
```

Figure: Tracing Result

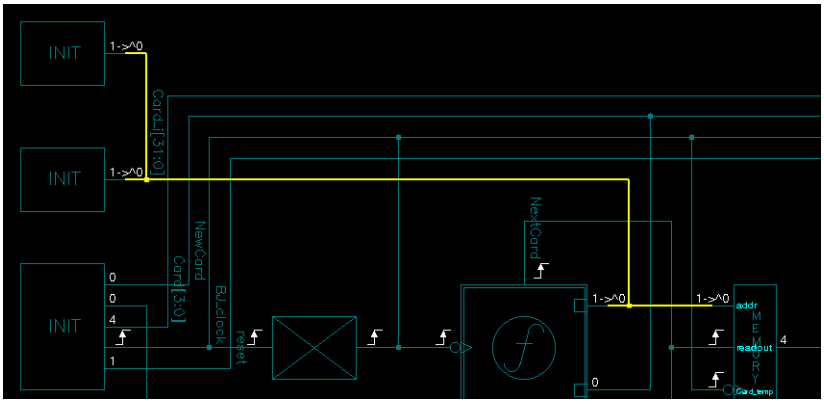
9. In the *nWave* frame, select the *Card_i* signal and use the 

icons to find the first force event. The cursor is located at 600ns.

10. In the main *nTrace* window, invoke the **Source -> Active Annotation** command. The value change with the current force values of the *Card_i* signal are annotated in the source code and *Signal List* frames.

```
114         if (NextCard == 1)
                f
115         begin
116             NewCard = 1;
                0
117             Card = Card_temp[Card_i%4];
                4    4          1->^0
118             Card_i = Card_i +1;
                1->^0    1->^0
119         end
```

11. 7.In the *nSchema* frame, invoke the **Schematic -> Active Annotation** command. The current force values are annotated.



12. Select the *Card_i* signal and invoke the **Trace -> Active Fan-in** command. As shown in the following figure, the forced event causes a value change of the *Card_i* signal and the active fan-in tracing stops at the cell whose output signal is in the force state.

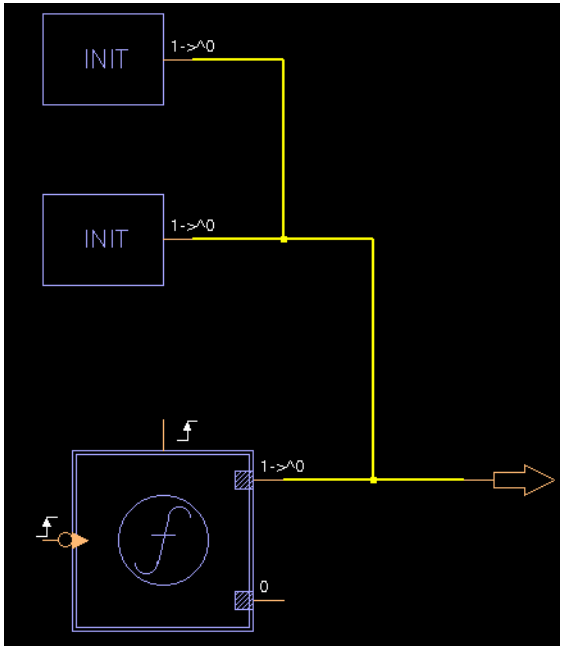


Figure: Fan-in Cone Result of Card_i

Debug with SystemVerilog

Before you begin this application, follow the instructions in the *Before You Begin* chapter.

The Verdi platform provides a set of features that allow you to debug SystemVerilog designs.

The design is based on the standard Verdi CPU case; however, it has been re-written in SystemVerilog. There are also some assertions coded in SVA.

Import the Design

There are two methods for importing the design: load the files directly or create a compiled library.

1. Change your context to the *systemverilog* sub-directory, which is where all of the demo source code files are located:

   ```
   % cd <working_dir>/demo/systemverilog
   ```

2. Modify the SETUP file to point to the correct Verdi and simulator installation paths in your environment and source the file.

   ```
   % source ./SETUP
   ```

Refer to the *Language Support* chapter of the *Verdi and Siloti Command Reference* manual for complete details on compiling and importing different languages.

Load Files Directly

Since the code is all SystemVerilog, you can compile and load the source files directly without pre-compiling.

1. Start the Verdi platform by referencing the design files. If you do not use a common file extension, you need specify the -sv command line option.

   ```
   % verdi -f run.f -sv -workMode hardwareDebug &
   ```

   Alternatively, if all files have the same file extension (e.g. .sv, .SV) you can specify the *+systemverilogext+* option instead. Refer to the RUN script in the demo directory for an example.

   The Verdi platform opens to display the SystemVerilog design source code.

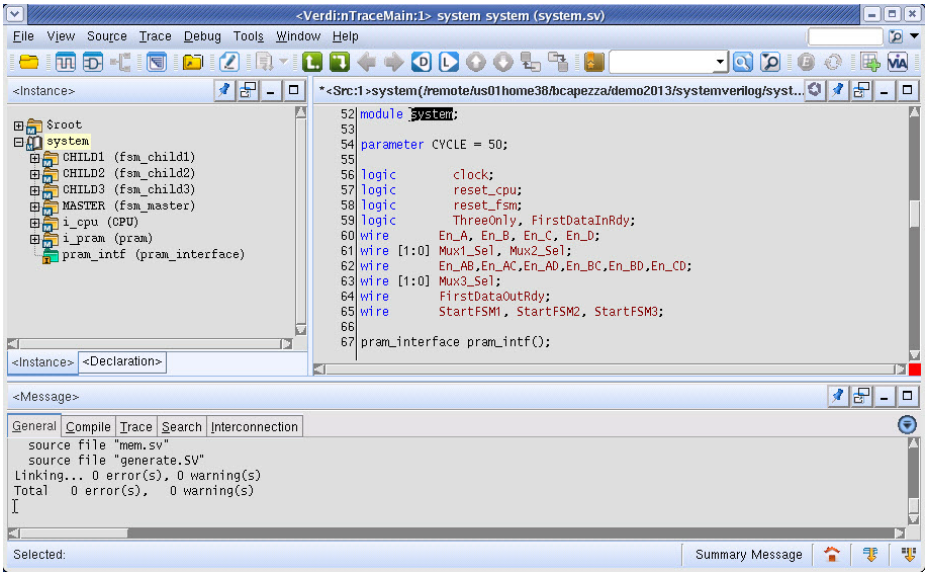


Figure: nTrace with SV Code Loaded

Use Compiled Library - Optional

You can compile the SystemVerilog design into a work.lib++ compiled library and load from there. For designs that are mixed-language, it is recommended to compile the design first and then load. This includes designs that are mixed Verilog/SystemVerilog as you may have a Verilog design (that is not SV-compliant) and add code (including SVA) that is in SystemVerilog.

1. Compile the library. By default work.lib++ is created. If all files have the same file extension (e.g. .sv, .SV) you can specify the *+systemverilogext+* option.

   ```
   % vericom -f run.f +systemverilogext+.sv+.SV+
   ```

   Alternatively, if you do not use a common file extension, you need specify the -sv command line option. Refer to the COMPILE script in the demo directory for an example.

2. Load the compiled library.

   ```
   % verdi -lib work &
   ```

Visualize SystemVerilog Source Code

The Verdi platform provides advanced visualization capabilities that allow you to quickly understand SystemVerilog code.

Design Browser Frame

1. In the **Instance** tab of the design browser frame, click the plus symbol to the left *i_pram* block instance name to expand its sub-blocks. You will see an interface, *pram_intf*.

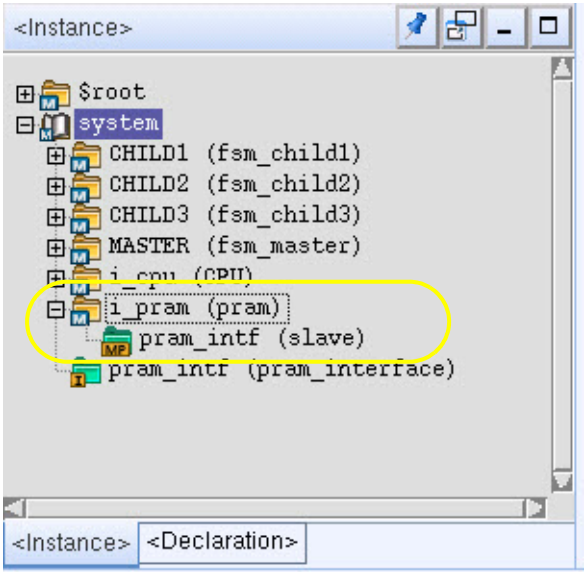


Figure: SV in nTrace

2. Double-click *i_pram* to display its associated source code.

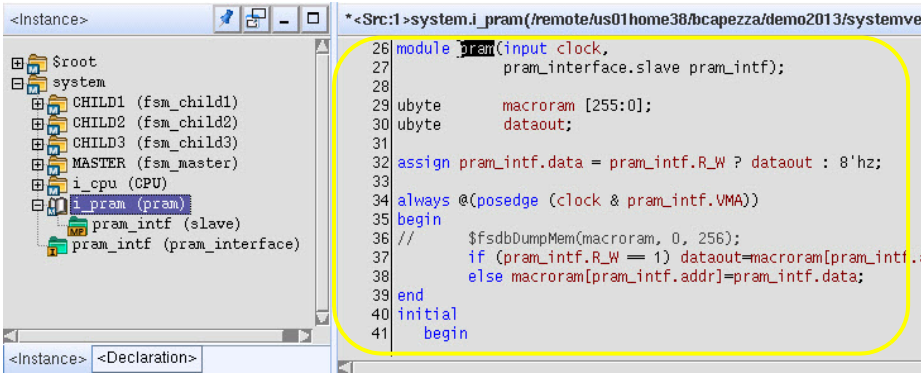


Figure: SVA Properties and Sequences in nTrace

3. In the design browser frame, click the plus symbol to the left *i_cpu*, *i_ALUB*, *i_alu* and *i_Nbit_adder* block instance names to expand their sub-blocks.

4. Click the plus symbol to the left *addbit[0]* and *addbit[1]* under *i_Nbit_adder* to expand the generate instances. They each contain 5 primitives.

Figure: Generate Instances

Each generate instance is a scope in the design browser frame with its elements within.

5. Double-click *addbit[0]* and then *addbit[1]* to display the associated source code.

   You will go to the same place in the source code but the corresponding simulation data will be annotated based on your current active scope.

Source Code

1. In the design browser frame, double-click *i_ALUB* to display its source code.
2. In the source code frame, double-click *S1* to trace drivers.

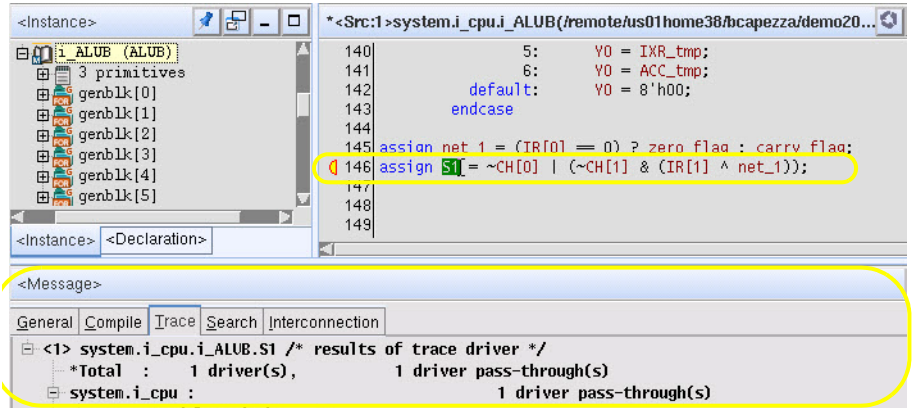


Figure: Trace Driver Results for S1

The source code changes to the driver location for *S1* and the message frame displays the results summary.

3. In the design browser frame, double-click *pram_intf(pram_interface)* to display the source code.
4. In the source code frame, left-click to select *data*.
5. Right-click to open the right mouse button menu and select the **Connectivity** command to trace connectivity for *data*.

   The **OneTrace** tab of the message frame indicates there are 2 drivers, 2 loads and several pass-throughs. You can double-click any line in the message frame to go to the equivalent code.
6. In the *nTrace* main window, click the **Show Previous/Show Next** icons to step through driver/load results in the current scope (*i_pram*).
7. Click the **Show Previous in Hierarchy** icon to locate results in a different scope (*i_PCU*).

   After tracing drivers or loads or both, you can easily traverse the hierarchy to locate the results.

Schematic

1. In the design browser frame, double-click *system* to display the associated source code.

   On the toolbar, click the **New Schematic** icon to open an *nSchema* frame of the top level design.

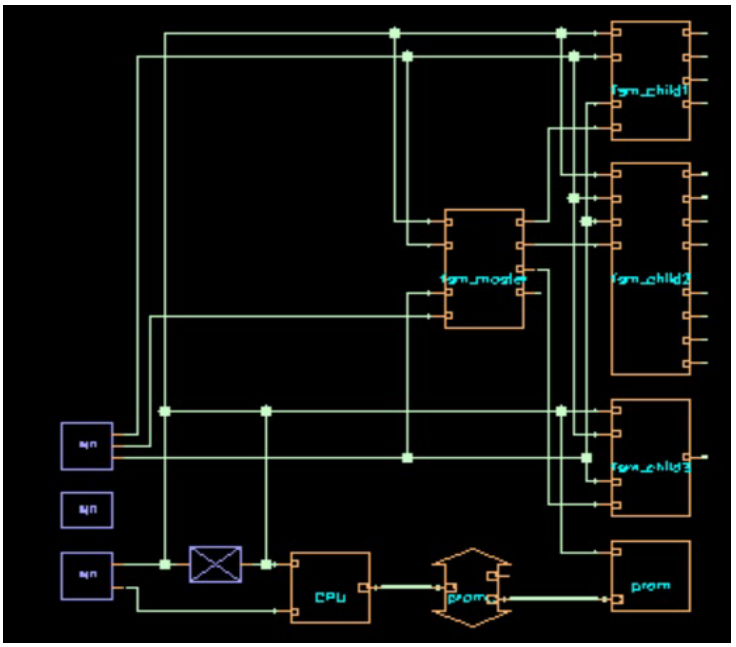
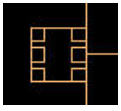


Figure: Schematic for system



2. Drag with left mouse button to zoom in on the lower right corner. Note the interface port symbols on *pram* and *pram_interface*.
3. Click the **Zoom All** icon.
4. Double-click the *CPU* block in the lower left to descend to the next level.
5. Double-click the ALUB block on the left to descend to the next level.

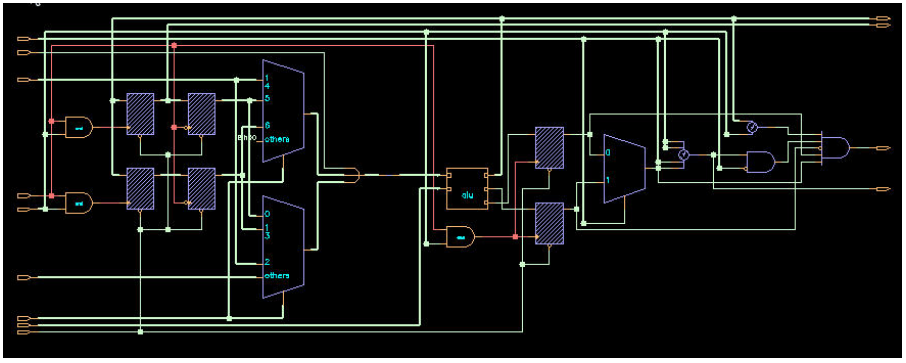


Figure: Schematic for ALUB

6. In the *nSchema* frame, choose the **File -> Close Window** command.

View SystemVerilog Simulation Results

You have loaded the source code and used the Verdi platform to understand SystemVerilog designs. Now you will load simulation results into the system to utilize the full power of the Verdi platform. A simulation results file (sv.fsdb) already exists; however, you can dump the simulation results again (including assertion results) from your simulator (VCS was used in this example). See the SETUP and SIMULATE scripts in the demo directory for details on appropriate simulator commands.

In this example, system.sv calls *$fsdbDumpfile* to specify the output FSDB file (sv.fsdb), *$fsdbDumpSVA* to dump SVA data, and *$fsdbDumpvars* to dump standard RTL design data.

After simulation, the FSDB data can be loaded into the Verdi platform so you can view waveforms, annotate on source code, and use the automatic tracing capabilities.

NOTE: To dump the successful assertion, you must perform any one of the following:

Set the environment variable as follows:

```
% setenv FSDB_SVA_SUCCESS 1
```

Or

Add the **+fsdb+sva_success** option at runtime as follows:

```
% simv +fsdb+sva_success
```

Waveform

1. In the *nTrace* main window, click the **New Waveform** icon to open the *nWave* frame. Alternatively, you can choose the **Tools -> New Waveform** command.
2. In the *nWave* frame, choose the **File -> Open** command to open the FSDB file or click the **Open File** icon on the toolbar to open the *Open Dump File* form.
3. In the *Open Dump File* form, left-click to select *sv.fsdb* and click **Add** and then **OK** to load the file.

NOTE: You can also load the FSDB file on the command line when you first bring up the Verdi platform. For example:
`% verdi -sv -f run.f -ssf sv.fsdb ...`

---

4. In the design browser frame, select *i_ALUB* and use the middle mouse button to drag and drop the scope to the *nWave* frame.

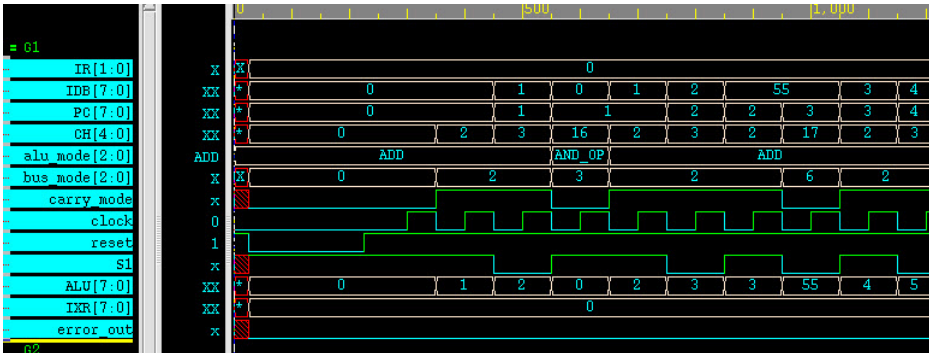


Figure: SV in nWave

5. Zoom in around time 800-1500.

   You can search for any changes in the waveform.

6. In the *nWave* frame, left click to select *ALU[7:0]*.
7. Choose the **Waveform -> Set Search Value** command to open the *Search Value* form.
8. In the *Search Value* form, enter 55 in the **Signal Value** field and click the **OK** button.

   The search **By:** field on the *nWave* toolbar changes to **Search by Bus Value**.

9. On the *nWave* toolbar, click the **Search Forward** icon to locate the value of 55 on ALU at time 950.
10. Double-click the 3->55 transition on *ALU[7:0]* to locate the active driver in the source code frame.

Source Code

Another very convenient method to visualize simulation data is through active annotation which allows you to see simulation results under the corresponding variable in the source code frame.

1. In the main window, choose the **Source -> Active Annotation** command (or press the bind key **X** on the keyboard) to enable active annotation.

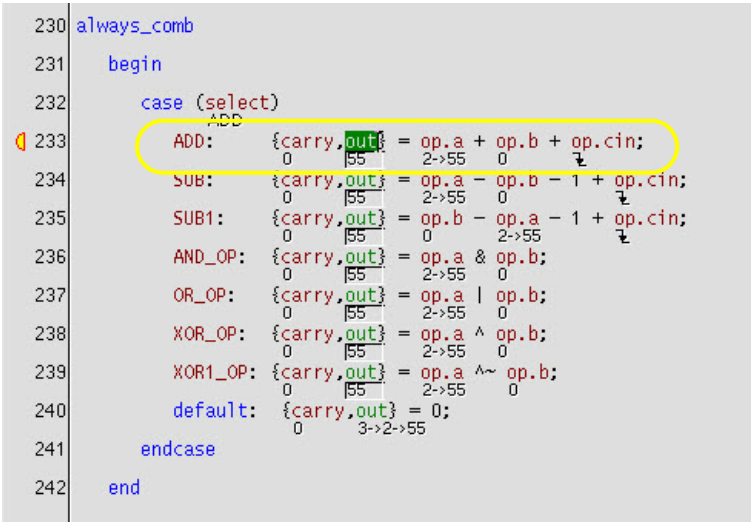


Figure: Active Annotation in the Source Code Frame

You will see the simulation values under the signal names. The time is synchronized with the cursor time in the *nWave* frame. If you change the cursor in *nWave*, the values annotated will change accordingly.

2. Left-click to select op.a on line 233.
3. Right-click to open the right mouse button menu and select **Active Trace** to locate the active driver for op.a.

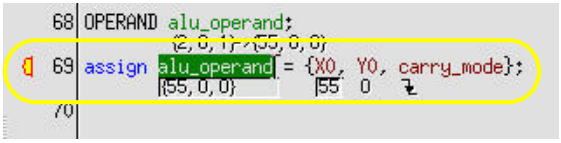


Figure: Active Trace Results for op.a

The driver is a complex signal structure in the *ALUB*.

4. In the *nTrace* main window, choose the **View -> Signal List** command to open the *Signal List* frame.
5. In the *Signal List* frame, select *alu_operand*. The frame will be similar to the following.

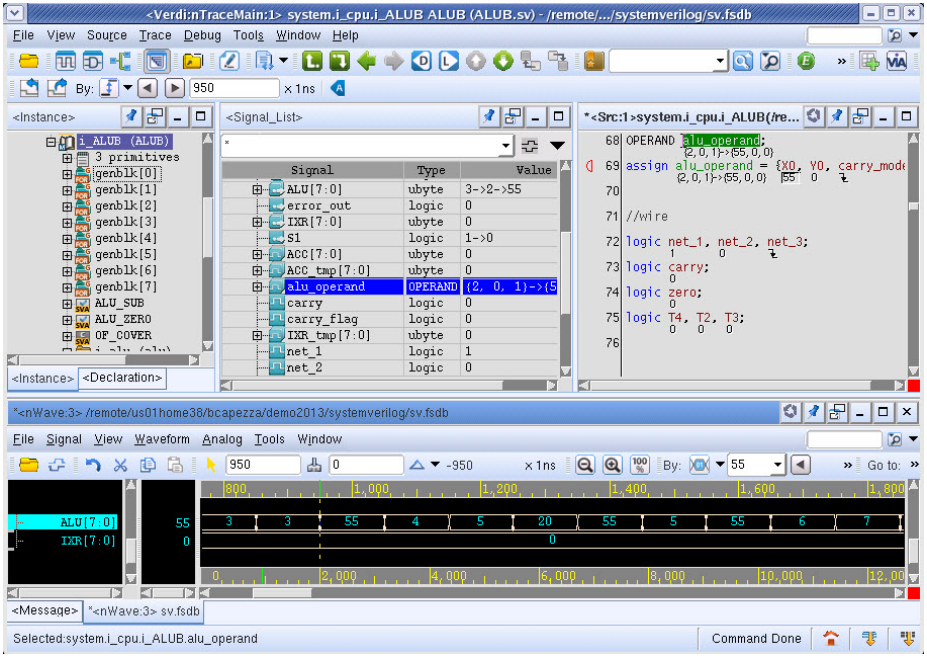


Figure: Signal List Frame

The *Signal List* frame makes it easier to view and understand complex signals. You can drag and drop between this frame and source code or *nWave* as needed.

Generate Constructs

For generate constructs, genvars are not dumped to FSDB during simulation. However, the Verdi platform can elaborate the values of genvars and also build the correct hierarchy for the generated instances in the design browser frame. These elaborated values can also be annotated.

1. In the design browser frame, double-click *addbit[1]* to change the scope. You will see the annotation under the design signals, and you can also see the value for *i* which is a genvar.

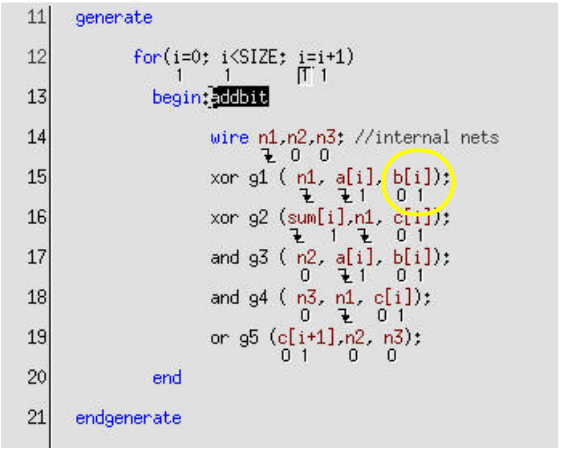


Figure: Parameter Annotation

2. Double-click *addbit[3]* under the ALUB hierarchy to change the scope again and you will see 3 annotated for *i*.

Debug with SystemVerilog Assertions (SVA)

Before you begin this application, follow the instructions in the *Before You Begin* chapter.

The Verdi platform provides a set of features that allow you to debug assertions.

The design is based on the standard Verdi CPU case; however, it has been re-written in SystemVerilog. There are also some assertions coded in SVA. This application assumes and uses simulation-based assertion checking. In particular, VCS (2006.06) was used to generate the SV(A) data.

Import the Design

There are two methods for importing the design: load the files directly or create a compiled library.

1. Change your context to the *systemverilog* sub-directory, which is where all of the demo source code files are located:

   ```
   % cd <working_dir>/demo/systemverilog
   ```

2. Modify the SETUP file to point to the correct Verdi and simulator installation paths in your environment and source the file.

   ```
   % source ./SETUP
   ```

Refer to the *Language Support* chapter of the *Verdi and Siloti Command Reference* manual for complete details on compiling and importing different languages.

Load Files Directly

Since the code is all SystemVerilog, you can compile and load the source files directly without pre-compiling.

1. Start the Verdi platform by referencing the design files. If you do not use a common file extension, you need to specify the -sv command line option.

   ```
   % verdi -f run.f -sv -workMode hardwareDebug &
   ```

   Alternatively, if all files have the same file extension (e.g. .sv, .SV) you can specify the +*systemverilogext*+ option instead. Refer to the RUN script in the demo directory for an example.

The Verdi platform opens to display the SystemVerilog source code. The *nTrace* main window serves as the main view from which the other views can be started.

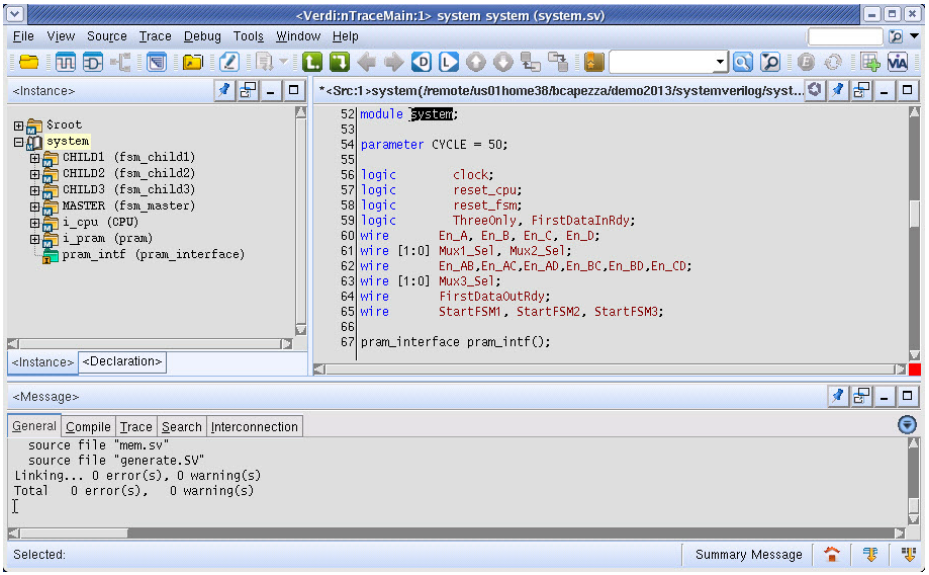


Figure: nTrace with SV and SVA Code Loaded

Use Compiled Library - Optional

You can compile the SystemVerilog design into a work.lib++ compiled library and load from there. For designs that are mixed-language, it is recommended to compile the design first and then load. This includes designs that are mixed Verilog/SystemVerilog as you may have a Verilog design (that is not SV-compliant) and add code (including SVA) that is in SystemVerilog.

1. Compile the library. By default work.lib++ is created. If all files have the same file extension (e.g. .sv, .SV) you can specify the *+systemverilogext+* option.

   ```
   % vericom -f run.f +systemverilogext+.sv+.SV+
   ```

   Alternatively, if you do not use a common file extension, you need to specify the -sv command line option. Refer to the COMPILE script in the demo directory for an example.

2. Load the compiled library.

   ```
   % verdi -lib work -workMode hardwareDebug &
   ```

Visualize SVA Source Code

The Verdi platform provides advanced visualization capabilities that allow you to quickly understand SystemVerilog Assertions. These capabilities are even

more critical to design teams when new methodologies such as assertions are introduced into the flow.

Design Browser and Source Code

1. In the design browser frame, click the plus symbol to the left *i_cpu* block instance name to expand its sub-blocks. You will see four assertions: 3 asserts (INCPC, INCPC2, LD) and a cover (COVER_e_r3).

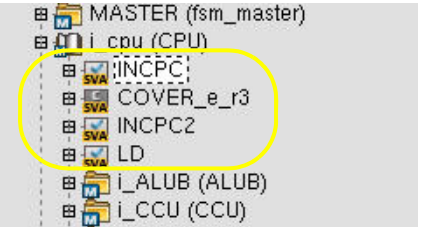


Figure: SVA in nTrace

2. Click the plus symbol to the left of the *i_ALUB* block instance name to display several more assertions, including one cover directive.
3. Double-click *INCPC* to display its underlying property (*e_INC*) and associated source code. You can expand any assertion to see the underlying properties the assertion is built upon.
4. Click the plus symbol to the left *e_INC* to display its underlying sequences (*e_l* and *e_r*). You can expand any property to see the sequences it is built upon.

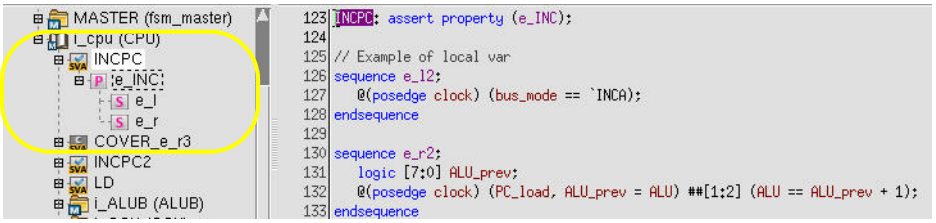


Figure: SVA Properties and Sequences in nTrace

5. Display the source code for *INCPC2* by double-clicking on *INCPC2* in the design browser frame.
6. In the source code frame, double-click the *e_INC2* property to trace to its description.

7. Left click to select *e_r2*, right-click to view the right mouse button context menu and select **Show Definition** to trace back to its description (it is a sequence).

NOTE: Both double-click and the **Show Definition** command will display the description for properties or sequences.

Note that *e_r2* has a local variable, *ALU_prev*. SVA allows local variables which in turn permits users to write powerful assertions. However, local variable and assertions containing them are difficult to debug since assertions by their nature can each have multiple attempts with several sequence threads within. Synopsys provides for the capture and advanced visualization of local variables.

8. In the design browser frame, click the plus symbol to the left *genblk[0]* and *genblk[1]* under *i_ALUB* to expand the generate instances. They each contain a cover directive, *cv*.

Figure: Generate Instances

Each generate instance is a scope in the design browser with its elements (assertion / property / sequence) within.

9. Double-click *genblk[0]* and then *genblk[1]* to display the associated source code.

   You will go to the same place in the source code but the corresponding simulation data will be annotated based on your current active scope.

View SVA Simulation Results

You have loaded the source code and used the Verdi platform to understand designs containing assertions. Now you can load simulation results into the

system to utilize the full power of the Verdi platform. A simulation results file (sv.fsdb) already exists; however, you can dump the simulation results again (including assertion results) from your simulator (VCS was used in this example). See the SETUP and SIM_SVA scripts in the demo directory for details on appropriate simulator commands.

In this example, system.sv calls *$fsdbDumpfile* to specify the output FSDB file (sv.fsdb), *$fsdbDumpSVA* to dump SVA data, and *$fsdbDumpvars* to dump standard RTL design data.

After simulation, the FSDB data can be loaded into the Verdi platform so you can view waveforms, annotate on source code, and use the automatic tracing capabilities.

1. In the *nTrace* main window, select the **Window -> Assertion Debug Mode** command to enable more frames to assist with assertion debug.
2. In the *nWave* frame, choose the **File -> Open** command to open the FSDB file or click the **Open File** icon on the toolbar to open the *Open Dump File* form.
3. In the *Open Dump File* form, left-click to select *sv.fsdb* and click **Add** and then **OK** to load the file.

NOTE: You can also load the FSDB file and enable the assertion debug work mode on the command line when you first bring up the Verdi platform. For example:

```
% verdi -sv -f run.f -ssf sv.fsdb
-workMode assertionDebug
```

Statistics Frame

On the *Statistics* frame, the assertion results can be shown in a tabular spreadsheet-like format. The **FSDB Statistics** tab summarizes the results for one or more FSDB files and the **Property Statistics** tab displays the results for individual assertions from all FSDB files. The **Property Details** table displays results for individual assertions.

1. In the *Statistics* frame, click the **FSDB Statistics** tab, select the cell with value 6 in the **Assert** row under the **Fail** column.

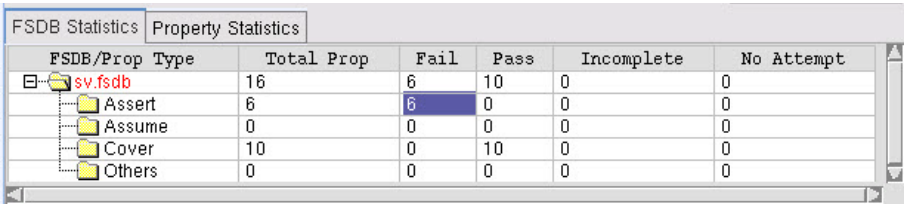


Figure: FSDB Statistics

2. Double-click the cell to add all failing assertions to the **Property Details** table.

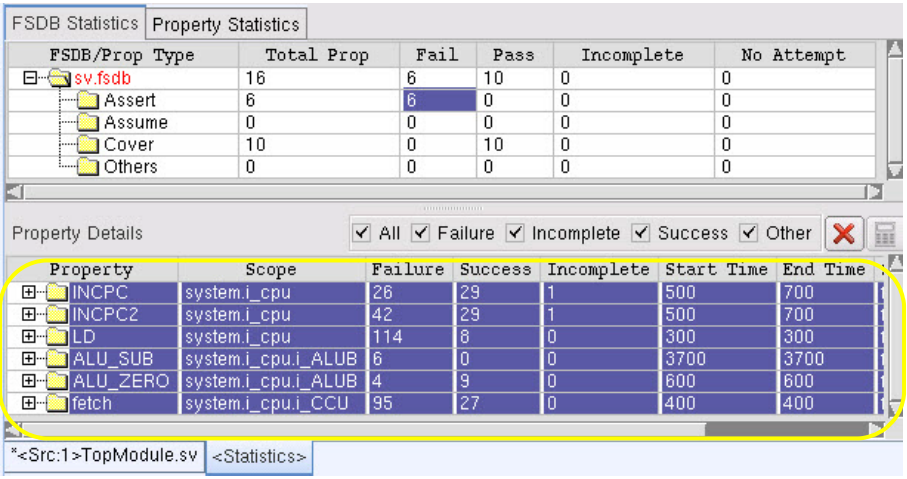


Figure: Property Details

You can select the vertical bar between column headers and drag-left to change the column width.

3. In the *Statistics* frame, click the **Options** icon to open the *Preferences* form. The options allow you to control the time range, the property status/type and the fields for display.
4. In the *Preferences* form, select the **Property** folder **-> Property Details** folder **-> View** page, and then specify *i_cpu in the **Scope** field.

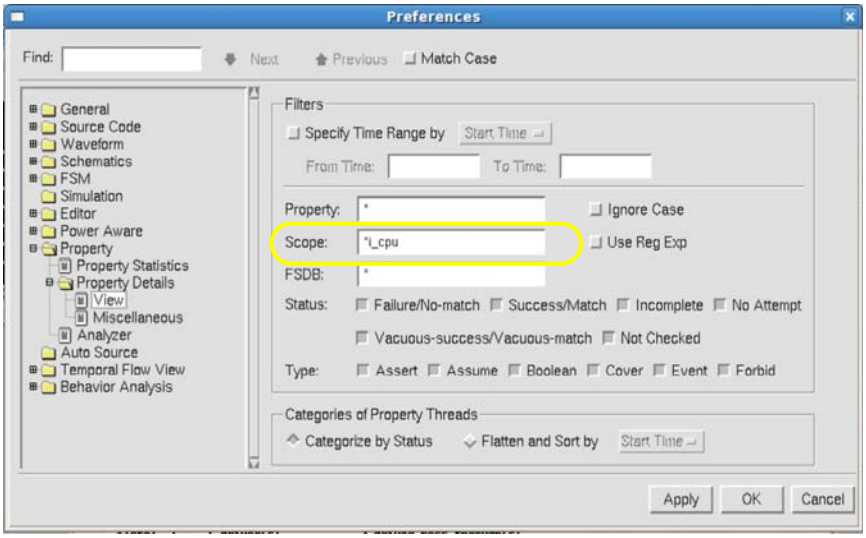
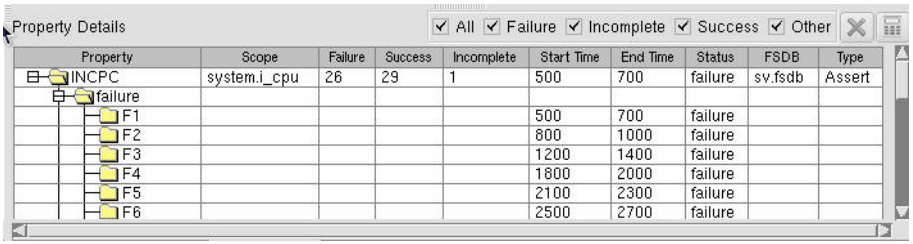


Figure: Preferences Form - Property Details Page

5. Click the **OK** button. Only the properties for scope *i_cpu* will be displayed.
6. Click the + symbol associated with *INCPC* to display failure and success groups. Click the + symbol on failure/success group to display the individual failures/successes.



Property	Scope	Failure	Success	Incomplete	Start Time	End Time	Status	FSDB	Type
INCPC	system.i_cpu	26	29	1	500	700	failure	sv.fsdb	Assert
failure									
F1					500	700	failure		
F2					800	1000	failure		
F3					1200	1400	failure		
F4					1600	2000	failure		
F5					2100	2300	failure		
F6					2500	2700	failure		

Figure: Assertion Failures

The failures are named F1 to Fn starting from the first failure in time. Successes are named S1 to Sn starting from the first success in time.

Waveform

There are a variety of ways to add signals to the waveform. You can drag and drop assertions, properties, sequences, or instances from the design browser frame, the source code frame, or the *Statistics* frame. You can also use the **Get Signals** command. The best method is to select properties and failures of interest from the **Property Details** section of the *Statistics* frame and have them automatically added to the *nWave* frame.

1. In the *Statistics* frame, click the **Options** icon to open the *Preferences* form.
2. In the *Preferences* form, select the **Property** folder **-> Property Details** folder **-> Mischellaneous** page and turn on the **Sync Cursor Time with Selected Property** and **Add Selected Property to nWave When Not Found** options.
3. Select the **Property** folder **-> Analyzer** page and turn on the **Add Evaluated Signals to nWave Automatically** option.
4. In the **Property Details** section of the *Statistics* frame, scroll down to the row containing *INCPC2* and select it. The property is added to the *nWave* frame and the time changes to its first failure.
5. In the **Property Details** section of the *Statistics* frame, scroll back up and select the row containing F2 under *INCPC*. *INCPC* is automatically added to the waveform and the cursor is located at the failure end point for F2. The assertions waveforms will be similar to the following:

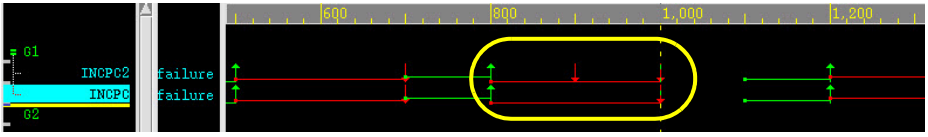


Figure: SVA in nWave

Both *INCPC2* and *INCPC* are temporal assertions. The *nWave* frame shows the time the assertion started evaluating to when it passed or failed using a horizontal line.

6. Zoom in around time 300-1100.

   It is quite difficult to make out the different assertion start times and pass/fail times because of overlapping. In such cases, you can expand the number of rows that are used to display the assertion waveform.

7. Select *INCPC2* (or *INCPC*) and choose the **Waveform -> Property -> Expand Overlapping** command.

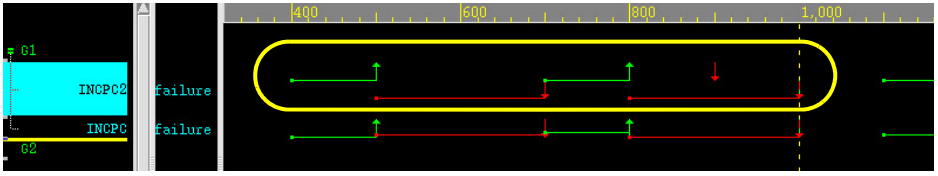


Figure: INCPC2 Expanded in nWave

The number of rows used to display the waveform for *INCPC2* will expand to 2 so that there is no overlap.

8. Choose the **Waveform -> Property -> Shrink Overlapping** command to go back to a single row.

You can search for assertion passes, fails, and evaluation begins in the waveform.

9. In the *nWave* frame, click the **By:** menu on the toolbar and select the **No-Match/Failure** option (red vertical line/down arrow).

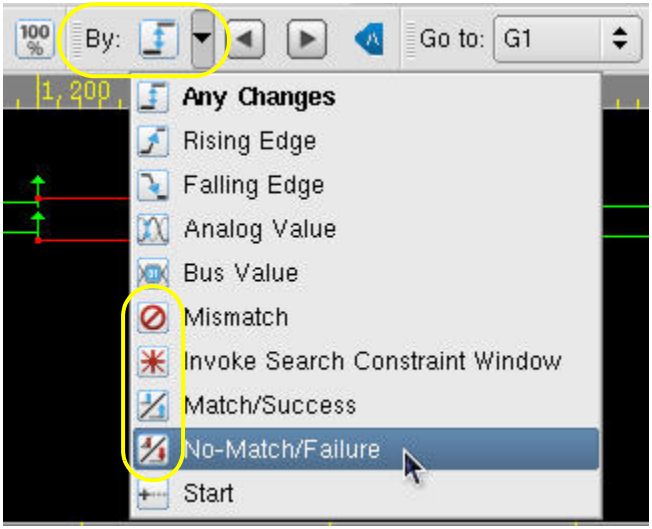


Figure: Search By Options

10. With *INCPC2* selected, click the **Search Forward** or **Search Backward** icons (right or left arrows), to move the cursor to the next or previous failure.

    Let's debug the first failure of the *INCPC2* assert.

11. In the waveform pane of the *nWave* frame, double-click the first failure of *INCPC2* (at 700 ns) to expand the underlying signals, properties, and sequences from which the assertion is created.

NOTE: If associated signals are not added into the *nWave* frame, click the **Options** icon in the *Statistics* frame, go to the **Property -> Analyzer** page and turn on the **Add Evaluated Signals to nWave Automatically** option.

All associated properties, signals, and local variables will be expanded, and sub-groups will be created automatically. All properties which associated with the assert are calculated by Verdi dynamically.

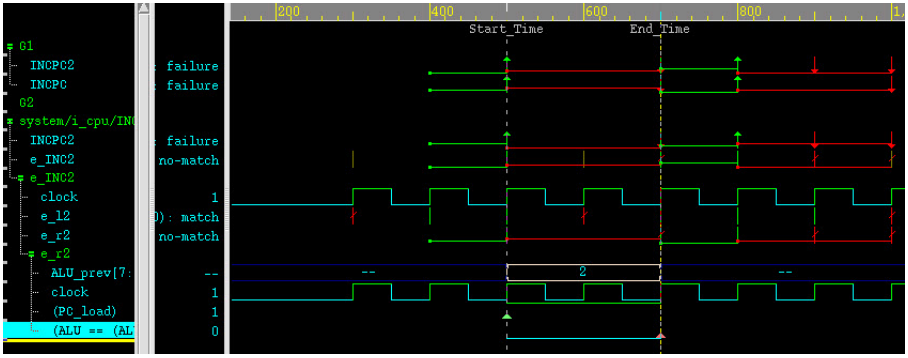


Figure: Expand Assertions, Properties and Sequences

In addition, the failure will be evaluated and displayed in the *Analyzer* frame. Refer to the *Analyze SVA Assertions* section for details.

Source Code

Another convenient method to visualize simulation data is through active annotation that allows you to see simulation results under the corresponding variable in the source code frame. This capability has been extended to assertion elements as well.

1. In the design browser frame, double-click *e_INC2* to display the associated source code.
2. In the main window, choose the **Source -> Active Annotation** command (or press **X** on the keyboard) to enable active annotation.

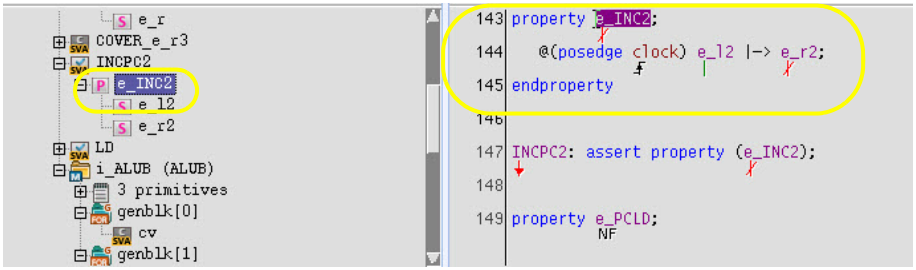


Figure: Active Annotation in Source Code Frame

You can see the simulation values under the variable names. For assertions, the following notations are used:

- Green up arrow / Green vertical line: Success / Match
- Red down arrow / Red vertical line: Fail / No-match
- SE: Start Evaluation

- UE: Under Evaluation
- NF: Not Found in FSDB file
- NV: No Value at current time

Active annotation will display NF for assertion-related variables (assertions, properties, sequences, local variables) if you are not in the scope where the variables are referenced. To see the values (instead of NF), change scope to the appropriate assertion, property, or sequence.

The time is synchronized with the cursor time in the *nWave* frame. If you change the cursor in the *nWave* frame, the values annotated change accordingly.

Generate Constructs

For generate constructs, genvars are not dumped to FSDB during simulation. However, the Verdi platform can elaborate the values of genvars and also build the correct hierarchy for the generated instances in the design browser frame. These elaborated values can also be annotated.

1. In the design browser frame, double-click *genblk[1]* to change the scope. You can see the annotation for cv (cover), ALU (a design signal), and *i* which is a genvar.

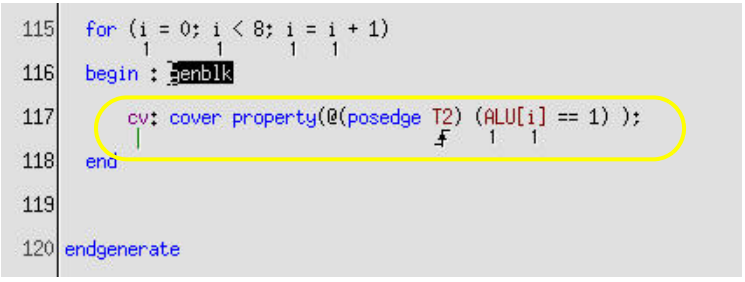


Figure: Parameter Annotation

   You can see the value of *i* annotated as 1.

2. Double-click *genblk[2]* under the ALUB hierarchy to change the scope again and you can see 2 annotated for *i*.

Analyze SVA Assertions

In addition to manually debugging the assertions using the waveform and source code, you can use the Assertion Analyzer to automatically debug the assertion and quickly locate the failing expression and signals.

1. In the **Property Details** section of the *Statistics* frame, select the row containing F2 under *INCPC2*.

2. Click the **Analyze Property** button. The *Analyzer* frame updates to display the results. The frame is similar to the following:

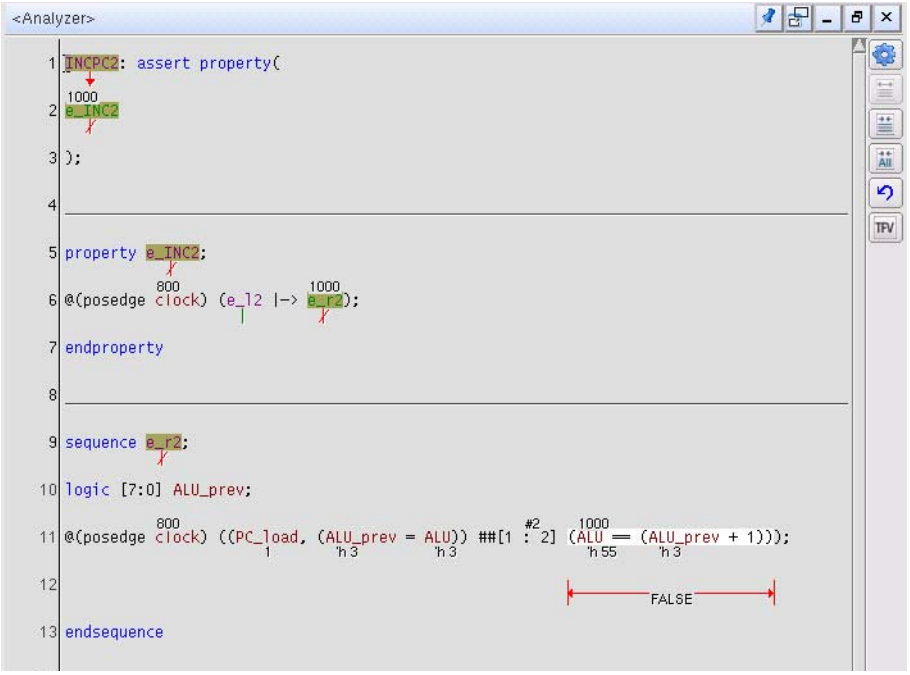


Figure: Property Tools Window - Analyzer Tab

The *Analyzer* frame is specifically designed for debugging assertions. The Verdi platform adds the related properties so you can easily see the relationship between assertion and property. The extracted source code and the annotated results are displayed the values are annotated according to the time it was evaluated.

Assertion failures always come from the violation of an expression. You can see that ALU does not have the correct value so you can use standard Verdi debug techniques to locate the root cause.

NOTE: In addition to analyzing an assertion from the *Statistics* frame, you can start the analysis from a failing signal in the *nWave* frame (just double-click) or in the source code frame (select **Assertion Analyzer** from the right mouse button menu).

NOTE: If you are using a FSDB file that was generated from the Assertion Evaluator, the local variables will not be saved in the file itself; however, when you analyze the assertion, the local variable value will be calculated on the fly.

Evaluate SVA Assertions

Assume you either have a simulation results file that does not contain SVA results or you have a FSDB file with SVA that you want to re-check. Rather than run the simulation again and dump the SVA results as well, let's use the Assertion Evaluator engine to check SVA against an existing FSDB file.

See the SETUP, SIM_RTL and SIM_SVA scripts and sim_sva_fsdb.do file in the demo directory for details on the appropriate simulator commands if you want to re-simulate the design file or use the simulator to check and dump SVA.

In this example, check the currently loaded FSDB file (sv.fsdb) again.

1. In the main window, choose the **Tools -> Property Tools -> Evaluator** command to open the *Evaluate Properties* form.
2. In the left pane click the *i_ALUB* scope (under *system -> i_cpu*), all assertions under the scope will be listed in the middle pane.
3. In the middle pane of the *Evaluate Properties* form, drag-left on assertions *ALUB_SUB* and *ALU_ZERO* to select, and then click the **Add Selected Properties** icon to add them to the right pane. These two assertions are flagged to be re-calculated by Verdi. The form will be similar to the following:

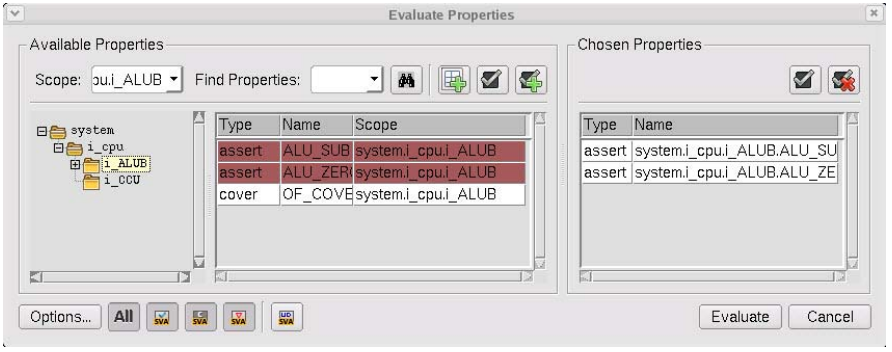


Figure: Evaluate Properties Form

4. In the *Evaluate Properties* form, click the **Evaluate** button.
5. Click **Yes** on the first *Question* dialog window and **OK** on the second.

   The Assertion Evaluator will generate an FSDB file called *sva_checker_results.fsdb.vf* containing the assertions and design signals. The **General** tab of the *Message* frame will be brought forward with a summary of the evaluation results. Select the Statistics frame to see the new FSDB file loaded in the **FSDB Statistics** tab. Now you can debug assertion failures as described previously.

SVA Evaluation of Runtime Assertions

The Assertion Evaluation Engine from the Verdi Automated Debug Platform provides a runtime assertion mechanism to add assertions during runtime without recompiling and reloading the design again.

In general, assertions are added in the design, compiled with the design, and then the results are checked by the Assertion Debug functions after they are loaded and undergo simulation. The Assertion Evaluation Engine can also check assertions against a signal level FSDB trace file for the design.

If any modifications are needed for assertion demands, the design, including the newly added assertion code must be compiled, loaded, and simulated again. The Verdi platform now provides an interface that makes it easy to insert assertions without modifying the original design and then quickly check the failure or success of the assertion. The tedious re-compiling, re-loading, and re-simulating efforts are no longer needed.

Import Designs and Assertions

The design and SystemVerilog Assertions (SVA) can be imported into the Verdi platform.

See the *Language Support and Compile/Import Methods* chapter in the *Verdi and Siloti Command Reference* manual for details.

Evaluate with Temporary Assertions

Use the following steps to evaluate assertions:

1. From the **Tools** -> **Property Tools** -> **Evaluator** pane, select the *Evaluator* pane to open it. Or click the **Evaluator** button in the **Add Temporary Assertions** form.

   The assertions that are temporarily added will be displayed in blue color to distinguish it from the original assertions.

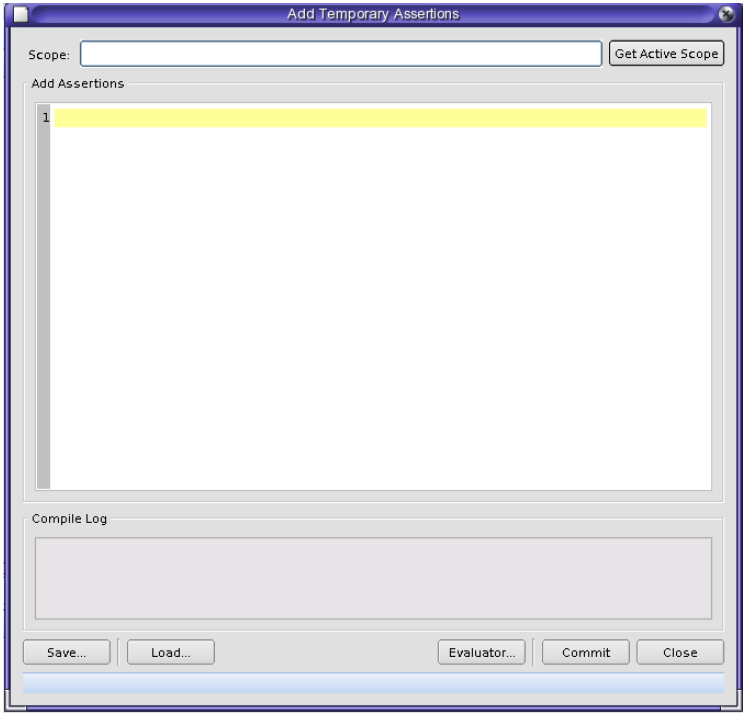


Figure: Add Temporary Assertions Form

2. Enter the **Scope** or click **Get Active Scope** to automatically enter the active scope.
3. Edit the assertion statement in the **Add Assertions** field as shown in the following figure.
4. Click the **Commit** command.

   The **Compile Log** field displays the compilation results of the new assertion code.

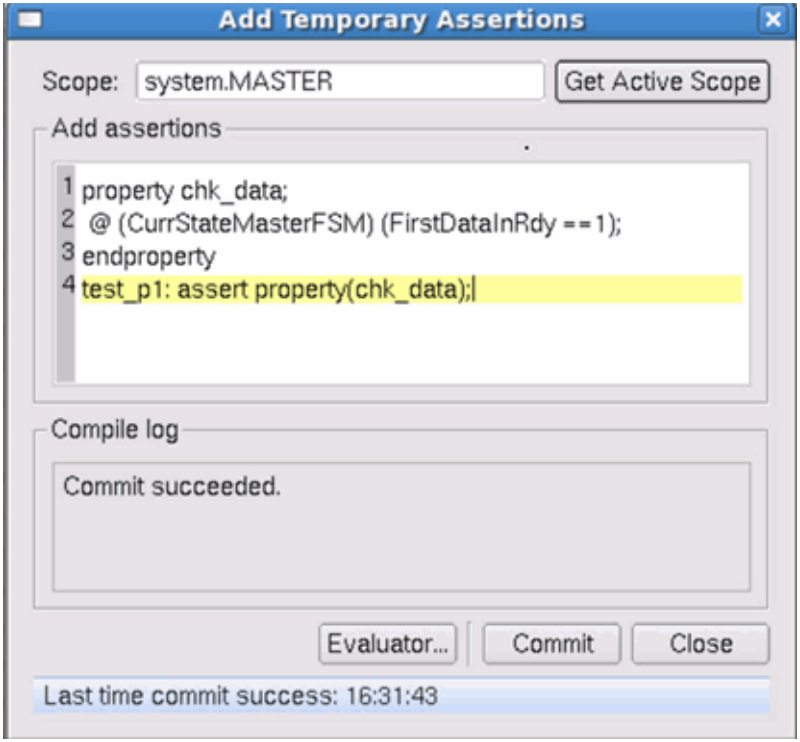


Figure: Add Temporary Assertion Example

Save Temporary Assertions

Perform the following steps to save temporary assertions:

1. Select **Tools** -> **Property Tools** -> **Add Temporary Assertion** command. The **Add Temporary Assertion** form opens.
2. After editing the assertion statements, click **Save** to save the assertions and scope to a file with `.tsva` extension as shown in the following figure.

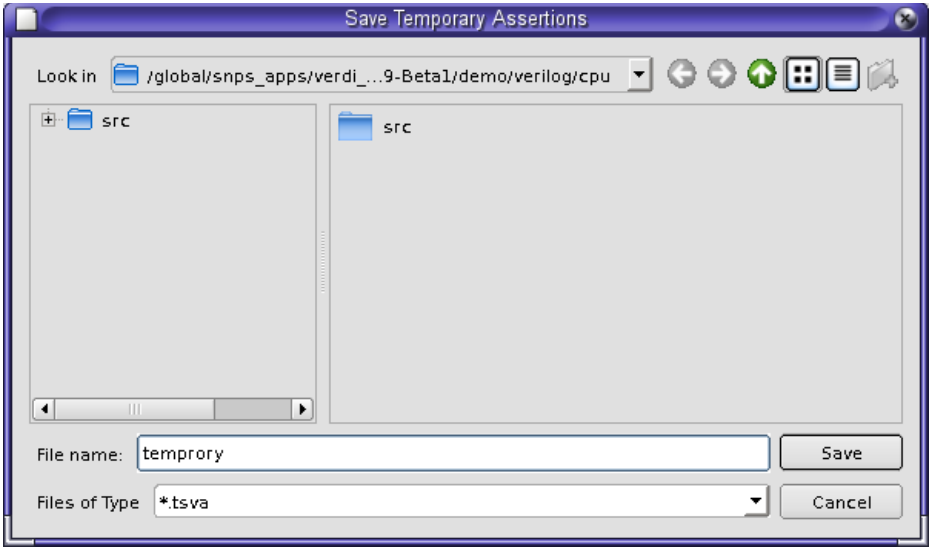


Figure: Save Temporary Assertion

Load Temporary Assertions

Perform the following steps to save temporary assertions:

1. Select **Tools** -> **Property Tools** -> **Add Temporary Assertion** command. The **Add Temporary Assertion** form opens.
2. Click **Load** to load the assertions and scope from a file with a `.tsva` extension as shown in the following figure.

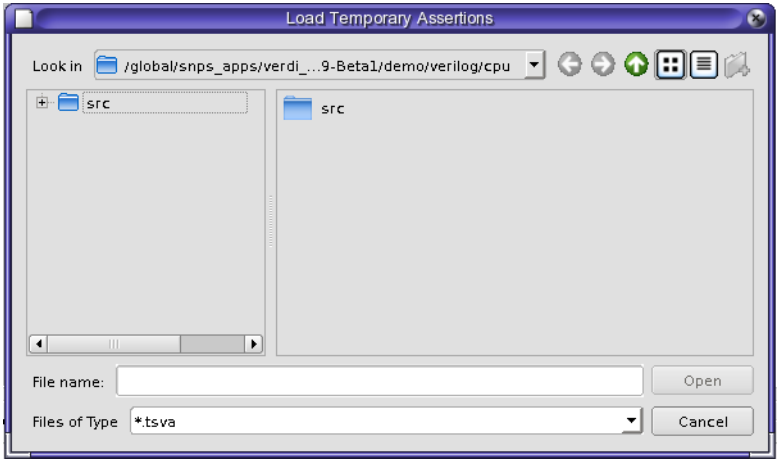


Figure: Load Temporary Assertion Form

Note the following:

The SVA evaluation of runtime assertions mechanism only supports the hierarchy name for those signals that belong to or are under the active scope. For example, as shown in the following figure, the active scope is CPU, ALU.signal_A is the allowed signal and system.MEM.MCU.signal_B is not.

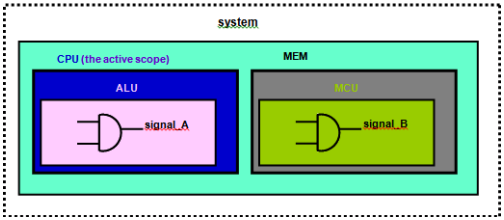


Figure: A Design Sample

- The SVA evaluation of runtime assertions mechanism only supports the assertion in a scope. Only one scope in the **Scope** text field is allowed.
- The SVA evaluation of runtime assertion mechanism does not support the following:
  - Immediate assertions
  - Property and sequence built-in functions
  - PSL related nodes
  - Using macros in the newly added SVA syntax
- The mechanism used for SVA evaluation of runtime assertions cannot commit an assertion with the same name in the same scope in the design.
- Previously committed assertions are removed after committing a new assertion.

Debug with Transactions

Before you begin this application, follow the instructions in the *Before You Begin* chapter.

Transactions are an important piece of abstraction in system design and debug. System design is in an early stage of the whole design process. Therefore, a powerful viewing mechanism for transactions is mandatory for system designers.

For testbench verification, if the entire system is to be verified, transaction level checking is efficient and easy to focus comparisons of system behavior against system specification. When an error is found in the transaction level, the signal level is then investigated.

This section covers the following topics:

- What is a Transaction?
- Generating Transaction Data
- View Transactions in nWave
- View Transactions in Transaction Table View Window

What is a Transaction?

Transactions are higher level abstractions of signal-level detailed activity and are organized into streams. Transaction streams can be dumped into FSDB format using dumping libraries provided by Synopsys and its partners or using the Open Transaction Interface (OTI) extension of the FSDB Writer API.

Streams hold transactions. Each transaction consists of a set of attributes and is independent of one another. That is, there is no such concept as "transaction type", as in SCV, even if the sets of attributes that constitute two transactions are the same.

When you create a transaction, you must follow the steps below:

1. Create a stream.
2. Create attributes.
3. Create a transaction.
4. Create relationships between existing transaction.

Generating Transaction Data

The transaction data can be obtained from the following sources.

Provided FSDB Dumpers

Dump transaction data from languages directly with native FSDB dumpers.

- SystemC/SCV -- supported OSCI and NCSC simulators.
- Specman/e
- SystemVerilog testbench in conjunction with simulator support (VCS, ModelSim)
- Vera

Refer to the *SystemC Linking* chapter in the *Linking Novas Files with Simulators and Enabling FSDB Dumping* manual for details on linking native FSDB dumpers, SystemC SCV, and HVL simulators.

Transaction IP Partners

Contact Denali (PCI-Express) or Spiratech (AMBA AXI, AHB) directly for details on dumping FSDB format from their available intellectual property (IP).

SVA Extraction

You can add SystemVerilog Assertions (SVA) constructs to your design code to represent transactions. The transactions can then be extracted from a signal level FSDB. Refer to the *Extracting Transactions Using SVA* section in *Appendix D* for more details.

FSDB Writer API and the Open Transaction Interface (OTI)

If you are unable to generate transaction data in FSDB format from any of the previously mentioned methods, you can use the Open Transaction Interface (OTI) extension of the FSDB writer API to dump transaction data.

SVTB Automatic Logging of OVM/UVM Component and Port Transactions

SVTB-based testbench environments are typically built on top of an SVTB verification library/methodology like OVM/UVM. Synopsys has leveraged the infrastructure provided in OVM/UVM to record component and port transactions flowing up and down the testbench into the FSDB file. The Synopsys mechanism uses the OVM/UVM's transaction recording capability to record OVM/UVM testbench transactions into the FSDB file for debug and analysis in the Verdi platform.

View Transactions in nWave

This tutorial familiarizes you with transaction viewing and search operations. All *nWave* manipulation functions (zoom, cursor, marker, re-size, etc.) are available with FSDB files containing transactions.

1. Change the directory to *<working_dir>/demo/transaction.*

   ```
   % cd <working_dir>/demo/transaction
   ```

2. Execute Verdi to import the FSDB file:

   ```
   > verdi -ssf ahb32.bus.fsdb -workMode hardwareDebug &
   ```

   The Verdi platform opens and the FSDB file is loaded.

NOTE: This tutorial only has an FSDB file that contains transactions and there is not a related design.

---

3. In the *nWave* frame, choose the **Signal -> Get Signals** command to open the *Get Signals* form.

   The transaction FSDB is loaded displaying *BusTop* as the top hierarchy and *MyAHB_1(_AHB_)* as the first hierarchical level which is for different protocols.

4. Click *MyAHB_1* to show the streams under this hierarchy.

   The results will be similar to the following example:

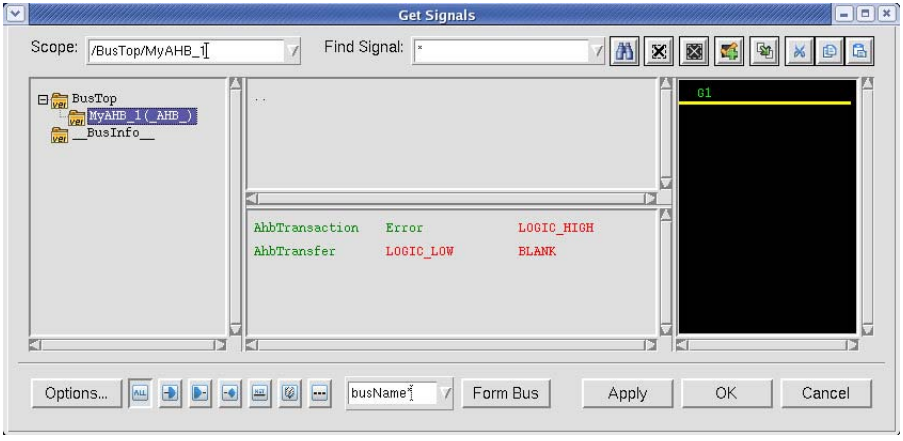


Figure: Get Signals - Displaying Streams

5. Select *AhbTransfer* and *AhbTransaction*, then click **OK**.

6. In the *nWave* frame, re-size the signal and value panes to more readily display the text and zoom in on the waveform pane to see the transaction details.
7. In the signal pan, click the *AhbTransaction* stream to select it.
8. Click the **Search Forward** icon (right arrow) in the toolbar to step through the transactions.

   The cursor moves to the begin time of each transaction.
9. Since there are more attributes than the default signal height can display, you can adjust the height by dragging the small grey line in the lower left corner of the stream name in the signal pane.
10. In the value pan, move the mouse cursor over the attributes to show the details in a tip.
11. With the *AhbTransaction* stream selected, choose the **Waveform -> Classic Transaction -> Expand Overlapping** command to make it easier to see the transaction overlap.
12. Choose the **Waveform -> Classic Transaction -> Shrink Overlapping** command to return to the overlapped view.
13. Click the **By:** icon in the *nWave* toolbar and choose the last option, **Transaction Attribute Values**.
14. In the *Set Search Attributes* form, enter "*BurstType*" for **Attributes** and "*incr 4*" for **Value**.

    The form will be similar to the following:

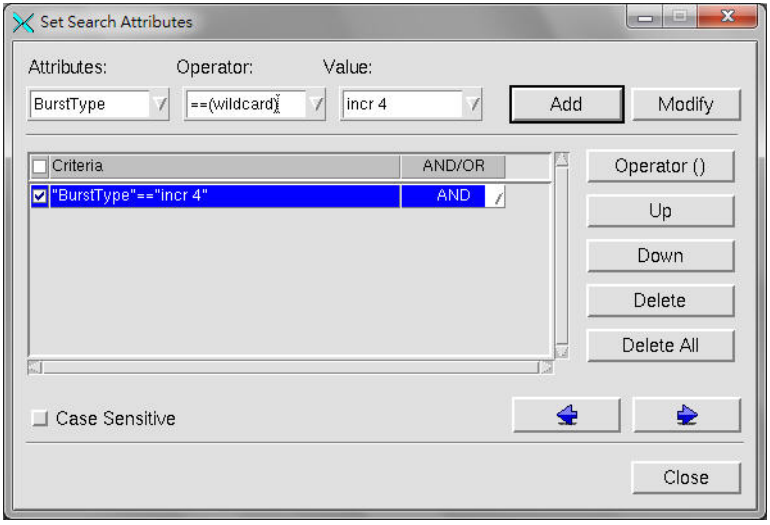


Figure: Set Search Attributes

15. Click the **Add** button.
16. Click the **Search Forward/Search Backward** icons on the *Set Search Attributes* form to locate a matching transaction at 4810000ps.
17. Click the transaction in the waveform pane at time 4,810,000ps, which is burst read of “incr 4” type.

    There will be 4 *AhbTransfer* burst read command transactions and 3 busy ones as the children of the selected transaction. The child transactions are highlighted in pink.

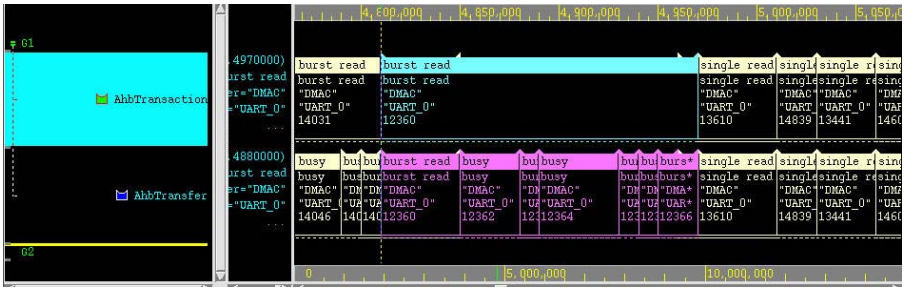


Figure: Search Results with Related Transactions

18. With the same transaction selected, right-click to open the right mouse button context menu and choose **Properties** to open the *Transaction Property* form which shows all the attributes and relationships for the selected transaction.

View Transactions in Transaction Table View Window

This tutorial will familiarize you with transaction viewing and search operations in a spreadsheet-like view.

1. Change the directory to *<working_dir>/demo/transaction.*

   ```
   % cd <working_dir>/demo/transaction
   ```

2. Execute Verdi to import the FSDB file:

   ```
   > verdi -ssf ahb32.bus.fsdb -workMode hardwareDebug &
   ```

   The *nTrace* main window and *nWave* frame open and the FSDB file is loaded.

NOTE: This tutorial only has an FSDB file that contains transactions and there is not a related design.

---

3. In the *nWave* frame, choose the **Signal -> Get Signals** command to open the *Get Signals* form.

The transaction FSDB is loaded displaying *BusTop* as the top hierarchy and *MyAHB_1(_AHB_)* as the first hierarchical level which is for different protocols.

4. Click *MyAHB_1* to show the streams under this hierarchy.
5. Select *AhbTransfer* and *AhbTransaction* and then click **OK**.

Add/Remove Transaction Streams

After loading a FSDB file with transaction data, you can view and manipulate the results in the *Transaction Table View* frame.

1. In the *nWave* frame, choose the **Tools -> Classic Transaction -> Analysis Window** command. The *Transaction Table View* frame will be opened as a new frame in the right half of the *nWave* frame, the FSDB file currently loaded in *nWave* will be the default in the *Transaction Table View* frame.
2. In the *Transaction Table View* frame, choose the **Stream -> Get Stream** command to open the *Select Stream* form.

   The form will be similar to the following:

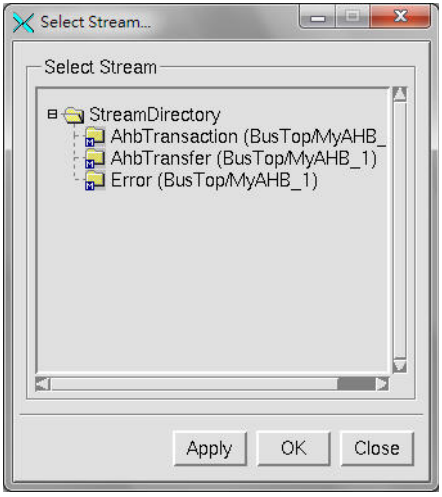


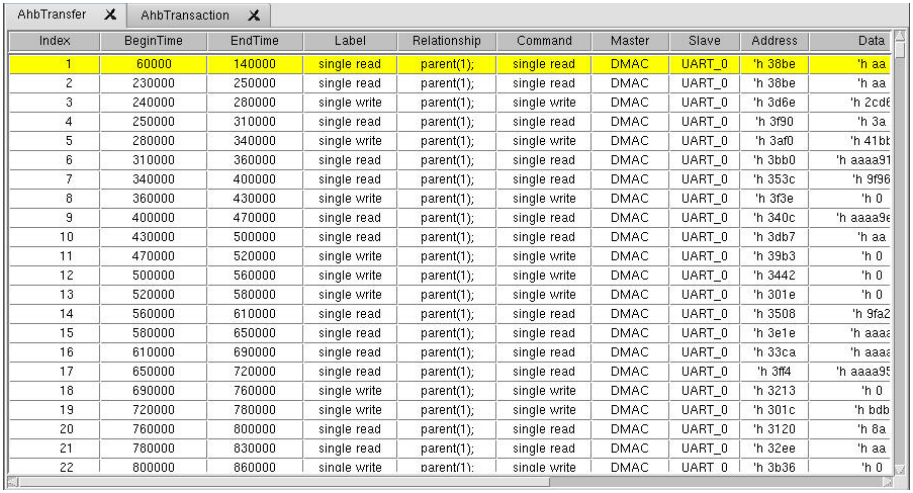
Figure: Select Stream Form

   All of the transaction streams available in the FSDB file will be listed in a tree-like format.

3. Double-click *AhbTransfer* to automatically add the stream to the *Transaction Table View* frame. The stream name changes to gray and is appended with a red dot.

4. Left-click to select *AhbTransaction* and click **OK** to add the stream and close the form.

   The *Transaction Table View* frame will be similar to the following:



AhbTransfer | AhbTransaction

Index	BeginTime	EndTime	Label	Relationship	Command	Master	Slave	Address	Data
1	60000	140000	single read	parent(1);	single read	DMAC	UART_0	'h 38be	'h aa
2	230000	250000	single read	parent(1);	single read	DMAC	UART_0	'h 38be	'h aa
3	240000	280000	single write	parent(1);	single write	DMAC	UART_0	'h 3d6e	'h 2cd8
4	250000	310000	single read	parent(1);	single read	DMAC	UART_0	'h 3f90	'h 3a
5	280000	340000	single write	parent(1);	single write	DMAC	UART_0	'h 3af0	'h 41bb
6	310000	360000	single read	parent(1);	single read	DMAC	UART_0	'h 3bb0	'h aaaa91
7	340000	400000	single read	parent(1);	single read	DMAC	UART_0	'h 353c	'h 9f96
8	360000	430000	single write	parent(1);	single write	DMAC	UART_0	'h 3f3e	'h 0
9	400000	470000	single read	parent(1);	single read	DMAC	UART_0	'h 340c	'h aaaa9e
10	430000	500000	single read	parent(1);	single read	DMAC	UART_0	'h 3db7	'h aa
11	470000	520000	single write	parent(1);	single write	DMAC	UART_0	'h 39b3	'h 0
12	500000	560000	single write	parent(1);	single write	DMAC	UART_0	'h 3442	'h 0
13	520000	580000	single write	parent(1);	single write	DMAC	UART_0	'h 301e	'h 0
14	560000	610000	single read	parent(1);	single read	DMAC	UART_0	'h 3508	'h 9fa2
15	580000	650000	single read	parent(1);	single read	DMAC	UART_0	'h 3e1e	'h aaaa
16	610000	690000	single read	parent(1);	single read	DMAC	UART_0	'h 33ca	'h aaaa
17	650000	720000	single read	parent(1);	single read	DMAC	UART_0	'h 3ff4	'h aaaa95
18	690000	760000	single write	parent(1);	single write	DMAC	UART_0	'h 3213	'h 0
19	720000	780000	single write	parent(1);	single write	DMAC	UART_0	'h 301c	'h bdb
20	760000	800000	single read	parent(1);	single read	DMAC	UART_0	'h 3120	'h 8a
21	780000	830000	single read	parent(1);	single read	DMAC	UART_0	'h 32ee	'h aa
22	800000	860000	single write	parent(1);	single write	DMAC	UART_0	'h 3b36	'h 0

Figure: Transaction Table View Frame with Streams Loaded

   There are two streams, *AhbTransfer* and *AhbTransaction*, in the *Transaction Table View* frame. Each stream has a tab of its own. You can select the stream name to see the details of the stream. The currently selected stream name is in blue. You can change the width of the columns by selecting the vertical line in the column header and dragging-left.

5. Left-click to select the *AhbTransaction*.
6. Choose the **Stream -> Close Stream** command. Note the stream has been removed from the *Transaction Table View* frame.

Merge Transaction Streams

You can also merge two or more streams in the *Transaction Table View* frame. When streams are merged you can search and filter all the transaction attributes simultaneously.

NOTE: The **Merge Stream** command only merges the transaction streams for viewing purposes; it does not effect the FSDB file.

---

1. In the *Transaction Table View* frame, choose the **Stream -> Merge Stream** command to open the *Merge Stream* form.

   The form will be similar to the following:

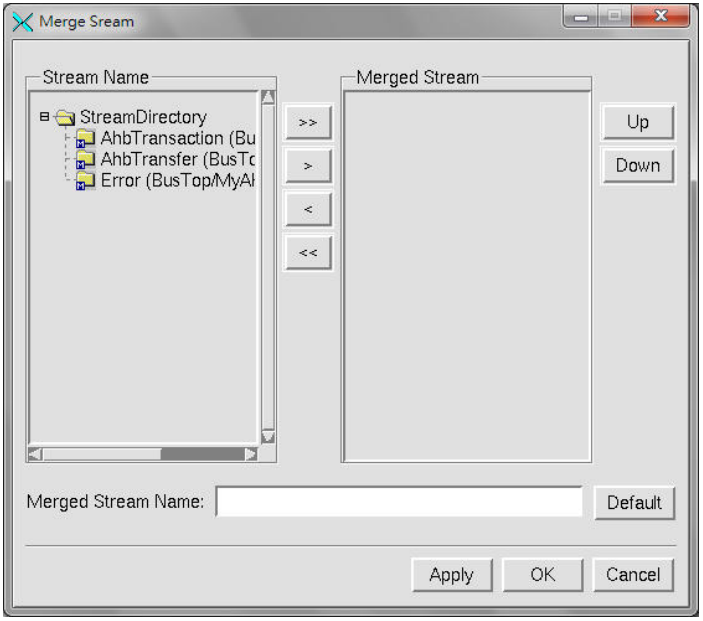
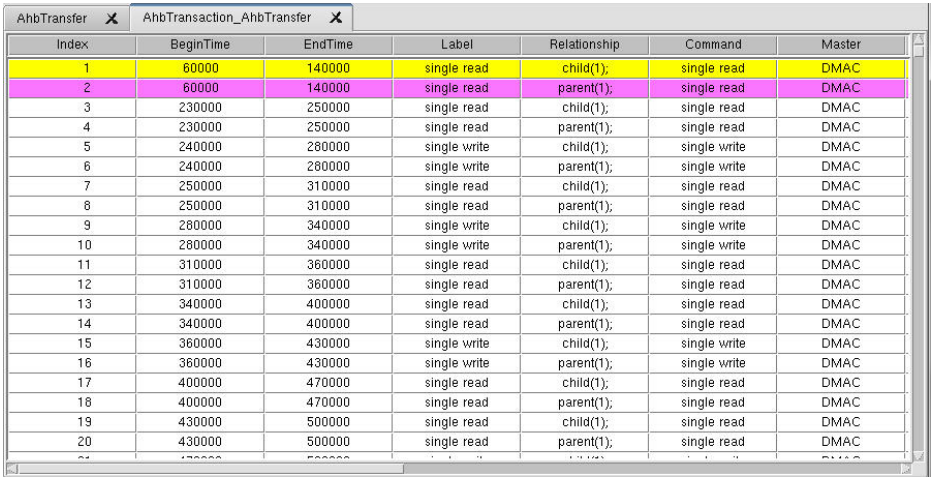


Figure: Merge Stream Form

All of the transaction streams available in the FSDB file will be listed in a tree-like format in the **Stream Name** column.

2. Click the >> button to move all streams to the **Merged Stream** column. After the stream is added, its name becomes gray with a red dot in the **Stream Name** column and can not be selected again.
3. Left-click to select the *Error* stream in the **Merged Stream** column.
4. Click the < button to move the selection back to the **Stream Name** column. The stream name is changed to black and is selectable again.
5. Left-click to select the *AhbTransaction* stream in the **Merged Stream** column.
6. Click the **Default** button to automatically generate the merged stream name which will consist of each stream name linked with an underscore.
7. Click the **OK** button.

   The *Transaction Table View* frame will be similar to the following:



AhbTransfer | AhbTransaction_AhbTransfer

Index	BeginTime	EndTime	Label	Relationship	Command	Master
1	60000	140000	single read	child(1);	single read	DMAC
2	60000	140000	single read	parent(1);	single read	DMAC
3	230000	250000	single read	child(1);	single read	DMAC
4	230000	250000	single read	parent(1);	single read	DMAC
5	240000	280000	single write	child(1);	single write	DMAC
6	240000	280000	single write	parent(1);	single write	DMAC
7	250000	310000	single read	child(1);	single read	DMAC
8	250000	310000	single read	parent(1);	single read	DMAC
9	280000	340000	single write	child(1);	single write	DMAC
10	280000	340000	single write	parent(1);	single write	DMAC
11	310000	360000	single read	child(1);	single read	DMAC
12	310000	360000	single read	parent(1);	single read	DMAC
13	340000	400000	single read	child(1);	single read	DMAC
14	340000	400000	single read	parent(1);	single read	DMAC
15	360000	430000	single write	child(1);	single write	DMAC
16	360000	430000	single write	parent(1);	single write	DMAC
17	400000	470000	single read	child(1);	single read	DMAC
18	400000	470000	single read	parent(1);	single read	DMAC
19	430000	500000	single read	child(1);	single read	DMAC
20	430000	500000	single read	parent(1);	single read	DMAC

Figure: Merged Stream in the Transaction Table View Frame

If the different streams have transactions at the same time, both will be displayed.

Manipulate the Stream View

There are several ways to manipulate the streams in the *Transaction Table View* frame. You can change which columns (attributes) are displayed and in what order. You can also filter the transactions based on one or two attribute conditions.

Set the Cursor/Marker

In this example, you'll set the cursor/marker position in the *Transaction Table View* frame and learn how to synchronize it with the other Verdi frames.

1. In the *Transaction Table View* frame, select the *AhbTransfer* stream.
2. Left-click anywhere on the row for **Index 13** to set the cursor time. The selected row is highlighted in yellow.
3. Scroll until you can see **Index 25**.
4. Middle-click anywhere on the row for **Index 25** to set the marker time. The selected row is highlighted in red.
5. Choose the **View -> Sync Cursor Time** command to synchronize the cursor globally.
6. Left-click anywhere on the row for **Index 18** to set the cursor time. Note the cursor time changes in the *nWave* frame as well. If you had a design loaded and active annotation enabled, the cursor time would change in the source code frame and *nSchema* frame as well.

Change the Column (Attribute) Display

In this example, you'll select some columns (attributes) to remove from the display and re-order the remaining columns.

1. In the *Transaction Table View* frame, select the *AhbTransfer* stream.
2. Choose the **View -> Column Configuration** command to open the *Config Bus Table* form.

   The form will be similar to the following:

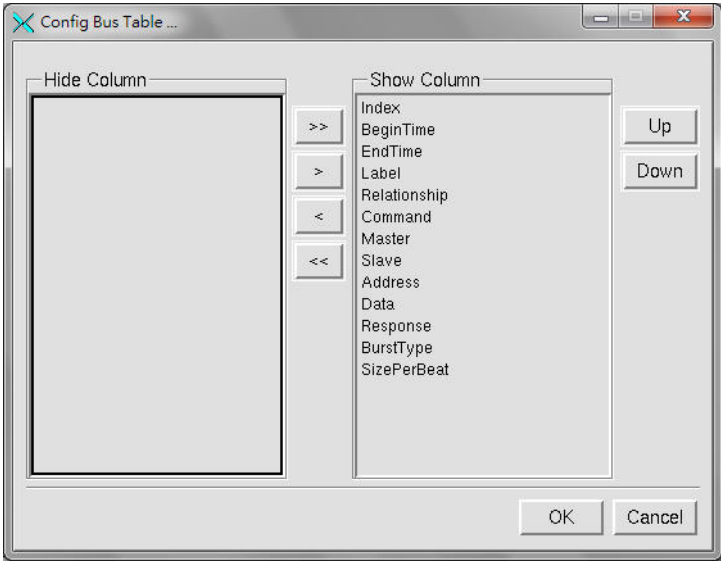


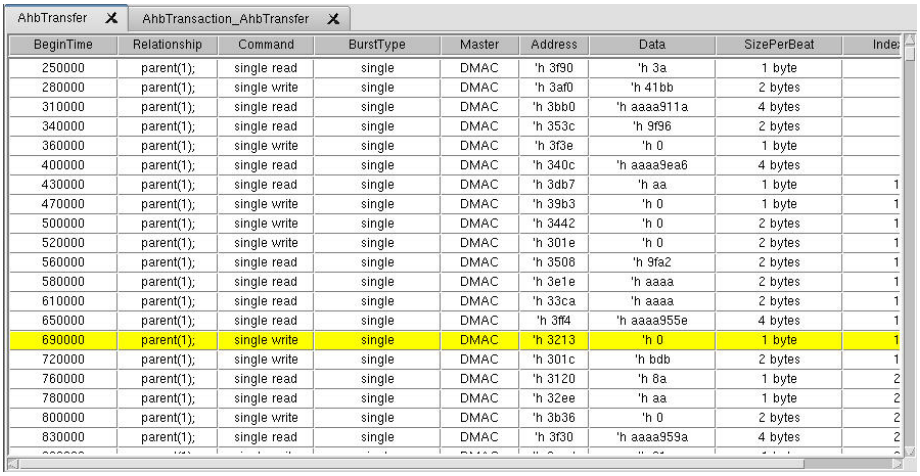
Figure: Config Bus Table Form

   By default, all the columns (attributes) will be listed in the **Show Column** section.

3. Select **Label** in the **Show Column** section.
4. Click the < button to move it to the **Hide Column** section.
5. Repeat the previous steps for **Response**, **Slave**, and **EndTime** individually. Only one attribute can be selected at a time.
6. In the **Show Column** section, select **Index**.
7. Click the **Down** button multiple times until **Index** is at the bottom of the list.
8. In the **Show Column** section, select **BurstType**.

9. Click the **Up** button multiple times until **BurstType** is located below **Command**.
10. Click **OK**.

    The *Transaction Table View* frame will be updated as follows:



AhbTransfer | AhbTransaction_AhbTransfer

BeginTime	Relationship	Command	BurstType	Master	Address	Data	SizePerBeat	Inde
250000	parent(1);	single read	single	DMAC	'h 3f90	'h 3a	1 byte	
280000	parent(1);	single write	single	DMAC	'h 3af0	'h 41bb	2 bytes	
310000	parent(1);	single read	single	DMAC	'h 3bb0	'h aaaa911a	4 bytes	
340000	parent(1);	single read	single	DMAC	'h 353c	'h 9f96	2 bytes	
360000	parent(1);	single write	single	DMAC	'h 3f3e	'h 0	1 byte	
400000	parent(1);	single read	single	DMAC	'h 340c	'h aaaa9ea6	4 bytes	
430000	parent(1);	single read	single	DMAC	'h 3db7	'h aa	1 byte	1
470000	parent(1);	single write	single	DMAC	'h 39b3	'h 0	1 byte	1
500000	parent(1);	single write	single	DMAC	'h 3442	'h 0	2 bytes	1
520000	parent(1);	single write	single	DMAC	'h 301e	'h 0	2 bytes	1
560000	parent(1);	single read	single	DMAC	'h 3508	'h 9fa2	2 bytes	1
580000	parent(1);	single read	single	DMAC	'h 3e1e	'h aaaa	2 bytes	1
610000	parent(1);	single read	single	DMAC	'h 33ca	'h aaaa	2 bytes	1
650000	parent(1);	single read	single	DMAC	'h 3ff4	'h aaaa955e	4 bytes	1
690000	parent(1);	single write	single	DMAC	'h 3213	'h 0	1 byte	1
720000	parent(1);	single write	single	DMAC	'h 301c	'h bdb	2 bytes	1
760000	parent(1);	single read	single	DMAC	'h 3120	'h 8a	1 byte	2
780000	parent(1);	single read	single	DMAC	'h 32ee	'h aa	1 byte	2
800000	parent(1);	single write	single	DMAC	'h 3b36	'h 0	2 bytes	2
830000	parent(1);	single read	single	DMAC	'h 3f30	'h aaaa959a	4 bytes	2

Figure: Modified Attribute Display for AhbTransfer Stream

Note four columns have been removed from the display and the remaining columns have been re-ordered. The **Index** column is now the right-most column and **BurstType** is next to **Command**.

11. Left-click the **Command** column to sort by the command attribute types.

Filter the Transactions

In this example, you will filter the transactions based on certain attributes.

1. In the *Transaction Table View* frame, select the *AhbTransfer* stream.
2. Choose the **View -> Filter/Colorize** command to open the *Filter/Colorize* form.
3. Toggle the **Attributes:** field and select **Command**.

   Toggle the **Operator** to **==(wildcard)** and enter *single write* in the **Value:** field, then click **Add** button. The form will be similar to the following:

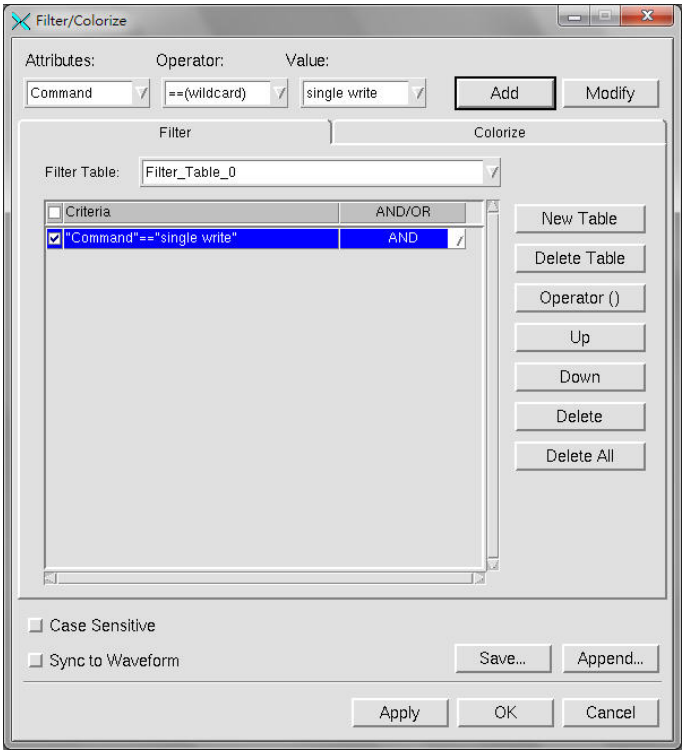


Figure: Filter/Colorize Form - Command = single write

4. Click the **Apply** button.

   The *Transaction Table View* frame will be updated to display transactions whose command attribute is of type single write, similar to the following:

AhbTransfer | AhbTransaction_AhbTransfer

BeginTime	Relationship	Command	BurstType	Master	Address	Data	SizePerBeat	Inde
240000	parent(1);	single write	single	DMAC	'h 3d6e	'h 2cd6	4 bytes	
280000	parent(1);	single write	single	DMAC	'h 3af0	'h 41bb	2 bytes	
360000	parent(1);	single write	single	DMAC	'h 3f3e	'h 0	1 byte	
470000	parent(1);	single write	single	DMAC	'h 39b3	'h 0	1 byte	1
500000	parent(1);	single write	single	DMAC	'h 3442	'h 0	2 bytes	1
520000	parent(1);	single write	single	DMAC	'h 301e	'h 0	2 bytes	1
690000	parent(1);	single write	single	DMAC	'h 3213	'h 0	1 byte	1
720000	parent(1);	single write	single	DMAC	'h 301c	'h bdb	2 bytes	1
800000	parent(1);	single write	single	DMAC	'h 3b36	'h 0	2 bytes	2
880000	parent(1);	single write	single	DMAC	'h 3cad	'h 31	1 byte	2
920000	parent(1);	single write	single	DMAC	'h 3cd0	'h 366b	4 bytes	2
930000	parent(1);	single write	single	DMAC	'h 3eb6	'h 0	2 bytes	2
1180000	parent(1);	single write	single	DMAC	'h 3902	'h 0	1 byte	3
1240000	parent(1);	single write	single	DMAC	'h 3cd4	'h 13e9	4 bytes	3
1260000	parent(1);	single write	single	DMAC	'h 33ea	'h 0	2 bytes	3
1280000	parent(1);	single write	single	DMAC	'h 30bc	'h 5c67	4 bytes	3
1450000	parent(1);	single write	single	DMAC	'h 3657	'h 0	1 byte	4

Figure: Filter Results for Command single write

At this point you have several options. You can sort the current results by clicking another column header or you can further reduce the display by specifying another filter or you can restore the stream and start over. Let's specify another filter.

5. In the *Filter/Colorize* form (which should still be open unless you closed it), toggle the **Attributes:** field and select **SizePerBeat**.
6. Toggle the **Operator:** to **>=** and enter *2 byte* in the **Value:** field, and click **Add** button.

   The form will be similar to the following:

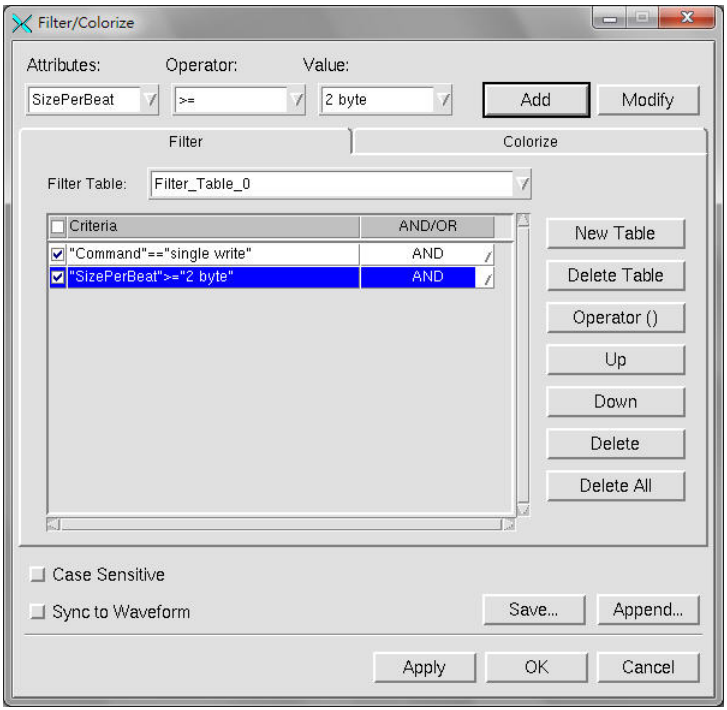
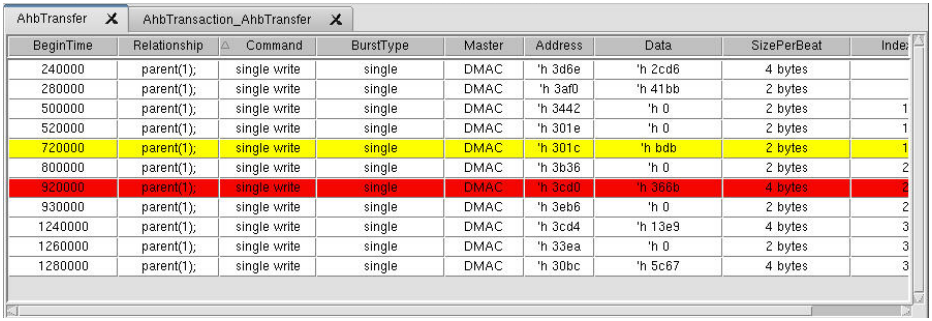


Figure: Filter/Colorize Form - SizePerBeat >= 2byte

7. Click the **Apply** button.

   The *Transaction Table View* frame will be updated to display transactions whose command attribute is of type single write and whose **SizePerBeat** attribute value is greater than or equal to 2 bytes, similar to the following:



AhbTransfer | AhbTransaction_AhbTransfer

BeginTime	Relationship	Command	BurstType	Master	Address	Data	SizePerBeat	Inde
240000	parent(1);	single write	single	DMAC	'h 3d6e	'h 2cd6	4 bytes	
280000	parent(1);	single write	single	DMAC	'h 3af0	'h 41bb	2 bytes	
500000	parent(1);	single write	single	DMAC	'h 3442	'h 0	2 bytes	1
520000	parent(1);	single write	single	DMAC	'h 301e	'h 0	2 bytes	1
720000	parent(1);	single write	single	DMAC	'h 301c	'h bdb	2 bytes	1
800000	parent(1);	single write	single	DMAC	'h 3b36	'h 0	2 bytes	2
920000	parent(1);	single write	single	DMAC	'h 3cd0	'h 366b	4 bytes	2
930000	parent(1);	single write	single	DMAC	'h 3eb6	'h 0	2 bytes	2
1240000	parent(1);	single write	single	DMAC	'h 3cd4	'h 13e9	4 bytes	3
1260000	parent(1);	single write	single	DMAC	'h 33ea	'h 0	2 bytes	3
1280000	parent(1);	single write	single	DMAC	'h 30bc	'h 5c67	4 bytes	3

Figure: Filter Results for Command single write with SizePerBeat >= 2 bytes

Enable the **Sync to Waveform** option in the *Filter/Colorize* form and then the transaction stream in the waveform will also be filtered.

8. Click the **Sync. Signal Selection Enabled** icon (see left) on both the *Transaction Table View* frame and the *nWave* frame to synchronize the views.
9. In the *Transaction Table View* frame, select the row containing Index 37. The waveform will automatically update and select the related transaction. You can also select a transaction in the waveform and the appropriate row will be highlighted.

You can continue sorting the current results by clicking another column header or you can further reduce the display by specifying another filter or you can restore the stream and start over. You can restore the stream and start over.

Generate Statistics

In addition to viewing and manipulating the transactions in a spreadsheet-like view, you can generate a variety of statistics for the stream.

1. In the *Transaction Table View* frame, select the *AhbTransaction_AhbTransfer* merged stream.

NOTE: Although this example will use the entire merged stream, you can filter the stream first and then generate statistics based on the reduced display.

---

2. Choose the **Tools -> Statistics Window** command to open the *Perform Statistical Calculation* form.

   The form will be similar to the following:

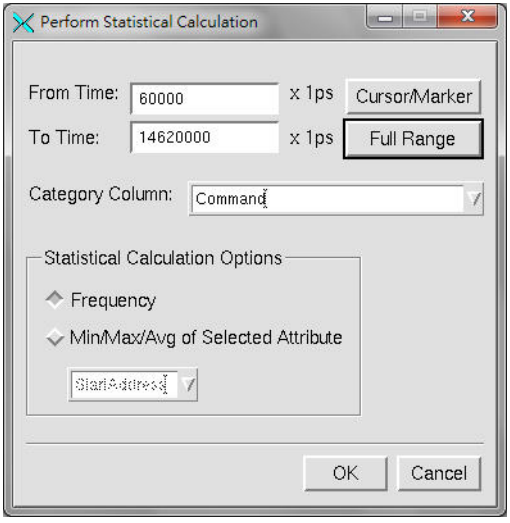


Figure: Perform Statistical Calculation Form

You have several options for setting up the form. In this example you want to view the frequency of **BurstType** for the entire simulation range.

3. Click the **Full Range** button to automatically enter the from and to times.
4. Toggle the **Category Column** field and select **BurstType**.
5. Click **OK**.

   A *Statistics* frame similar to the following will open as a new tab in the same location as the *Transaction Table View* frame.

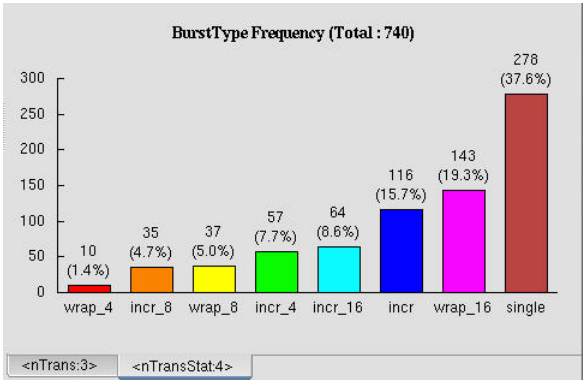


Figure: Bar Chart for BurstType

For the stream combination, you can easily see the frequency of the burst types. At this point, you can capture the results in PNG format. You can also change the view to a pie chart or table, or duplicate the window.

6. In the *Statistics* frame, choose the **View -> Pie Chart** command.
   The frame is updated similar to the following.

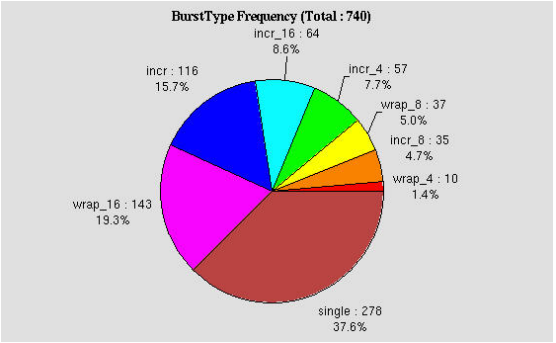


Figure: Bar Chart for BurstType

7. Choose the **File -> Close** command to close the *Statistics* frame.
   You can generate more statistics for different attribute types.
8. In the *nTrace* main window, choose the **File -> Exit** command to close the Verdi session.

Appendix A: Supported Waveform Formats

Overview

In addition to the FSDB waveform format, the following formats are supported:

- VCD (Value Change Dump)
- EVCD (Extended Value Change Dump)
- Analog - Powermill, Spice

This appendix covers the following topics:

- Fast Fourier Transformers (FFT)
- EVCD
- Analog Waveform Example

Fast Fourier Transformers (FFT)

nWave provides the capability of viewing and analyzing analog signals in the frequency domain. An FFT window in *nWave* is used to display and process frequency waveforms. *nWave* can process analog signals through FFT to get the frequency results.

Getting Data from Analog Signal

After analog signals are imported into *nWave*, choose the **Analog -> FFT** command to access the FFT window.

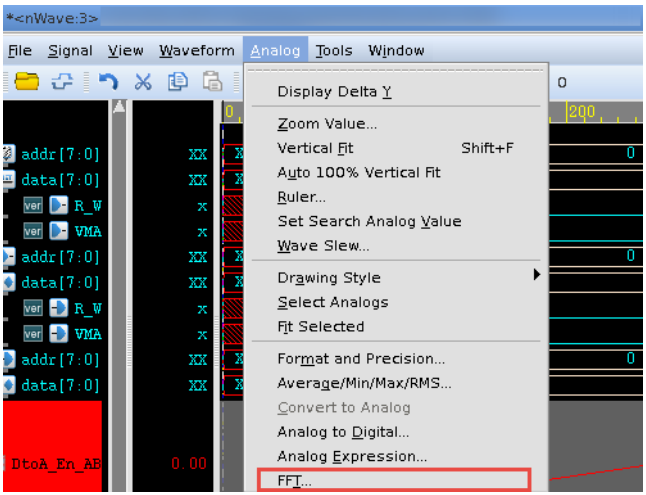


Figure: Open FFT from nWave

The FFT window is created without any waveforms displayed.

Choose the **Signal -> Add FFT Signal** command in the FFT window.

The *FFT Input Parameters* form displays, as shown below. You can specify parameters for FFT in this form.

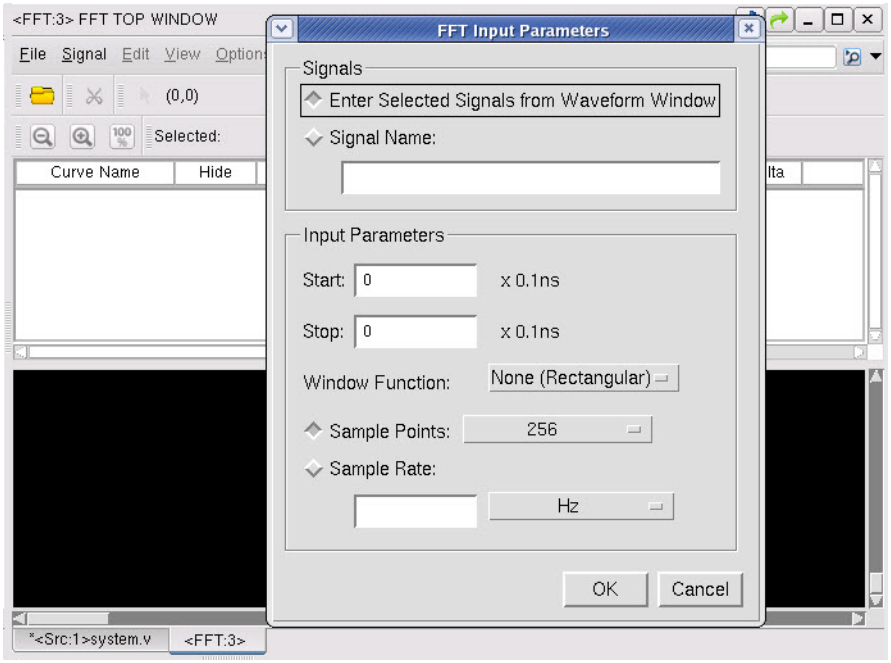


Figure: Add FFT Signal Form

Target signal can be specified through *nWave* window selection or drag from *nWave* into the **Signal Name** text box. The **Start** and **Stop** times have to be specified to define the range of the FFT process.

Seven types of window functions (rectangular, Blackman, Hamming, Hanning, Parzen, Triangular and Welch) can be used in *nWave*. The resolution of the FFT result can be specified with sample point number or sample rate.

Different FFT methods on the same target signal can be compared in the FFT window.

Select the signal in the FFT window, then use the **Edit -> Calculate Selected** command. The different parameters are specified in the *FFT Input Parameters [2]* form (shown below), and the re-calculated waveform are displayed for comparison.

Enable the **Override** option to overwrite the original result.

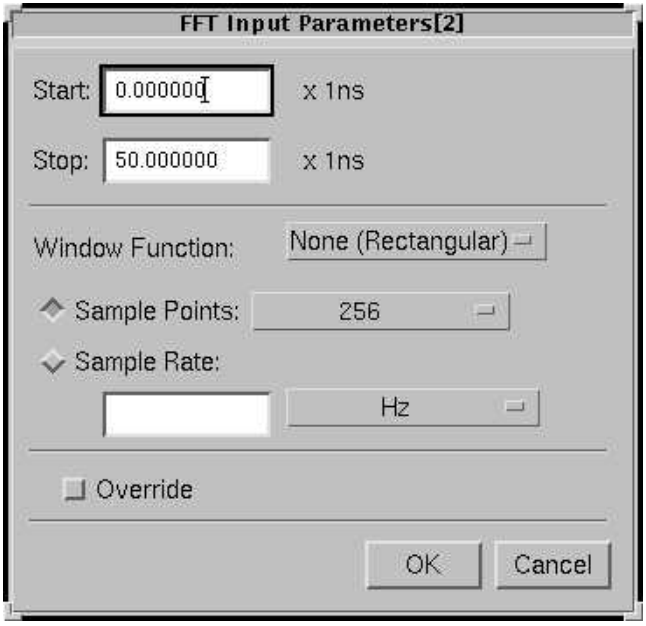


Figure: FFT Input Parameters [2] Form

The FFT results of analog waveforms can be saved in the FSDB, and restored for use later. You can also export the FFT result to an ASCII format text file.

Data Manipulation in FFT Window

The FFT window provides several methods to change data display formats for analysis under the **Options -> Preferences** command.

In the *Preferences* form, the *Display Option* tab contains several setting for data manipulation, as shown below:

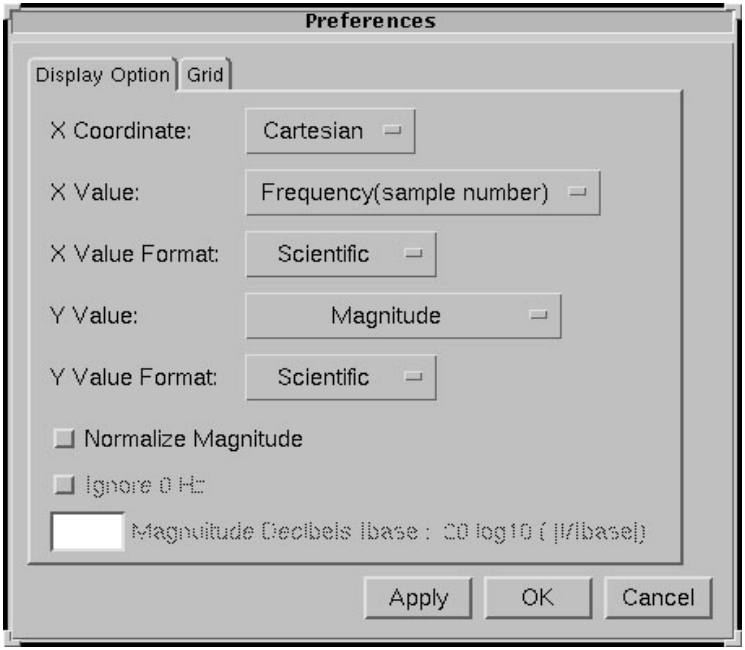


Figure: Preferences Form

The Y-axis can be changed to **Magnitude decibels (dB)** as conventional notation. Changing the Ibase value can modify the offset value of Y-axis in dB.

For example, changing Ibase value from 1 to 10 means the dB offset value changed from 0 to -20. If Ibase is specified as 0.1, the offset value is changed to +20.

The Ibase value only affects analog waveform FFT results.

The following figures are example FFT waveforms fixed are -40 dB:

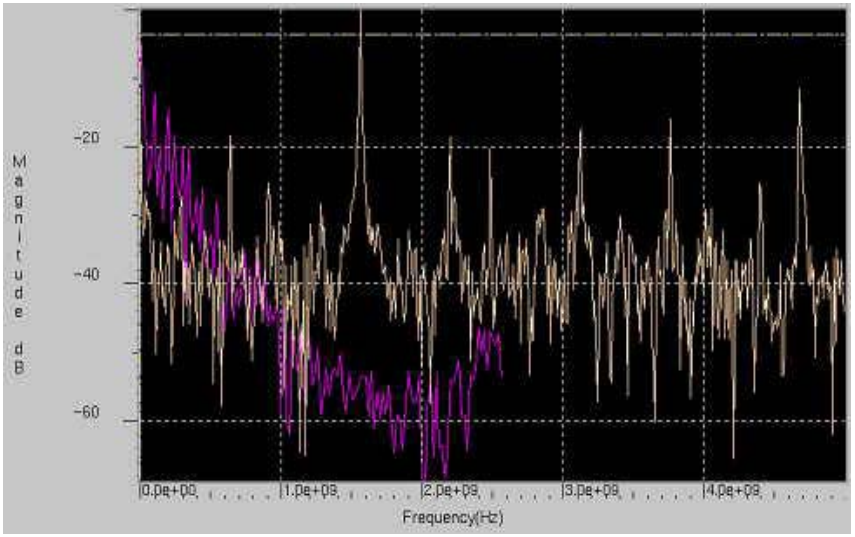


Figure: FFT Waveform

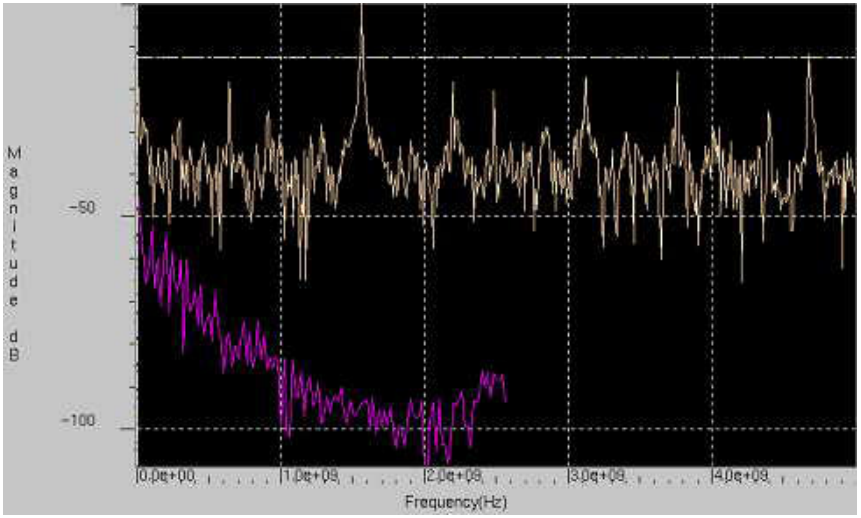


Figure: FFT Waveform

EVCD

A EVCD (Extended VCD) file saves the instance ports' logic and driver information. It is valuable if it can be read as a waveform and back annotated to the design source code and schematics. The capabilities for supporting EVCD are listed below.

- Convert EVCD to FSDB -- You can convert your dumped EVCD to FSDB by *vfast*.
- The converted FSDB retains full range of values to represent logic and driver information. The values include 25 port values, 8 strengths for 0 and 8 strengths for 1.
- *nWave* displays full range of values to represent logic and driver information -- *nWave* can display 25 port values, 8 strengths for 0 and 8 strengths for 1. In total, *nWave* can display 1600 patterns (25*8*8) for EVCD. Through the *Preferences* form (**Waveform** folder, **Extended VCD** page), you can configure all of the patterns as you like.

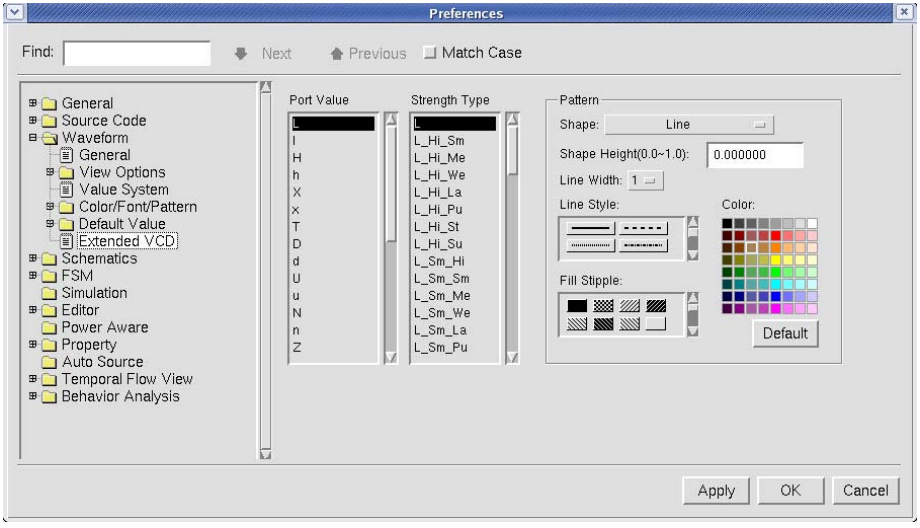


Figure: EVCD Preferences Form

- Map the logic and driver information to standard VCD value -- For bus ports, values are mapped to IEEE standard Verilog value space (0,1,z.x). To see each individual port's value, you can expand the bus port to single bit ports as shown below:

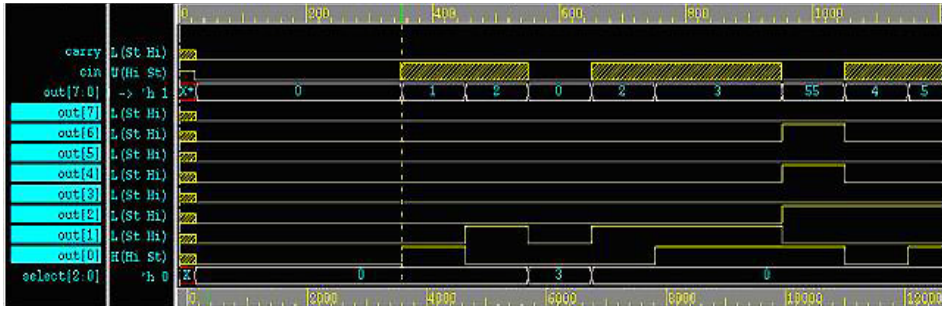


Figure: Example Expanded Bus Ports

- Map values of all single bit ports to IEEE standard Verilog value space by turning on the **Normalize EVCD Display Value** option under the **Waveform** folder -> **General** page of the *Preferences* form (invoked with the **Tools -> Preferences** command).

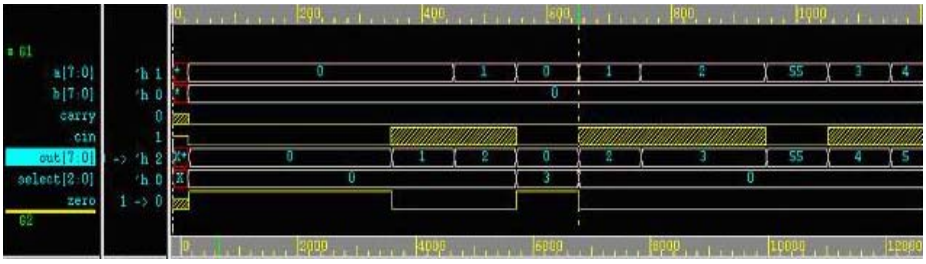


Figure: Example Normalized EVCD Display Value

- Support search value for any changes, rising and falling -- The search value for bus ports is the same as VCD since their values have been mapped to IEEE standard Verilog value space. When you search value for single bit ports, the Verdi platform maps them to IEEE standard Verilog value space internally. For example, the Verdi platform maps L(str0,str1)->H(str0,str1) to 0->1 and recognize it as a rising change.
- Annotate EVCD value or the mapped VCD value to *nTrace* and *nSchema* -- In *nTrace*, port value is annotated. In *nSchema*, port value and direction are annotated.

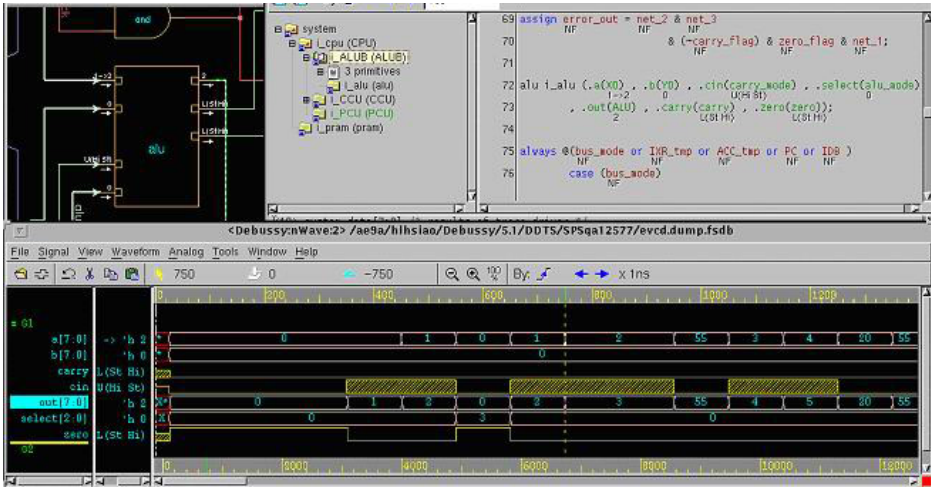


Figure: Annotate EVCD / Mapped VCD Value to nTrace and nSchema

The following features are not supported:

- Utilities (*fsdb2vcd*, *fsdbextract*, *fsdbmerge*, *fsdbReport*) for converted FSDB.
- *Trace-X* and *List-X* for the converted FSDB.

Analog Waveform Example

This section covers the following topics:

- View the Analog Waveform
- Manipulate the Analog Waveform
- View Different Simulation Results in the Same Window
- Overlap Analog Signals from Different Simulation Results

View the Analog Waveform

1. Change your context to the analog sub-directory, which is where all of the demo source code files are located:

   ```
   % cd <working_dir>/demo/analog
   ```

2. Start *nWave*.

   ```
   % nWave &
   ```

3. Choose the **File -> Open** command, and type *.* in the **Filter** text box of the *Open Dump File* form.
4. Select the file *PowerMill.out* and click the **Add** and then the **OK** button.
5. Click the **Yes** button on the *Question* dialog window for direct read.
6. Choose the **Signal -> Get All Signals** command.
7. Click the **Yes** button on the *Confirmation* dialog window.

   An analog waveform is displayed in *nWave*, as shown below:

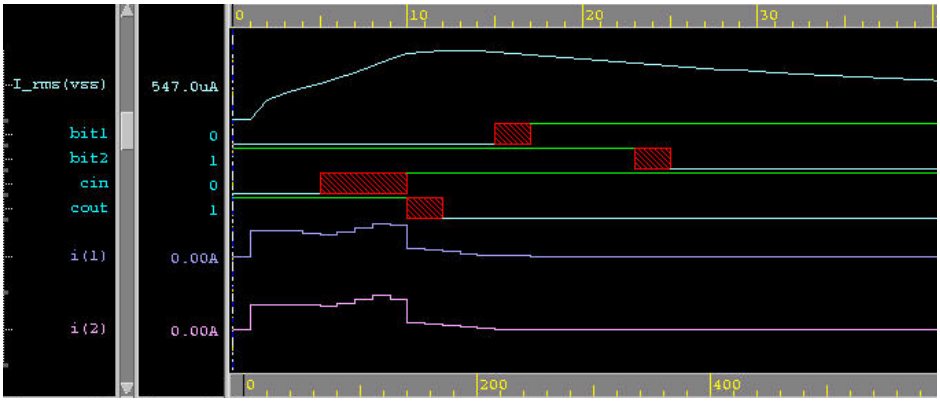


Figure: Analog/Digital Waveforms in nWave

The *nWave* frame displays analog waveforms differently from digital waveforms in two ways:

- *nWave* makes analog signals taller than digital signals.
- *nWave* uses different colors to display each newly added analog signal.

Manipulate the Analog Waveform

Similar to the way you change display formats for digital signals, you can change the format for analog signals by clicking-right on the value pane to change to the desired format. The supported analog display formats are: V, mV, A, mA, and uA.

NOTE: Choose the **Analog -> Format & Precision** command, to open the *Format* form and set the **Analog Format** to **Scientific** or **Engineering** first.

Change the Signal Height

1. Select the signals *i(1)* and *i(2)*.

2. Choose the **Waveform -> Height** command, and enter *200* pixels for the new signal height.

   The signal height increases, and only one signal can be seen in the *nWave* window.

3. Maximize the window to see both signals.

NOTE: *nWave* limits the minimum signal height to the signal name height and the maximum signal height to the height of the waveform window. After you resize an *nWave* window, *nWave* changes the signal height automatically if the signal height is taller than the waveform pane.

Display the Analog Ruler

1. Choose the **Analog -> Ruler** command to display the *Analog Ruler* form.
2. Enter *10000* in the **Grid Step** field to display the vertical grid step at every 10000 value unit.
3. Click **Apply** to display the ruler on the selected signals.
4. Click the **Cancel** button to close the *Analog Ruler* form.

View Different Simulation Results in the Same Window

You can open multiple simulation result files in the same *nWave* window. This capability is especially useful for analyzing analog waveforms from different simulation runs or different simulators. For example, you can mix your Verilog VCD waveforms with waveforms from PowerMill.

1. Highlight the existing signals in *nWave* (use the **Signal -> Select All** command), and use the **Cut** icon to remove them from the display.
2. Choose the **Signal -> Get Signals** command.
3. Select the signals *i(cin)* and *i(node1)*, and click the **OK** button.
4. Select group *G1*.
5. Right-click and choose **Rename** to change the group name *G1* to *PwrMill*.
6. Set the signal cursor under group *G2*.
7. Choose the **File -> Open** command to open the *Open Dump File* form.
8. Select the file *SmartSpice.out,* and click the **Add** and then the **OK** button.
9. Click the **OK** button on *Information* dialog window.
10. Choose the **Signal -> Get Signals** command.
11. Select the signals *i(cin)* and *i(node1)*, and click the **OK** button.
12. Select group *G2*.

13. Right-click and choose **Rename** to change the group name *G2* to *SmtSpice*.
    You should now see two simulation results in the same window.
14. If you want to add more signals from the first open file, choose the **File -> Set Active** command to switch the current active file. *nWave* places no logical limit on the number of files you can open.

Overlap Analog Signals from Different Simulation Results

1. Set the signal cursor under group *G3*.
2. Select the signal *i (node1)* from group *PwrMill* and the signal *i (node1)* from group *SmtSpice* (hold the <Ctrl> key to select multiple non-contiguous signals).
3. Choose the **Signal -> Overlay** command to overlap the two signals.
4. Choose the **Analog -> Ruler** command to turn on the ruler.

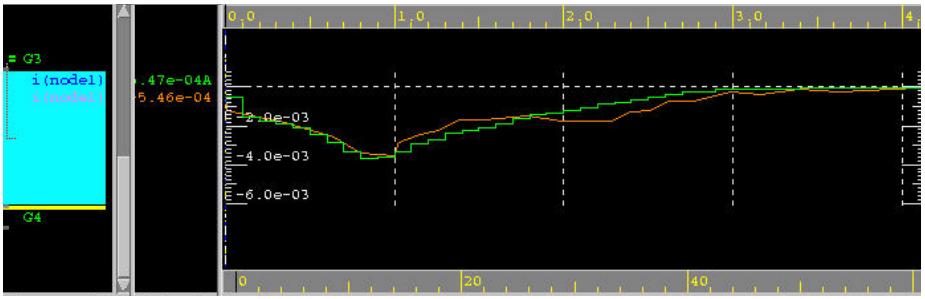


Figure: Overlapped Analog Waveforms

Now you can analyze the differences between these two signals.

Appendix B: Supported FSM Coding Styles

Overview

Finite State Machine (FSM) Coding is very common in RTL design. The Verdi platform extracts FSM from the source code automatically and provides a visual state diagram and state animation to trace whole FSM actions. It is very helpful for an IC designer to analyze and debug an RTL design, especially for large FSMs. There are various kinds of FSM coding styles. The Verdi platform supports the following FSM coding styles:

- *One-Process (Always)*
- *Two-Process (Always)*
- *One-Hot Encoding*
- *Shift Arithmetic Operation*
- *Case-Statement vs. If-Statement*
- *Gate-Like FSM*
- *Next_State = signal*
- *Next_State = Current_State + N*
- *VHDL Record Type*

The following sections give Verilog and VHDL code examples for the different coding styles listed above.

One-Process (Always)

The definition of a one-process FSM is that all of its functions are specified in one VHDL process or one Verilog `always` statement. The following sections contain examples of one-process FSMs:

- *Example 1 - Verilog (one_process.v)*
- *Example 2 - VHDL (one_process.vhd)*

The functions of these two FSMs are equal. In *nState*, the state diagrams of these two FSMs are identical.

Example 1 - Verilog (one_process.v)

```
module FSM1_BAD (Clock, SlowRAM, Read, Write);
      input  Clock, SlowRAM;
      output Read, Write;
      reg Read, Write;
      integer State;
   always @(posedge Clock)
     begin: SEQ_AND_COMB
       case (State)
         0 :
           begin
             Read  = 1;
             Write = 0;
             State = 1;
           end
         1 :
           begin
             Read  = 0;
             Write = 1;
             if (SlowRAM == 1)
               State = 2;
             else
               State = 0;
           end
         2 :
           begin
             Read  = 0;
             Write = 0;
             State = 0;
           end
       endcase
     end
 endmodule
```

Refer to *Figure: Verilog (one_process.v)* for the *nState* diagram.

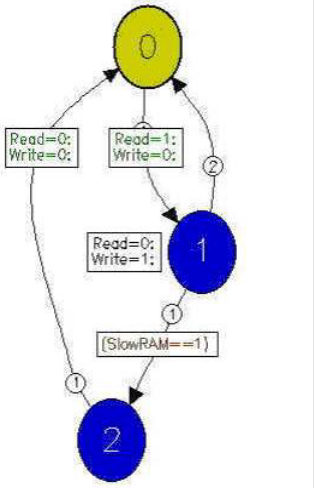


Figure: Verilog (one_process.v)

Example 2 - VHDL (one_process.vhd)

```
Library IEEE;
Use IEEE.STD_Logic_1164.all;
Entity FSM1_BAD is
   port (Clock:      in  std_logic;
         SlowRAM:    in  std_logic;
         Read,Write: out std_logic);
End entity FSM1_BAD;
Architecture RTL of FSM1_BAD is
Begin
    SEQ_AND_COMB: process
            variable State: integer;
    begin
      wait until rising_edge(Clock);
      case State is
         when 0=>
            Read  <= '1';
            Write <= '0';
            State := 1;
         When 1=>
            Read  <= '0';
            Write <= '1';
            if (SlowRAM = '1') then
               State := 2;
            Else
               State := 0;
            end if;
         when 2=>
```

```
            Read  <= '0';
            Write <= '0';
            State := 0;
         When others=> null;
      end case;
   end process SEQ_AND_COMB;
end architecture RTL;
```

Refer to *Figure: VHDL (one_process.vhd)* for the *nState* diagram.

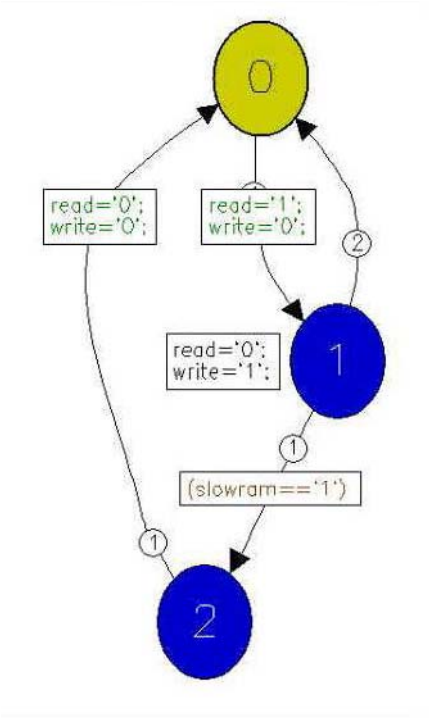


Figure: VHDL (one_process.vhd)

Two-Process (Always)

In two-process FSM, the FSM is split into a combinational circuit and a sequential circuit. The combinational circuit of the FSM is written in one process statement and the sequential circuit is written in the other process statement. Synopsys strongly recommends using this type of FSM. The following sections contain examples of two-process FSMs:

- Example 1 - *Verilog (two_process.v)*
- Example 2 - VHDL (*two_process.vhd)*

Example 1 - *Verilog (*two_process.v*)*

```
module FSM1_GOOD (Clock, Reset, SlowRAM, Read, Write);
      input  Clock, Reset, SlowRAM;
      output Read, Write;
      reg Read,Write;
      reg  [1:0] CurrentState, NextState;
 always @(posedge Clock)
     begin: SEQ
       if (Reset)
         CurrentState = 0;
       else
         CurrentState = NextState;
     end
 always @(CurrentState or SlowRAM)
     begin: COMB
       case (CurrentState)
         0 :
           begin
             Read  = 1;
             Write = 0;
             NextState = 1;
           end
         1 :
           begin
             Read  = 0;
             Write = 1;
             if (SlowRAM)
               NextState = 2;
             else
               NextState = 0;
           end
         2 :
           begin
             Read  = 0;
             Write = 0;
             NextState = 1;
           end
         default :
```

```
            begin
              Read  = 0;
              Write = 0;
              NextState = 0;
            end
        endcase
      end
endmodule
```

Refer to *Figure: Verilog (two_process.v)* for the *nState* diagram.

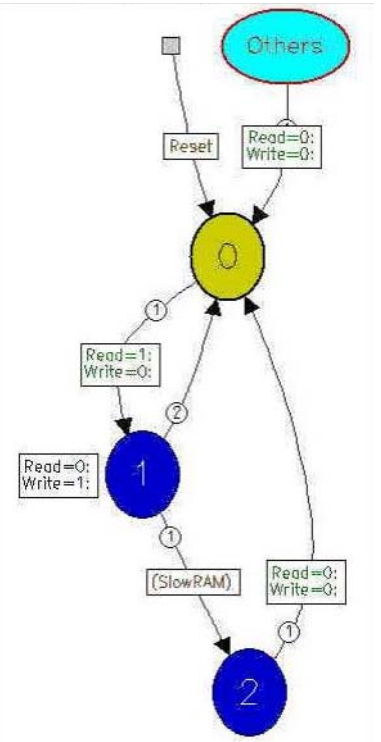


Figure: Verilog (two_process.v)

Example 2 - VHDL (two_process.vhd)

```
library IEEE;
use IEEE.STD_Logic_1164.all;
entity FSM1_GOOD is
   port (Clock, Reset: in  std_logic;
         SlowRAM:      in  std_logic;
         Read, Write:  out std_logic);
end entity FSM1_GOOD;
architecture RTL of FSM1_GOOD is
   type StateType is (ST_Read, ST_Write, ST_Delay);
   signal CurrentState,NextState: StateType;
begin
    SEQ: process
   Begin
      wait until rising_edge(Clock);
         if (Reset = '1') then
            CurrentState <= ST_Read;
         Else
            CurrentState <= NextState;
         end if;
   end process SEQ;
   COMB: process (CurrentState)
   Begin
      case CurrentState is
         when ST_Read =>
            Read  <= '1';
            Write <= '0';
            NextState <= ST_Write;
         when ST_Write =>
            Read  <= '0';
            Write <= '1';
            if (SlowRAM = '1') then
               NextState <= ST_Delay;
            Else
               NextState <= ST_Read;
            end if;
         when ST_Delay =>
            Read  <= '0';
            Write <= '0';
            NextState <= ST_Read;
         when others =>
            Read  <= '0';
            Write <= '0';
            NextState <= ST_Read;
      end case;
    end process COMB;
end architecture RTL;
```

Refer to *Figure: VHDL (two_process.vhd)* for the *nState* diagram.

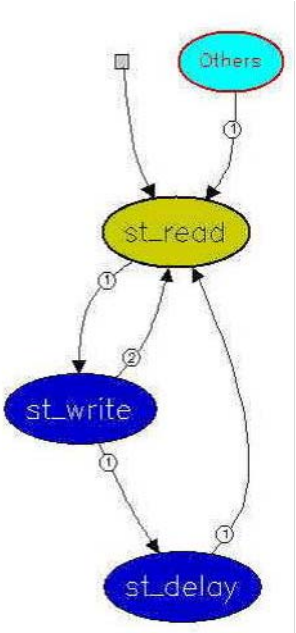


Figure: VHDL (two_process.vhd)

One-Hot Encoding

State-encoding is the way in which binary numbers are assigned to states. The different state encoding formats commonly used are sequential, gray, johnson, one-hot, and define-your-own. The Verdi platform supports sequential, gray, johnson, and one-hot. However, one-hot is written in a different manner than the other formats. An example of Verilog one-hot format is as follows:

```
module prep3 (clk, rst, in, out) ;
input clk, rst ;
input [7:0] in ;
output [7:0] out ;
parameter [2:0]
    START = 0 ,   SA    = 1 ,
    SB    = 2 ,   SC    = 3 ,
    SD    = 4 ,   SE    = 5 ,
    SF    = 6 ,   SG    = 7 ;
reg [7:0] state, next_state ;
reg [7:0] out, next_out ;
always @ (in or state) begin
    // default values
    next_state = 8'b0 ;
    next_out = 8'bx ;
    case (1'b1) // synopsys parallel_case full_case
    state[START]:
        if (in == 8'h3c) begin
            next_state[SA] = 1'b1 ;
            next_out = 8'h82 ;
            end
        else begin
            next_state[START] = 1'b1 ;
            next_out = 8'h00 ;
            end
    state[SA]:
        case (in) // synopsys parallel_case full_case
            8'h2a:
                begin
                next_state[SC] = 1'b1 ;
                next_out = 8'h40 ;
                end
            8'h1f:
                begin
                next_state[SB] = 1'b1 ;
                next_out = 8'h20 ;
                end
            default:
                begin
                next_state[SA] = 1'b1 ;
                next_out = 8'h04 ;
                end
     endcase
    state[SB]:
```

```
            if (in == 8'haa) begin
                next_state[SE] = 1'b1 ;
                next_out = 8'h11 ;
                end
            else begin
                next_state[SF] = 1'b1 ;
                next_out = 8'h30 ;
                end
        state[SC]:
            begin
            next_state[SD] = 1'b1 ;
            next_out = 8'h08 ;
            end
        state[SD]:
            begin
            next_state[SG] = 1'b1 ;
            next_out = 8'h80 ;
            end
        state[SE]:
            begin
            next_state[START] = 1'b1 ;
            next_out = 8'h40 ;
            end
        state[SF]:
            begin
            next_state[SG] = 1'b1 ;
            next_out = 8'h02 ;
            end
        state[SG]:
            begin
            next_state[START] = 1'b1 ;
            next_out = 8'h01 ;
            end
        endcase
        end
    // build the state flip-flopsalways
      always @(posedge clk or negedge rst)
        begin
        if (!rst) begin
            state <= #1 8'b0 ;
            state[START] <= #2 1'b1 ;
            end
        else
            state <= #1 next_state ;
        end
        endmodule
```

Refer to *Figure: One-Hot State-Encoding FSM* for the *nState* diagram.

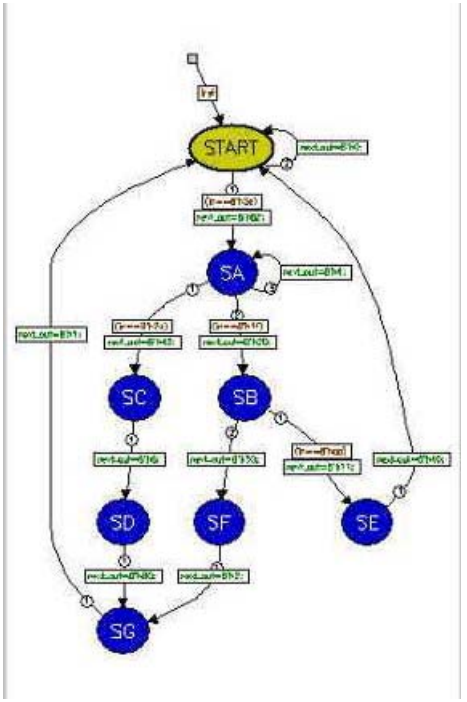


Figure: One-Hot State-Encoding FSM

Shift Arithmetic Operation

FSMs with the next state using simple shift-left arithmetic operations are supported, which is another way to specify one-hot FSM transitions.

An example of Verilog shift arithmetic format is as follows:

```
module sig_control(clock);
parameter
    S0 = 0,
    S1 = 1,
    S2 = 2,
    S3 = 3;
input clock;
reg [3:0] state;
always @(posedge clock)
begin
      case (1'b1)
         state[S0]:
          begin
             state = 1<<S1;
          end
         state[S1]:
          begin
             state = 1<<S2;
          end
         state[S2]:
          begin
             state = 1<<S3;
          end
         state[S3]:
          begin
             state = 1<<S0;
          end
      endcase
end
endmodule
```

Refer to *Figure: FSM Using Shift Arithmetic Operation* for the *nState* diagram.

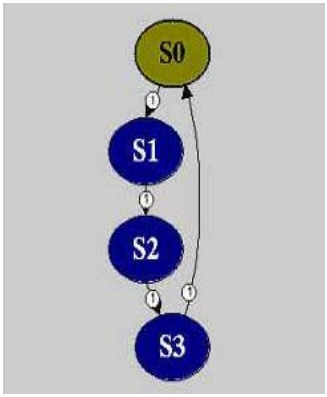


Figure: FSM Using Shift Arithmetic Operation

Case-Statement vs. If-Statement

Designers often use a `case` statement to specify the relationship between the current state and next state. The Verdi platform also allows designers to use `if` statements to do this, even for the conditional operator.

Two Verilog examples are as follows:

Example 1

```
module FSM_1ProcIf;
wire clk, rst;
wire a, b;
reg [1:0] cs, ns;
parameter [1:0] S0=2'b00, S1=2'b01,
                S2=2'b10, S3=2'b11;
always @(posedge clk or posedge rst or a or cs)
begin
    if (rst)
        cs=S3;
else
    if (cs==S0 && a)
            cs=S1;
        else
            if (cs==S1)
                if (b)
                    cs=S2;
                else
                    cs=S3;
            else
                if (cs==S2)
                    cs=S0;
                else
                   if (cs==S3)
                        if (a & b)
                            cs=S2;
                        else
                            cs=cs;
                    else
                        cs=S0;
end
endmodule
```

Refer to *Figure: FSM Using if Statement - 1* for the *nState* diagram.

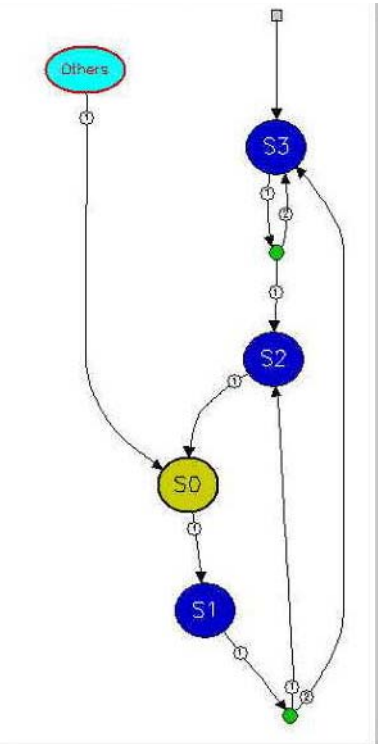


Figure: FSM Using if Statement - 1

Example 2

```
module FSM_2Passign;
wire clk, rst;
wire a, b;
reg [1:0] cs;
wire [1:0] ns;
parameter [1:0] S0=2'b00, S1=2'b01,
        S2=2'b10, S3=2'b11;
always @(posedge clk or rst)
if (rst)
    cs=S3;
else
    cs = ns;
assign ns = ((cs == S0) & a) ? S1 :
    (cs == S1) ? ((b) ? S2 : S3) :
    (cs == S2) ? S0 :
    (cs == S3) ? ((a & b) ? S2 : S3) : S0;
endmodule
```

Refer to *Figure: FSM Using if Statement - 2* for the *nState* diagram.

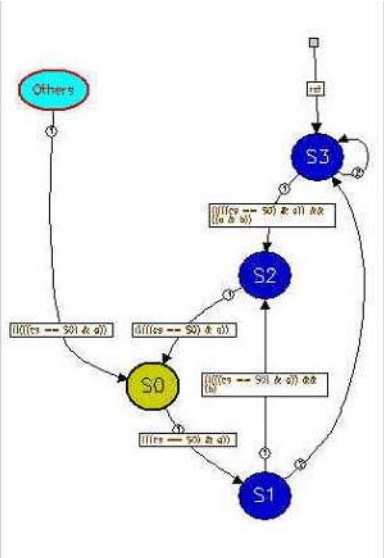


Figure: FSM Using if Statement - 2

Gate-Like FSM

Gate-like FSM is a special type of FSM. Some high performance ASIC vendors prefer this type of FSM because designers perform the optimization themselves, writing it as an RTL statement, as shown in the following example:

```
module state_machine (clk,a,b,c,d,e,f,g,h,i,j,k,l);
input clk,a,b,c,d,e,f,g,h,i,j,k,l;
output STATE0,STATE1,STATE2,STATE3,STATE4, STATE5,STATE6,STATE7;
wire STATE0,STATE1,STATE2,STATE3, STATE4,STATE5,STATE6,STATE7;
wire NSTATE0,NSTATE1,NSTATE2,NSTATE3,
NSTATE4,NSTATE5,NSTATE6,NSTATE7;
assign NSTATE0 = (STATE5 & !a & b & !c) |
                 (STATE0 & !d & !e) |
                 (STATE7 & f  & !c) |
                 (STATE1 & !g & !c) |
                 (STATE0 & c) |
                          (h) |
                          (i);
assign NSTATE1 = (STATE0 & !i & !h & d & !c) |
                          (STATE1 & !i & !h & g) |
                          (STATE1 & !i & !h & c);
assign NSTATE2 = (STATE0 & !i & !h & !d &
                          e & !c) |(STATE2 & !i & !h & c) |
                          (STATE2 & !i & !h& !j);
assign NSTATE3 = (STATE2 & !i & !h & !c & j) |
                          (STATE3 & !i & !h & !k) |
                          (STATE3 & !i & !h & c);
assign NSTATE4 = (STATE3 & !i & !h & k & !c) |
                          (STATE4 & !i & !h & !f) |
                          (STATE4 & !i & !h & c);
assign NSTATE5 = (STATE4 & !i & !h & f & !c) |
                          (STATE5 & !i & !h & !b) |
                          (STATE5 & !i & !h & c);
assign NSTATE6 = (STATE5 & !i & !h & a & b
                          & !c) | (STATE6 & !i & !h & !l) |
                          (STATE6 & !i & !h & c);
assign NSTATE7 = (STATE6 & !i & !h & l & !c) |
                          (STATE7 & !i & !h & !f) |
                          (STATE7 & !i & !h & c);
        sffp #(1)
STATE0l(.ck(clk), .d(NSTATE0), .q(STATE0));
        sffp #(1) STATE1l(.ck(clk), .d(NSTATE1), .q(STATE1));
        sffp #(1) STATE2l(.ck(clk), .d(NSTATE2), .q(STATE2));
        sffp #(1) STATE3l(.ck(clk), .d(NSTATE3), .q(STATE3));
        sffp #(1) STATE4l(.ck(clk), .d(NSTATE4), .q(STATE4));
        sffp #(1) STATE5l(.ck(clk), .d(NSTATE5), .q(STATE5));
        sffp #(1) STATE6l(.ck(clk), .d(NSTATE6), .q(STATE6));
        sffp #(1) STATE7l(.ck(clk), .d(NSTATE7), .q(STATE7));
endmodule
module sffp(ck, q, d);
        parameter width = 1;
```

```
        parameter init = {width {1'b0}};
        output [width-1:0] q;
        input ck;
        input [width-1:0] d;
        reg [width-1:0] q;
     reg [width-1:0] m;
        always @(posedge ck) q <= m;
        always @(negedge ck) m <= d;
endmodule
```

Refer to *Figure: Gate-like FSM* for the *nState* diagram.

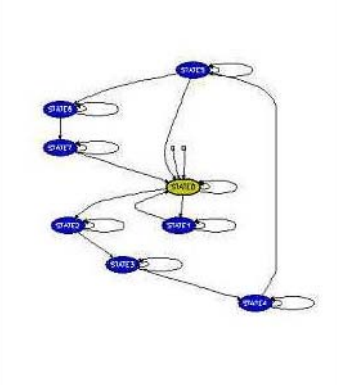


Figure: Gate-like FSM

Next_State = signal

Usually, FSMs are written as `next_state = constant_value`. If `next_state = signal`, the signal's value cannot be determined. This kind of statement is created as a transition to a special bundle node, which means that it may have many undeterminable transitions. The following example shows this type of FSM:

```
module sig_control(clock);
parameter
    S0 = 2'h1,
    S1 = 2'h2,
    S2 = 2'h3,
    S3 = 2'h0;
input clock;
reg [3:0] current_state;
reg reset,enable;
reg [3:0] return;
always @(posedge clock)
begin
   if (reset)
      return <= S0;
   else
   begin
      case (current_state)
     S0: begin
          return <= S1;
          if (enable)
           current_state <= S3;
          else
           current_state <= S0;
         end
     S1: current_state <= S2;
     S2: current_state <= S0;
     S3: current_state <= return;
     default: current_state<= S0;
      endcase
   end
end
endmodule
```

Refer to *Figure: next_state = signal FSM* for the *nState* diagram.

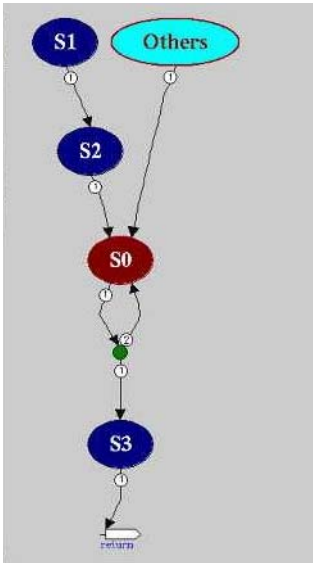


Figure: next_state = signal FSM

Next_State = Current_State + N

For a one process state machine, you may write `current_state = next_state` within the sequential circuit part, and `next_state = current_state + N` (N: positive integer) within the combinational circuit part. The value of `next_state` value is computed automatically.

```
module FSM(Clock,Read,Write);
   input Clock;
   output Read,Write;
   reg Read,Write;
   reg [1:0] State;
   wire [1:0] next_state;
   assign next_state = State + 1;
   always @(posedge Clock)
     begin: SEQ_AND_COMB
     parameter S0=0,S1= 1,S2=2,S3=3;
     case (State)
     S0:
        State = next_state;
     S1:
      begin
        if (Read)
             State = next_state;
        end
     S2:
        State = S0;
     endcase
    end
endmodule
```

Refer to *Figure: next_state = current_state + N FSM* for the *nState* diagram.

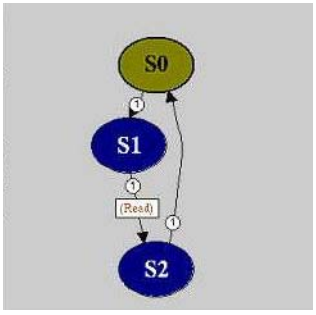


Figure: next_state = current_state + N FSM

VHDL Record Type

The FSM extractor in the Verdi platform supports the VHDL `record` type. Usually, the register is the whole `record`, for example:

```
process
begin
   wait until rising_edge(Clock);
   r <= v;
end process SEQ;
process
begin
   case r.state is
        IDLE=> v.state <= WAIT; -- state transitions
         ....
   endcase
end
```

NOTE: One-hot coding style using `record` is not supported.

```
library IEEE;
use IEEE.STD_Logic_1164.all;
package RecordTypes is
   type StateType is (ST_Read, ST_Write, ST_Delay);
   type R1_Type is record
      State: StateType;
      Output:std_logic;
   end record;
end package RecordTypes;
library IEEE;
use IEEE.STD_Logic_1164.all,
IEEE.Numeric_STD.all;
use work.RecordTypes.all;
entity FSM2_GOOD is
   port (Clock: in  std_logic;
         SlowRAM:      in  std_logic;
         Read, Write:  out std_logic);
end entity FSM2_GOOD;
architecture RTL of FSM2_GOOD is
signal r,v: R1_Type;
begin
   SEQ: process
   begin
      wait until rising_edge(Clock);
            r <= v;
   end process SEQ;
   COMB: process (r)
   begin
      case r.State is
         when ST_Read =>
            Read  <= '1';
            Write <= '0';
```

```
            v.Output <= '1';
               v.State <= ST_Write;
            when ST_Write =>
               Read  <= '0';
               Write <= '1';
            v.Output <= '1';
               if (SlowRAM = '1') then
                  v.State <= ST_Delay;
               else
                  v.State <= ST_Read;
               end if;
            when ST_Delay =>
               Read  <= '0';
               Write <= '0';
            v.Output <= '1';
               v.State <= ST_Read;
            when others =>
               Read  <= '0';
               Write <= '0';
            v.Output <= '0';
               v.State <= ST_Read;
         end case;
      end process COMB;
   end architecture RTL;
```

Refer to *Figure: VHDL Record Type* for the *nState* diagram.

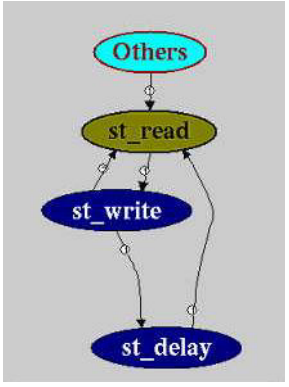


Figure: VHDL Record Type

Appendix C: Enhanced RTL Extraction

Overview

nSchema can display instance array, for loop statements and create detailed extracted schematic view. Typically, these complex functions are displayed with a function symbol. The RTL can be extracted to show more detailed view.

To turn on detail RTL extraction feature, choose the **Tools -> Preferences** option to open the *Preferences* form and then turn *on* the **Enable Detail RTL** option on the **RTL** page under the **Schematics** folder. With this option turned *on*, the Verdi platform extracts more RTL as follows.

- Expand *instance array* to individual instance bit.
- Expand contents of *for loop* statement.
- Handle *aggregate* with positional notation, named notation and others.
- For partial bit assignment, split constant, concatenation and aggregate correctly according to their specific bit range.

The following figure shows a list of recognized RTL blocks:

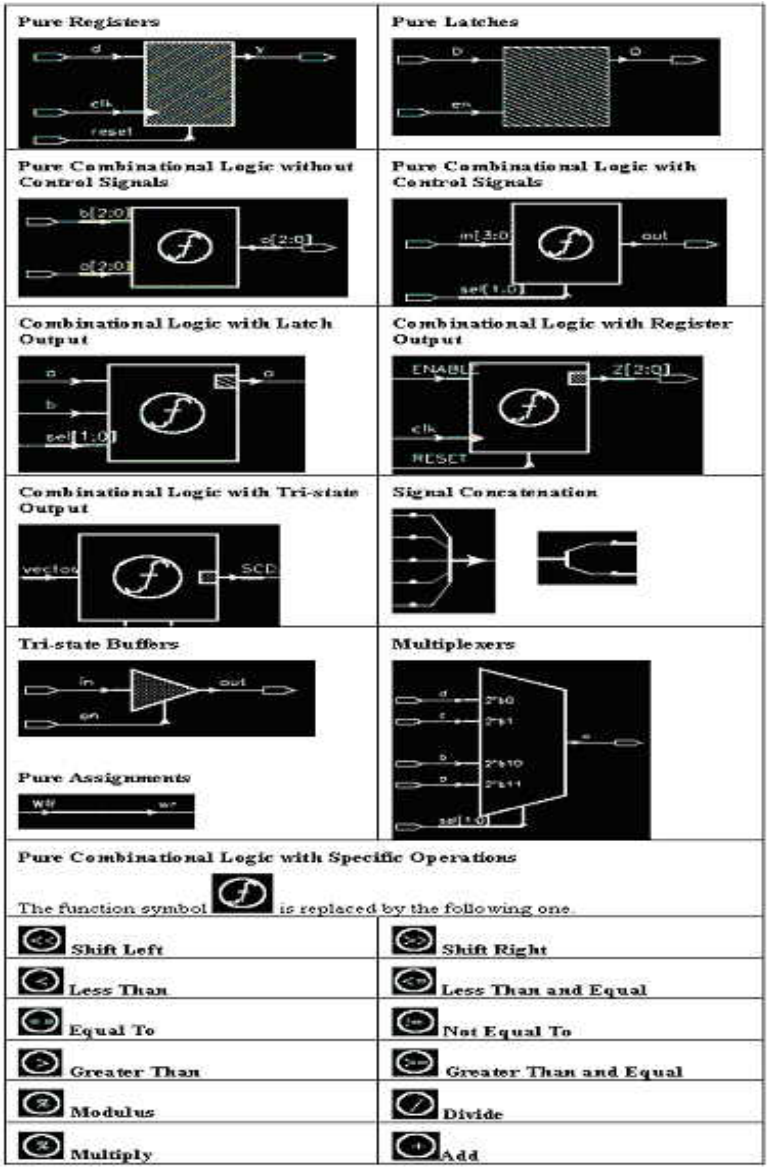


Figure: List of Recognized RTL Blocks

The following sections are examples of the enhanced detail RTL extractions:

- *Instance Array*
- *For Loop*

- *Aggregate*
- *Partial Bits Assignment*
- *Displaying Pure Memory Blocks*

Instance Array

The *instance array* can be expanded to individual bits. In the following example, *U1[3:0]* can be expended to *U1[0]*, *U1[1]*, *U1[2]* and *U1[3]* with correct port connections.

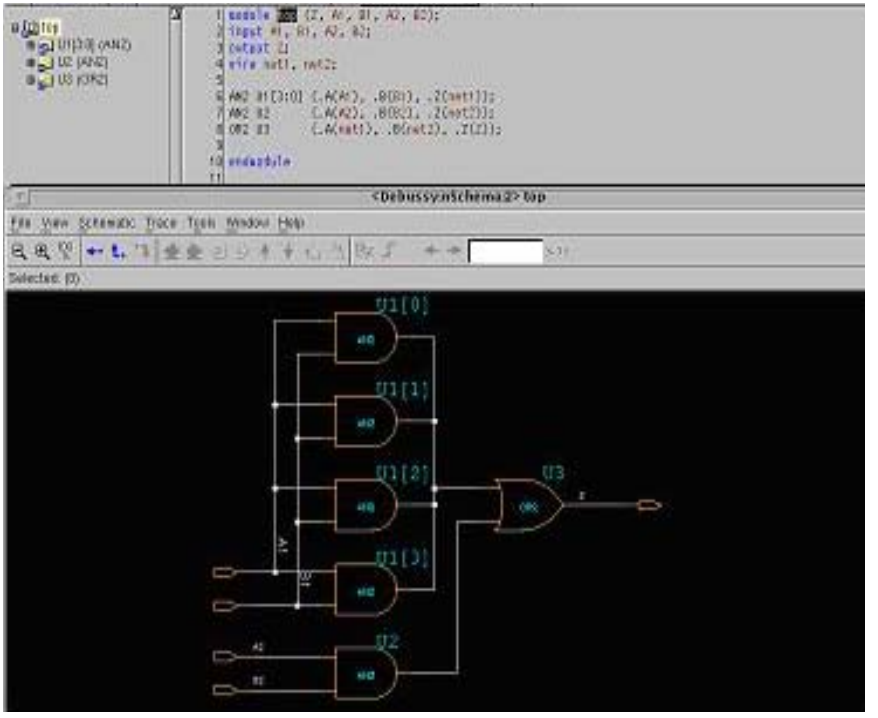


Figure: Instance Array

For Loop

The for loop can be expanded. In the following for loop example, net is expended to *net(1)*, *net(2)* and *net(3)*.

NOTE: Nested for loops are not supported.

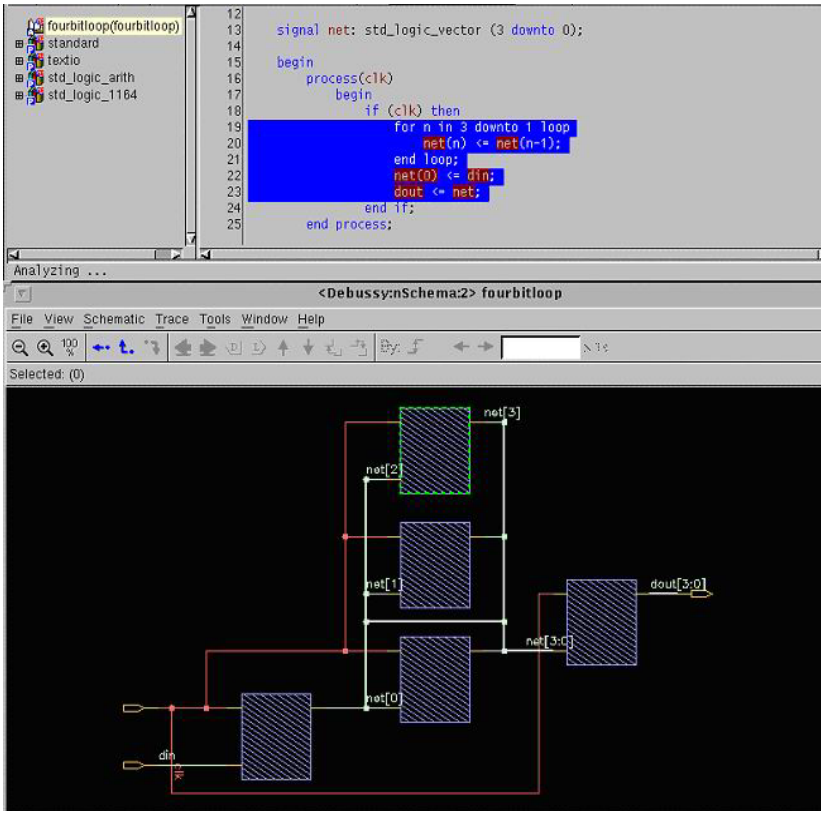


Figure: For Loop

Aggregate

1. Positional notation: ('1', 'a', 'b')

   In the following example, the extracted RTL shows the following:

```
Y(3) <= a;
Y(2) <= '1';
If (c = '1') Y(1) <= '1' else Y(1) <= b;
If (c = '1') Y(0) <= '1' else Y(0) <= '1';
```

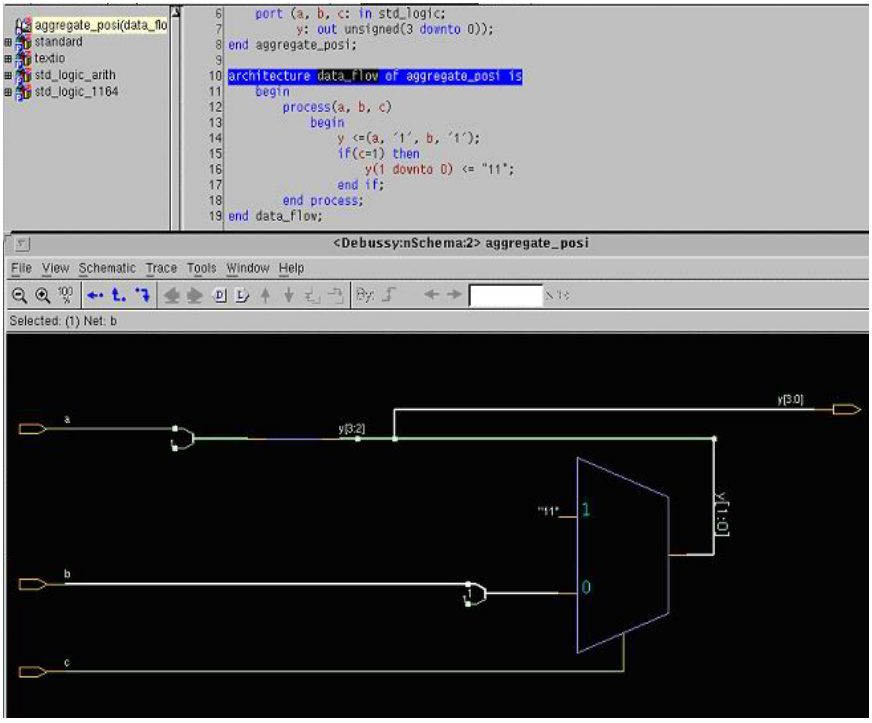


Figure: Aggregate: Positional Notation

2. Named notation: (2=>'a', 3=>'b', 0=>'0', 1 => '0')

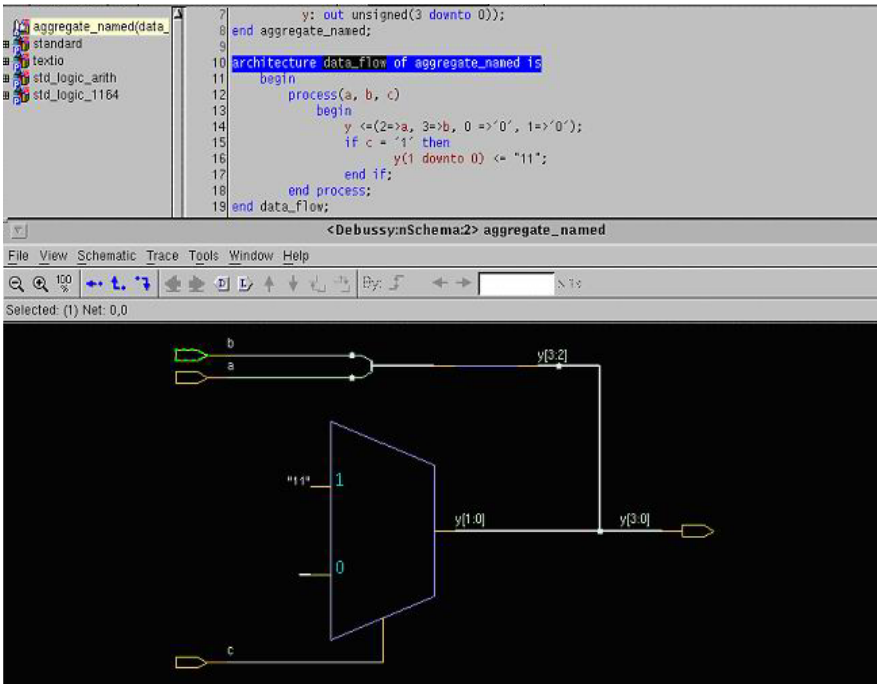


Figure: Aggregate: Named Notation

3. Others: (2=>'a', 3=>'b', others=>'1')

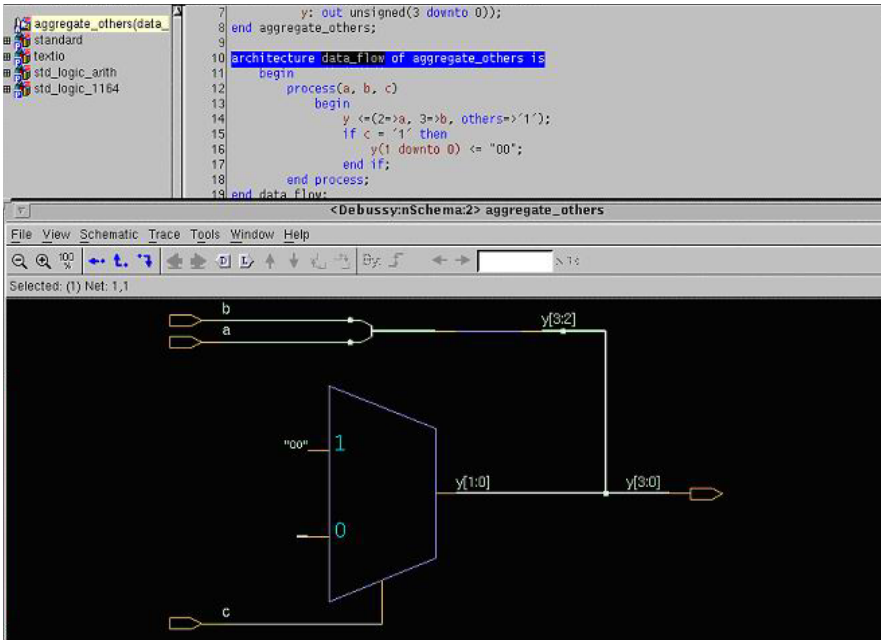


Figure: Aggregate: Others

Partial Bits Assignment

1. Constant

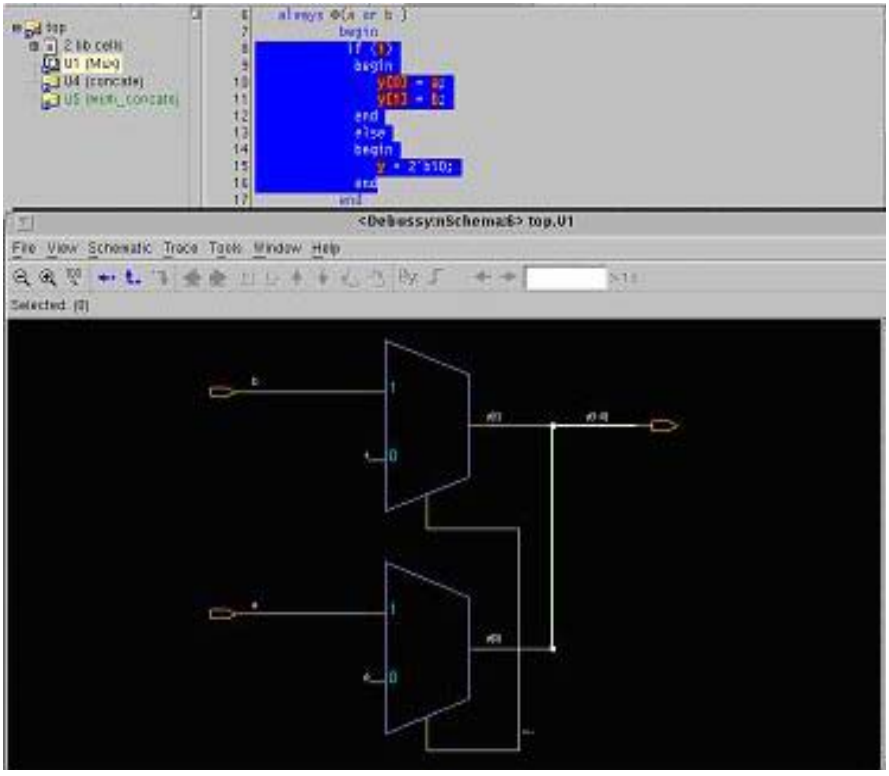


Figure: Partial Bits Assignment: Constant

2. Concatenation

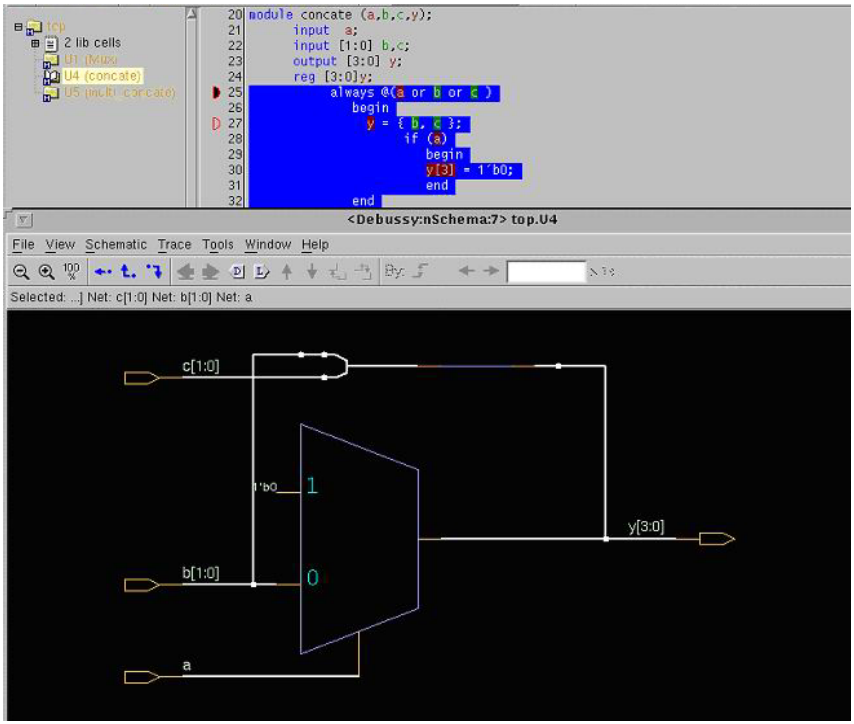


Figure: Partial Bits Assignment - Concatenation

3. MultiConcatenation

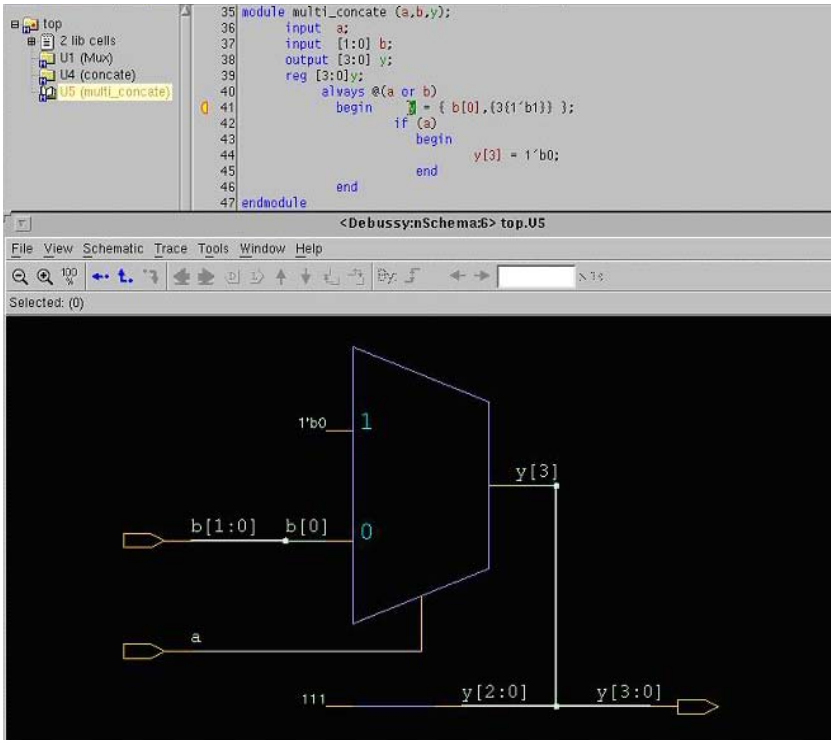


Figure: Partial Bits Assignment -- Multi-Concatenation

Displaying Pure Memory Blocks

To show the pure memory block in *nSchema*, the memory block is separated from other circuits.

For example, in the case of the FIFO, `FIFO_r` is used to store information. `WrPntr_r/RdPntr_r` is a pointer that increases or decreases as the FIFO is pushed or popped:

```
Pointers_Proc : process is
  begin  -- process Pointers_Proc
    wait until Clk = '1';
    case PushnPopn is
      when "00" =>                      -- Push and pop at same
clock
        -- no change to pointers or status
        FIFO_r(WrPntr_r) <= data_in;    -- store data
      when "01" =>                      -- Push, no pop
          FIFO_r(WrPntr_r) <= data_in;  -- store data
          WrPntr_r        <= (WrPntr_r + 1) mod Depth_g;
          -- right argument must evaluate to a constant integer
power of 2
      when "10" =>                      -- no push, pop
          RdPntr_r <= (RdPntr_r + 1) mod Depth_g;
      when "11" =>                      -- no push, no pop
        null;
      when others => null;
    end case;
    if Resetn = '0' then
      WrPntr_r <= 0;
      RdPntr_r <= 0;
      FIFO_r   <= (others => (others => '0'));
    end if;
  end process Pointers_Proc;
  DataOut_r <= FIFO_r(RdPntr_r);
```

The detail RTL view of this example is shown in the following figure:

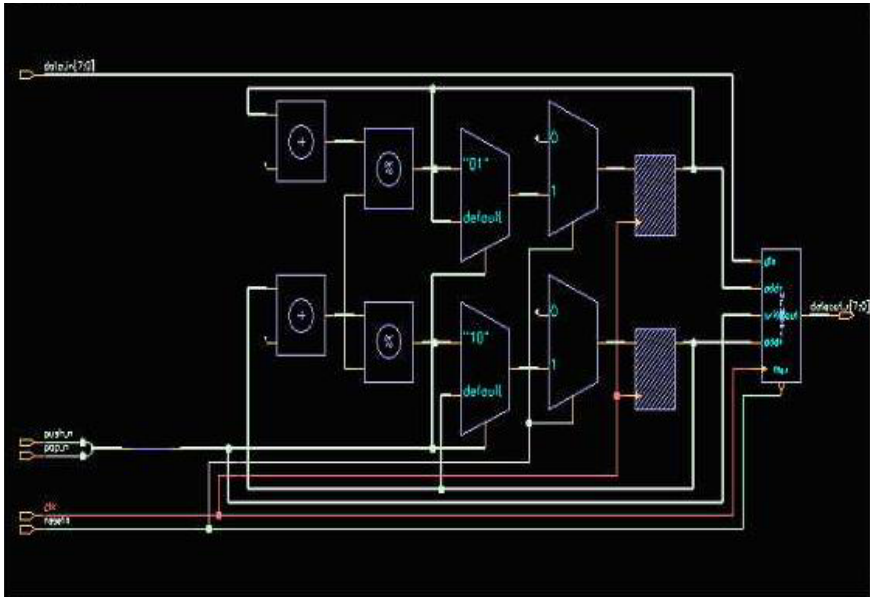


Figure: Memory Block

Appendix D: Additional Transaction Example

This appendix introduces how to extract transactions using SystemVerilog Assertions (SVA).

Extracting Transactions Using SVA

SystemVerilog Assertions (SVA) are added to your design and then extracted to display as transactions.

Before you can extract transactions from SVA, you must do the following:

1. Add the SVA code to your design either inlined or as a separate file.
2. Generate an FSDB file containing design data with your preferred simulator.
3. Load the design and FSDB files into the Verdi GUI.

After the design and FSDB files are loaded into the Verdi GUI, you can extract the transactions by invoking the **Tools -> Transaction -> Evaluator** command in the *nTrace* main window. This opens the *Transaction Evaluator* form where all SVA assert signals are listed.

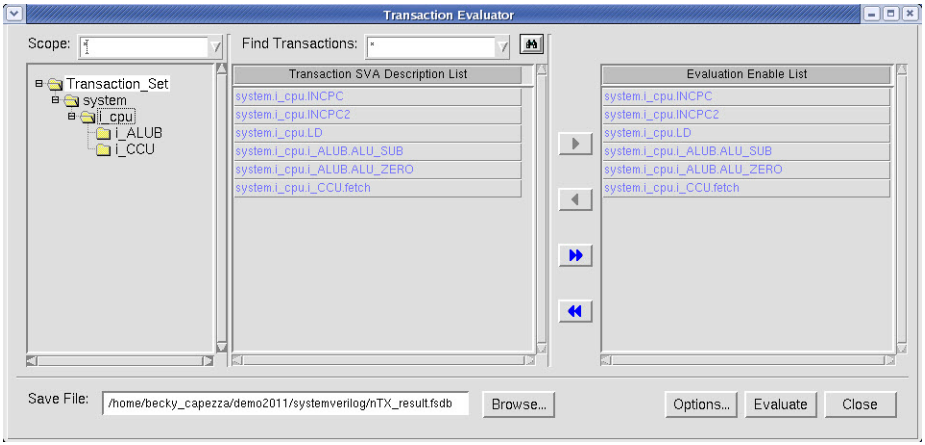


Figure: Transaction Evaluator Form

In the *Transaction Evaluator* form, the design hierarchy is displayed in the left pane. After you have traversed to the scope of interest, the transactions are listed in the middle pane. You can select the assertions to be extracted and click either the **Add Selected Transaction** button or **Add All Transactions** button to move the selection to the *Evaluation Enable List* pane. You can also drag any of the assertions to the source code frame to see the related code.

After you click **Evaluate**, the transactions are extracted from the assertion code and are saved to the specified file. This FSDB file is automatically loaded into the Verdi GUI and you can start using all transaction viewing and analysis commands for debug in addition to the standard Verdi capability.

NOTE: You need to add the transaction waveforms using the **Get Signals** command of *nWave*. Transaction signals have an `_nTX` suffix appended to the assertion name.

SVA Code

The following sections contain a summary of recommended and unsupported coding styles:

- *Recommended Coding Style*
- *Unsupported Coding Style*
- *Code Example*

Recommended Coding Style

The following coding styles are recommended for optimum transaction extraction results:

- Only “assert” directive is supported.
- Most SVA constructs are supported. For details, see *Unsupported Coding Style*.
- Using constructs, the sequence layer is recommended for modeling transactions. Deep nesting range repetition and unbound range delay, for example `##[0:$]`, are not recommended as it impacts performance.
- SVA local variables, including those declared in the sub-sequence of a specific assertion, are recorded as attributes of transactions. Therefore, it is not recommended to declare local variables with the same name across different sequences or properties.

Example 1:

```
sequence single_read;
  logic [31:0] addr;
  logic [31-1:0] data;
  int ws;

  @(posedge hclk)
     (`true,ws = 0) ## 0
     (hready) ##1
     (!hready && hsel) [*0:$] ##1
     ((hready && hsel && `SR_CTRL), addr = haddr) ##1
     ((!hready && hsel), ws = ws + 1) [*0:$] ##1
     (hready, data = hrdata);
endsequence

SINGLE_READ: assert property(single_read);
```

The `addr`, `data`, and `ws` local variables of the `single_read` sequence are recognized as the attributes of the assertion statement, `SINGLE_READ`.

Example 2:

```
sequence s1;
     int localvar;
     ...
endsequence
sequence s2;
     int localvar;
     ...
endsequence
a_trans1: assert property(@(posedge clk) s1 and s2);
```

You can modify the sequence as follows:

```
sequence s1;
     int localvar1;
     ...
endsequence
sequence s2;
     int localvar2;
     ...
endsequence
a_trans1: assert property(@(posedge clk) s1 and s2);
```

- You can specify the transaction label name of a specific sequence by declaring a string type local variable named `label_nTX`, and assigning a label name to it. For example, if you specify the following for a sequence/property, the transaction label name is `my_single_read`:

```
sequence single_read;
  string label_nTX;
  (..., label_nTX = “my_single_read”,...) ...;
endsequence
```

If you specify the following for an assertion statement, the `label_nTX` variable (if exists) of the sub-sequence/property is used as its transaction label:

```
sequence s1;
  string label_nTX;
  (..., label_nTX = “my_s1”,...) ...;
endsequence

sequence s2;
  string label_nTX;
  (..., label_nTX = “my_s2”,...) ...;
endsequence

a_s1 : assert property((@posedge clk) s1 ##1 s2);
```

In this case, the label is either `my_s1` or `my_s2`.

Unsupported Coding Style

The following coding styles are not supported for transaction extraction:

- Multiple clocking is not supported.
- Immediate assertion is not supported.
- `cover` and `assume` directives of SVA are not supported. Only the `assert` directive is supported.
- Three types of assertion successes are not recognized as a transaction:
  - The vacuous success of the implication is not recognized as a transaction.
  - The abort success of `disable iff` is not recognized as a transaction.
  - Empty matches, for example `seq1[*0];`, are not recognized as a transaction.

Code Example

Consider the following SVA code example:

```
bind test assert_checker bind_transaction_evaluator(
.EN        (test.uFL_AMBA_SRAM.ram_2kx32.mem.EN),
.WE        (test.uFL_AMBA_SRAM.ram_2kx32.mem.WE),
.ADDR      (test.uFL_AMBA_SRAM.ram_2kx32.mem.ADDR),
.DI        (test.uFL_AMBA_SRAM.ram_2kx32.mem.DI),
.DO        (test.uFL_AMBA_SRAM.ram_2kx32.mem.DO),
.CLK       (test.uFL_AMBA_SRAM.ram_2kx32.mem.CLK),
.RST       (test.uFL_AMBA_SRAM.ram_2kx32.mem.RST),
.RDInvalid (test.uFL_AMBA_SRAM.uSMI.iXOEN_d)
);

module assert_checker (
 input  EN,
 input  WE,
 input  [10:0] ADDR,
 input  [31:0] DI,
 output [31:0] DO,
 input  CLK,
 input  RST,
 input  RDInvalid
);

sequence core_memory_write;
 logic [10:0] Addr;
 logic [31:0] Data;

 (1) ## 0
 (EN == 1'b1 && WE == 1'b1, Addr = ADDR, Data = DI) ##1
 (!(EN == 1'b1 && WE == 1'b1));
endsequence

sequence core_memory_read;
 logic [10:0] Addr;
 logic [31:0] Data;

 (1) ## 0
 (WE==1'b0 && RST==1'b0 && RDInvalid==1'b0, Addr = ADDR) ##1
 (RDInvalid == 1'b0) ##1
 (1, Data = DO);
endsequence

CORE_MEM_WRITE : assert property(@(posedge CLK)
core_memory_write);
CORE_MEM_READ  : assert property(@(posedge CLK)
core_memory_read);
```

```
endmodule
```

The example is extracted and displayed as transaction waveforms similar to the following:

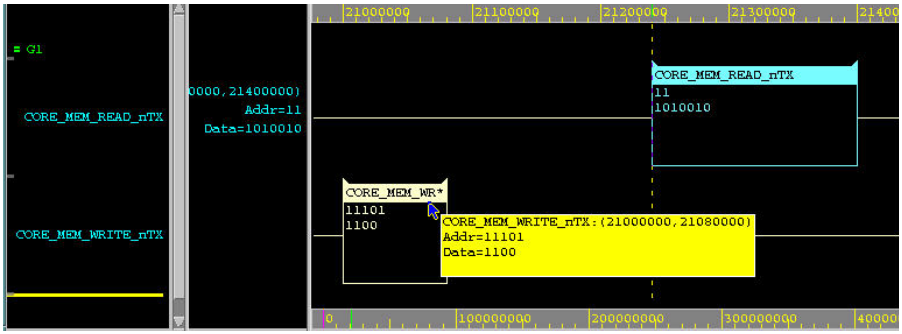


Figure: Extracted Transaction Waveform

Integration Features

This chapter provides the information about the following native integrations that are available as part of the Verdi platform:

- Native Integration of Verdi and VCS
  - Unified Compile Front End
  - Unified Debug Solution

    Interactive and Post Simulation Debug Flow

    UCLI Dump Command for FSDB Dumping
  - Optimized Performance of Gate-Level Designs Using Native FSDB Gate
- Unified Transaction Debug- Verdi and Protocol Analyzer Integration
- Unified UVM Library

The following section provides the information about the enhancements of Switching Analysis that are available as part of the Verdi platform J-2014.12-SP1 release:

- Scope-Based Peak Analysis

All these features are Limited Customer Availability (LCA). Limited Customer Availability features are features available with the select functionality. These features will be ready for a general release, based on customer feedback and meeting the required feature completion criteria. LCA features do not need any additional license keys.

Native Integration of Verdi and VCS

This section consists of the following native integrations that are available as part of Verdi:

Unified Compile Front End

This section consists of the following subsections:

Introduction

Unified Compile Front End uses VCS compiler scripts to compile the Knowledge Database (KDB) for Verdi. Consequently, only one common compiler script needs to be maintained for both VCS and Verdi, which ensures consistency between their databases.

The benefits offered by Unified Compile Front End are as follows:

- Single VCS and Verdi compilation
- Consistent HDL language support
- Consistency in utilizing or handling VCS and Verdi options

Prerequisite

Specify the `VCS_HOME` environment variable to the VCS installation path.

For example:

```
%> setenv VCS_HOME <VCS Install Path>
```

Set the UNIX PATH variable to `$VCS_HOME/bin` as follows:

```
%> set path =($VCS_HOME/bin $path)
```

OR

```
%> setenv PATH $VCS_HOME/bin:$PATH
```

Generating Verdi KDB With Unified Compile Front End

Unified Compile Front End is supported in both the VCS two-step and three-step flows. In the VCS two-step flow, add the `-kdb` option to the command line to generate the KDB. In case of the VCS three-step flow, add it in all the `vlogan`/`vhdlan`/`vcs` command lines.

When you specify the `-kdb` option, Unified Compile Front End creates the Verdi KDB and dumps the design into the libraries specified in the `synopsys_sim.setup` file.

For example:

- `-kdb`

  Generates both the VCS database for simulation and the Verdi KDB for debugging. The Verdi KDB is required for both post-process and interactive simulation debug. For example:

  ```
  // Compile design using VCS and generate both the VCS
  // database and the Verdi KDB

  // -kdb in the VCS two-step flow
  % vcs -kdb <compile_options> <source files>

  // -kdb in the VCS three-step flow
  %> vlogan -kdb <vlogan options> <source files>
  %> vhdlan -kdb <vhdlan options> <source files>
  %> vcs -kdb <top_name>
  ```

- `-kdb=only`

  To generate only the Verdi KDB and skip the simulation database generation, specify the following argument with the `-kdb` option:

  ```
  -kdb=only
  ```

  Generates only the Verdi KDB that is needed for both post-process and interactive simulation debug with Verdi.

  This option is supported only in the VCS two-step flow. It is not supported in the VCS three-step flow.

  In the VCS two-step flow, this option does not generate the VCS compile data/executable, and does not disturb the existing VCS compile data/executables.

  For example:

  ```
  % vcs -kdb=only <compile_options> <source files>
  ```

Reading Compiled Design With Verdi

To read a compiled design, add the `-simflow` option to the Verdi command line. The `-simflow` option imports the KDB compiled by Unified Compile and enables the Verdi platform and its utilities to use the library mapping from the `synopsys_sim.setup` file. For example:

```
%> verdi -simflow -lib work
```

You can also use the `-simBin` option to import design directly from the KDB library paths. For more details about the `-simBin` option usage, see the *Interactive and Post Simulation Debug Flow* section. For example:

```
%> verdi -simflow -simBin <simv_path>
```

You can perform the same operations through the Verdi GUI as follows:

1. Click **File -> Import Design**.
2. In the *Import Design* form, select the *From Library* tab.

In the **From** field, select the **VC/VCS Native Compile** option, as shown in the figure below.

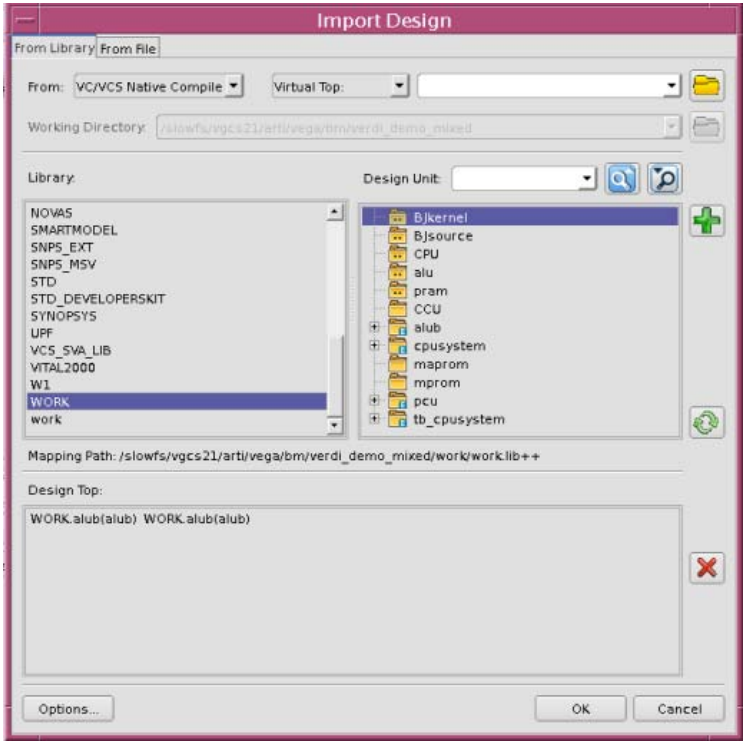


Figure: The Import Design Form

You can also add the `-simdir <path>` option to the Verdi command line to ensure that VCS and Verdi use the same data from the `synopsys_sim.setup` file. For example:

```
%> verdi -simflow –simdir [<path>] -lib work -top [<your top
module>]
```

The `<path>` argument points to the directory from where the `simv` (VCS simulation executable) is executed. Use this option if you want to invoke Verdi from a working directory that is different from the VCS working directory.

NOTE: When compile in the 64-bit platform machine, add `-full64` to `vlogan/vhdlan/vcs`. This is because Verdi selects the executable automatically according to the current platform, but VCS uses 32-bit executable, by default. It causes problem for Verdi 64-bit executable to read the KDB generated from 32-bit VCS. For example:

```
% vcs -full64 -kdb <compile_options> <source files>
```

Notes

- The `vericom` utility exists in Verdi. For VCS users, Unified Compile flow is recommended to generate KDB for data consistency and better performance. For third-party simulator users, the compile flow does not change and continues to use `vericom`. When loading the compiled design library (KDB) from the GUI (loading from the command line stays the same), ensure that the **Verdi Compile** option is selected in the **From** field in the *From Library* tab of the *Import Design* form.
- As VCS and `vericom` are different Verilog compilers, there are some behavioral differences between them. In such cases, Unified Compile follows the behavior of `vlogan` (VCS) for consistency reasons. The supported language subset also follows the supported subset of VCS.
- All the compilation information including the compile log of the Verdi KDB is logged to the regular VCS compiler log file.
- The library mapping information is obtained from the `synopsys_sim.setup` file in the VCS three-step flow. The library mapping information in the `novas.rc` resource file is ignored in the VCS three-step unified compile flow.
- Unified Compile does not apply to the *import-from-file* flow of Verdi. The *import-from-file* flow continues to use the `vericom` parser to read in the Verilog source code directly. It uses the library mapping information from the `novas.rc` resource file similar to the Verdi behavior.
- In the VCS two-step flow, the `vcs` generated KDB is saved as the `work.lib++` directory in the same working directory as `simv.daidir`.
- In the VCS three-step flow, the `vlogan` generated KDB is saved as a `work.lib++` directory in the same working directory as `AN.DB`. You can verify the KDB in the directory where it is generated. Note that you can specify the working directory with the `-work` option of `vlogan`. Use the `verdi -simflow -lib` option to specify the working directory to load KDB.

Limitations

The following are the limitations with Unified Compile:

- Verilog-AMS (AMS) and Property Specification Language (PSL) are not supported. Verdi can parse constructs successfully without an error message. However, Verdi has a limited support for debug functionality for AMS and PSL.
- Parallel compilation is not supported.

- Fault tolerance compilation is not supported.

Interactive and Post Simulation Debug Flow

This section consists of the following subsections:

Introduction

To debug a simulation failure in a design and to bring up the desired debugger GUI, you may need to remember and explore different options, which result in spending a lot of time on setting up debugging tools rather than real debugging. Additionally, you need to manually configure Verdi to perform interactive simulation debugging in Verdi with VCS. You also need to manually load the design to Verdi to perform post-simulation debugging.

After the Verdi Knowledge Database (KDB) is generated using Unified Compile, you can invoke Verdi with the KDB in a single step for the following debug modes respectively:

- Interactive Simulation Debug Mode

  You can automatically invoke Verdi with the KDB through the simulator command-line option to perform interactive simulation debugging in Verdi without other configurations.

- Post-Simulation Debug Mode

  The KDB and the `synopsys_sim.setup` file information is automatically loaded into Verdi through a command-line option to perform post-simulation debugging. There is no need to manually specify the compiled design. VCS and Verdi get the same information from the `synopsys_sim.setup` file.

Prerequisites

The following is the prerequisite to perform *interactive simulation debugging* using the Unified Debug solution:

- Specify the VCS_HOME environment variable to the VCS installation path. For example:

  ```
  %> setenv VCS_HOME <VCS Install Path>
  ```

- Generate the Verdi KDB using Unified Compile. For more information, see Unified Compile Front End.

The following are the prerequisites to perform *post-simulation debugging* using the Unified Debug solution:

- Specify the VCS_HOME environment variable to the VCS installation path. For example:

  ```
  %> setenv VCS_HOME <VCS Install Path>
  ```

- Generate the Verdi KDB using Unified Compile. For more information, see Unified Compile Front End.
- Specify the -debug_access+<option> compile-time option on the VCS command line. This option automatically picks up the Novas tab file and the Novas PLI file and there is no need to pass these files explicitly during compilation. For example,

  ```
  // Add -debug_access[+<option>] in the VCS two-step flow
  % vcs -kdb -debug_access+all <source files>
  ```

  For more information on this option, see the *VCS documentation*.

NOTE: To enable the complete set of debug capabilities, specify the -debug_access+all option.

- Enable FSDB file dumping using the dumping tasks present in the source file or at runtime using fsdbDumpvars from the UCLI command line.

Interactive Simulation Debug Flow

When executing the simv simulator executable, perform one of the following steps to invoke Verdi within the interactive simulation debug mode:

- Add the -gui/-verdi/-gui=verdi options to specify Verdi as the debug tool. For example:

  ```
  // invoke Verdi
  %> simv <simv_options> -verdi [-verdi_opts “<verdi_options>”]
  %> simv <simv_options> –gui=verdi [-verdi_opts
  “<verdi_options>”]
  ```

- Set the SNPS_SIM_DEFAULT_GUI environment variable to verdi to specify Verdi as the debug tool. For example:

  ```
  // invoke Verdi
  %> setenv SNPS_SIM_DEFAULT_GUI verdi
  ```

```
%> simv <simv_options> -gui [-verdi_opts "<verdi_options>"]
```

Key Points to Note

- Use the `-verdi_opts` options to specify other Verdi-specific options.
- The UVM Interactive Debug in Verdi is enabled by default while using the Unified Debug solution.
- If the design includes SystemC and the `default.ridb` is not available in the `simv.daidir` directory, Verdi generates it automatically.

Post-Simulation Debug Flow

To automatically load the KDB compiled by Unified Compile, use the following Verdi command-line options:

- `-simflow`

  Enables Verdi and its utilities to use the library mapping from the `synopsys_sim.setup` file and import a design from the KDB library paths.

- `-simBin <simv_path>`

  Specifies the path of the `simv` executable. This ensures that VCS and Verdi have the same data from the `synopsys_sim.setup` file.

  For example:

  ```
  %> verdi -simflow -simBin [<simv_path>]

  //import the FSDB file into Verdi
  %> verdi -simflow -simBin [<simv_path>] -ssf novas.fsdb
  ```

  After specifying the path of `simv`, you can also directly start the Verdi Interactive Simulation Debug mode by using the **Tools -> Run Simulation** menu command in the Verdi *nTrace*.

  If the design contains SystemC and the `default.ridb` file exists in the `simv.daidir` directory, the `default.ridb` file is also loaded into the KDB for SystemC debugging.

NOTE:

* When the `-simflow` and `-simBin` options are used together, all other options related to importing KDB are ignored.
* If you are trying to perform the post-simulation debug from a directory different than the compilation directory, you must specify the absolute physical path mapping in the `synopsys_sim.setup` file.

- `-simdir <path>`

  Specifies the path of the library directory when you want to invoke Verdi from a working directory that is different from the VCS working directory. For more information, see Unified Compile Front End.

Limitations

The following is the limitation when performing power debug with UPF:

- The UPF file needs to be manually imported into Verdi both for Interactive and Post-simulation debug flows:
  - In Interactive simulation debug flow, add the `-upf <UPF file>` option to import your UPF file.

  For example:

```
%> vlogan -kdb <compile_options> <source files>
%> vcs -kdb -upf <UPF file>
%> simv -gui -upf <UPF file>
```

  - In the Post-simulation debug flow, add the `-upf <UPF file>` option to import your UPF file.

  For example:

```
%> vlogan -kdb <compile_options> <source files>
%> vcs -kdb -upf <UPF file>
%> simv
%> verdi –ssf novas.fsdb –simflow –simBin <simv_path/simv> -upf
<UPF file>
```

UCLI Dump Command for FSDB Dumping

This section consists of the following subsections:

Introduction

The UCLI `dump` command is enhanced to dump the Fast Signal Database (FSDB) file in addition to the VPD and EVCD file dumping.

Now, you can use the UCLI `dump` command to dump the FSDB file by default, instead of calling the FSDB system tasks or using FSDB commands on the UCLI command prompt.

You can also perform the following operations using the `dump` command:

- Simultaneously open single VPD, EVCD, and FSDB dump files and manage them individually.
- Simultaneously open multiple FSDB dump files and manage them individually.

Use Model

The following steps describe the use model:

1. The default dump type of VCS is VPD. You can use the following environment variable to set the default GUI as Verdi and the default dump type as FSDB

   ```
   % setenv SNPS_SIM_DEFAULT_GUI verdi
   ```
2. Set `VERDI_HOME` as provided in the following command line:

   ```
   % setenv VERDI_HOME <novas_path>
   ```
3. Compile your designs with the `-debug_access+cbk` option, as provided in the following command line:

   ```
   % vcs -debug_access+cbk <file_name>
   ```
   OR

   Compile your designs with a debug option (that is, `-debug`, `-debug_pp`, or `-debug_all`), as provided in the following command line:

```
% vcs debug_option -p novas.tab pli.a <file_name>
```

NOTE: If you use `-debug`, `-debug_pp`, and `-debug_all` options, you must specify `novas.tab` and `pli.a` files in the `vcs` command line. The `-debug_access+cbk` option automatically sets the `novas.tab` and `pli.a` files.

Key Points to Note

- If a single dump file is open, it is not required to specify the `-fid` argument with the dump commands that follow the `dump -file` command. If multiple dump files are open, you must specify the `-fid` argument with the dump commands that follow the second dump `-file` command.
- During simulation, if the number of open dump files return to one, you can exclude the `-fid` argument. VCS issues an error message, if a dump command is specified without the `-fid` argument when multiple dump files are open

Enhanced and New UCLI Dump Options

Several UCLI `dump` options are enhanced and new added for dumping FSDB file. For details, see the *VCS documentation*.

Limitation

For details, see the *VCS documentation*.

Optimized Performance of Gate-Level Designs Using Native FSDB Gate

This section consists of the following subsections:

Introduction

Verdi provides the Fast Signal Database (FSDB)-Gate feature for gate-level designs without Standard Delay Format (SDF) information. You can invoke the FSDB-Gate feature using VCS, which supports optimized FSDB gate-level dumping.

To enable this feature, use the VCS `+fsdb+gate` runtime option. It directs VCS to analyze essential signals and the netlist information including the signature, function table, and partition mapping, and uses the FSDB Dumper to record this information in an FSDB file. Applications, such as Waveform Viewer and FSDB Reader, retrieve the mapping data stored in the FSDB file. The retrieved data is further used by the VCS computation engine to generate the complete signal data during debugging.

Additionally, if you enable the VCS force capability along with the `-debug`, `-debug_all`, or `-debug_access+f+fwn` compilation options, the VCS dynamic de-aliasing capability is also enabled while dumping forced signals into the FSDB file. The FSDB Reader then interprets the event and generates the related waveform in Verdi.

The FSDB Gate and dynamic de-aliasing acceleration features reduce the FSDB file dumping size and optimize the VCS simulation time for specific coding styles and forced signal flows.

Prerequisites

These features are available starting with the following versions:

- VCS simulator 2014.12
- Verdi 2014.12
- FSDB Reader 5.2 (for user of FSDB reader API)
- If the API libraries of the FSDB Reader are used to read the FSDB file with new format, a Verdi license is required.

Using the FSDB Gate Feature

Use the +fsdb+gate runtime option in the VCS simulation command line to enable these features.

For example,

```
%> ./simv +fsdb+gate
```

Alternatively, set the following environment variable before starting the simulation:

```
%> setenv FSDB_GATE 1
```

Key Points to Note

- After simulation, a new format of the FSDB file is generated.
- FSDB cannot be read by previous Verdi version, for example 2014.03.
- Expect to see a higher simulation speed in the SystemVerilog Gate-Level design without SDF.
- FSDB reading performance (CPU or memory) when using Verdi debug might be impacted.

Limitations

The following are the limitations for the FSDB-Gate feature:

- The +fsdb+gate option is disabled with a warning message, if you add any of the following FSDB Dumper options in the simulation command line or if you specify them using the setenv command:

  - +fsdb+glitch=<num> (its corresponding environment variable is NOVAS_FSDB_ENV_MAX_GLITCH_NUM or FSDB_GLITCH): If the <num> argument is not equal to 1, the +fsdb+gate option is disabled.

  - +fsdb+dumpon_glitch+time and +fsdb+dumpoff_glitch+time

  - +fsdb+region (its corresponding environment variable is FSDB_REGION)

  - +fsdb+sequential (its corresponding environment variable is NOVAS_FSDB_ENV_DUMP_SEQ_NUM)

  - +fsdb+strength=on (its corresponding environment variable is NOVAS_FSDB_STRENGTH)

  - +fsdb+esdb (its corresponding environment variable is FSDB_ESDB)

- If the `+fsdb+gate` option is enabled, the `+strength` option in dumping tasks is ignored with a warning message.
- The FSDB Gate acceleration does not support VCS MVSIM Native flow to have the optimized performance.
- The FSDB utilities require many computations. A performance slowdown is expected when using the FSDB utilities.
- Siloti Data Expansion does not work with FSDB-GATE. The following message is displayed when a Data Expansion setup is applied on the FSDB-GATE FSDB:

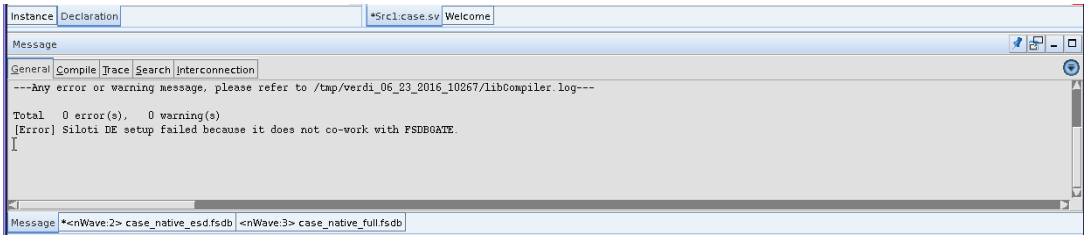


Figure: Warning Message Displayed -Siloti Data Expansion and FSDBGATE

Unified Transaction Debug- Verdi and Protocol Analyzer Integration

This section consists of the following subsections:

Introduction

Protocol Analyzer and Verdi are now integrated for VIP to improve the productivity in protocol, transaction, and signal-level debugging. With this integration, you can directly invoke Protocol Analyzer from Verdi and the protocol-related information is automatically loaded into Protocol Analyzer. After Protocol Analyzer is launched, it gets synchronized with Verdi automatically. You can also directly invoke Verdi with the loaded FSDB file from Protocol Analyzer. This invoking mechanism reduces the time consumed in setting different configurations for Verdi and Protocol Analyzer, and in comparing the corresponding objects at different levels. This also enables more efficient protocol-level analysis and signal-level debugging of issues, and increases the productivity of the debug process.

Additionally, transaction-based FSDB can be directly generated or converted from the result of VIP simulation and the FSDB files can be used in both Protocol Analyzer and Verdi.

The transaction debug capability offers the following features:

- Integrating Verdi and the Protocol Analyzer GUI
- Generating Transaction-Level FSDB File for VIP
- Reading Transaction-Based FSDB in Protocol Analyzer
- Loading Protocol Extension Files into Verdi

Use Model

For the details about the features, see the *New Transaction Debug Platform in Verdi* application note.

Unified UVM Library

This section consists of the following subsections:

Introduction

The unified UVM library is now provided to integrate the instrumented UVM libraries of VCS and Verdi. With the introduction of the unified UVM library, VCS and Verdi transaction recorder and message catcher now coexist and are compiled together. You can directly use the unified UVM library with the Verdi-provided recording mechanism during simulation, and for debugging with Verdi. Thus, accelerating the overall verification cycle. The unified UVM library also improves the debug productivity while debugging UVM-based environments with VCS and Verdi, as both the tools use the same UVM library. This eliminates the disparity between simulation and debug libraries.

Single compilation, UUM and UVM-VMM interoperability flows are supported in the unified UVM library. The unified UVM library can also be qualified and validated using Synopsys VIPs.

Use Model

For the details about its usage, see the New Transaction Debug Platform in Verdi application note.

Scope-Based Peak Analysis

The scope-based peak analysis is now provided to analyze which time has the most transitions (peak analysis) based on the scope for the design and FSDB file. The report of the analysis is displayed in the scope-based table view. You can also check the result in the waveform and in a Comma Separated Values (CSV) format. After checking the report, you can further generate the What-if configuration files to perform advanced power estimation based on the checking result.

The scope-based peak analysis capability offers the following features:

- Viewing reports in the scope-based table view
- Generating waveform of certain scopes
- Saving and restoring report as an XML file
- Dumping report in the Comma Separated Values (CSV) format
- Exporting the What-if configuration file

NOTE:

* Only a trigger at the top scope of the same net is counted.
* The following two triggered types are not counted as a trigger:
- A value changes to **X**
- **X** changes to a value

This section consists of the following subsections:

Use Model

To enable this feature, specify the following environment variable before starting the Verdi platform to enable this feature:

```
%> setenv TFV_SCOPE_PEAK_ANALYSIS 1
```

After loading your design and the FSDB file, use the **Tools -> Switching Analysis -> New Query** command to invoke the *Switching Analysis* form.

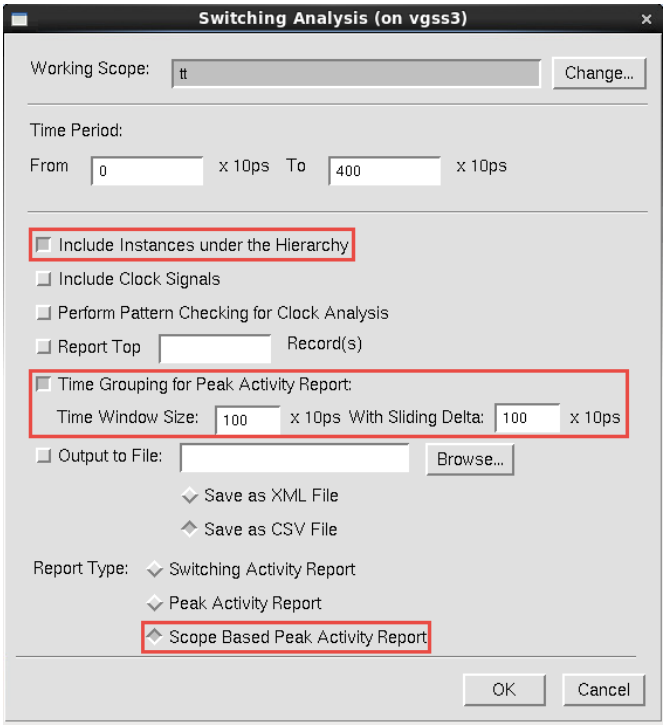


Figure: Switching Analysis Form

Enable the following options to configure the scope-based peak analysis for power estimation:

- **Include Instances under the Hierarchy**

  Includes sub-scopes of the specified scope to the analysis. If this option is not enabled, the analysis only includes the selected scope.

- **Time Grouping for the Peak Activity Report**

  Groups time in the report based on the time window size and sliding delta specified respectively in the **Time Window Size** and **With Sliding Delta** fields. It is strongly recommended to enable this option and specify the time window size and sliding delta.

- **Report Type: Scope Based Peak Activity Report**

  Enables to generate the scope-based peak activity report.

You can also generate the report with the following options:

- **Output to File**

Generates the output file with the specified file type (based on the enabled **Save as XML File** or **Save as CSV File** toggle options) for the report. The output file is generated while the analysis is completed and the report is ready.

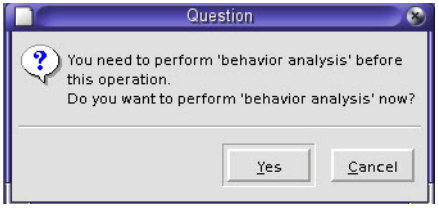
- **Save as XML File**

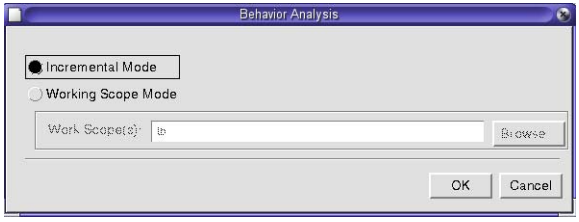
  When the file is saved as an XML format, the XML file can be loaded in the *Switching Analysis Report* form using the **File -> Load from XML file** command.

- **Save as CSV File**

  When the file is saved as a CSV format, the CSV file is readable and is used in the power estimation flow.

After completing the configurations, click the **OK** button in the *Switching Analysis* form and click the **Yes** button in the *Question* dialog to open the *Behavior Analysis* form.

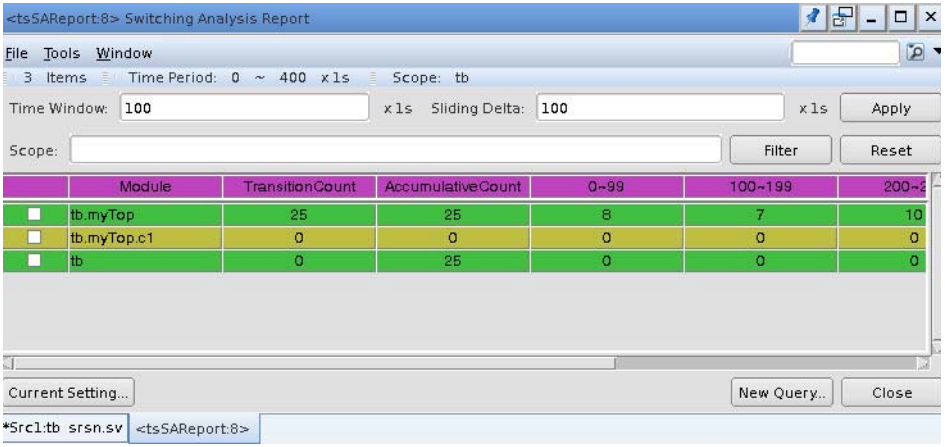




Click the **OK** button to perform the behavior analysis and the scope-based peak analysis.

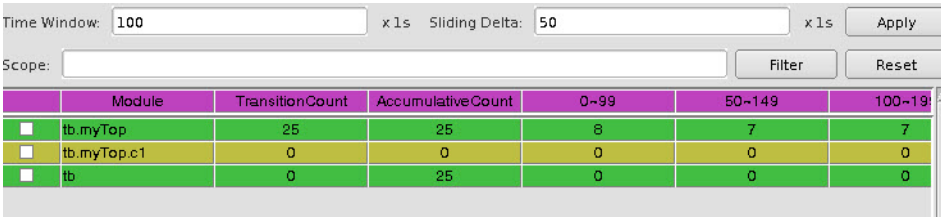
NOTE: If the behavior analysis is performed in the current working directory, the dialog box is not displayed.

The analysis report is shown in the invoked *Switching Analysis Report* frame.

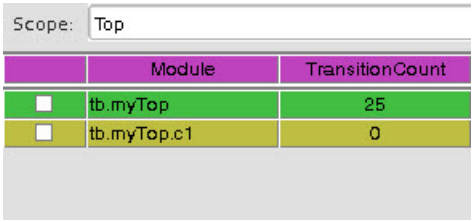


You can see the transition counts and accumulative counts for the module in the **Module**, **Transition Count**, and **Accumulative Count** columns accordingly.

The time windows, for example, **0~99**, **100~199**, **200~299**, are displayed based on the values specified in the **Time Window Size** and **With Sliding Delta** fields of the *Switching Analysis* form. You can also change them in the *Switching Analysis Report* frame.

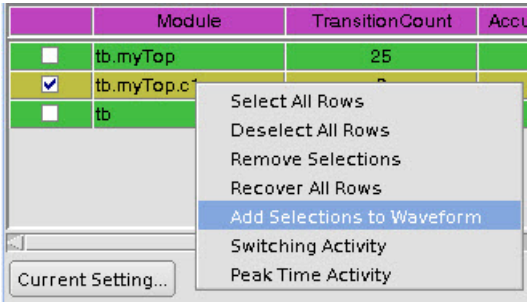


You can enter a scope name in the **Scope** filter and click the **Filter** button to select the scope..

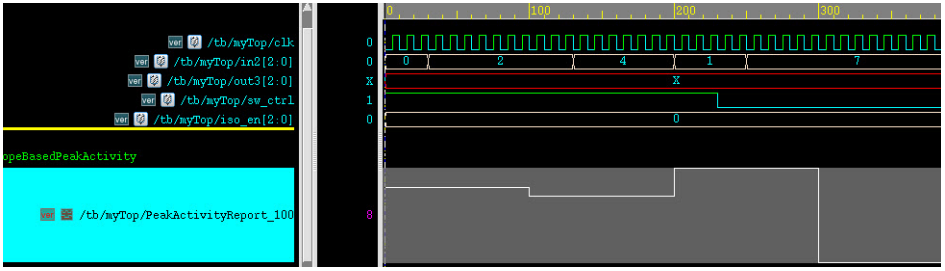


Generating Waveform With Report Entry

You can select a report entry and use the **Add Selections to Waveform** right-click command or the **Tools -> Add Selections to Waveform** menu command to add the selected entry into the *nWave* frame.



The selected entry is added in the *nWave* frame as a computed signal.

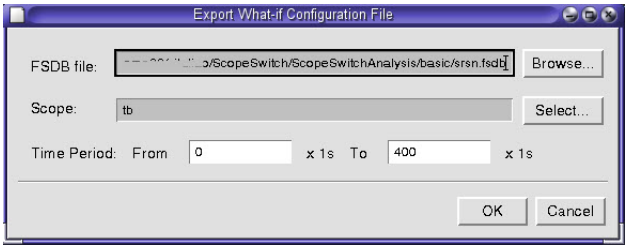


Exporting the What-if Configuration File

After checking the analysis report, you may want to perform the What-if correlation flow for power estimation with the specified scope and time duration. You can use the **File -> Export What-if Configuration File** command to generate the What-if configuration file and its related scripts that are needed in the What-if flow. The generated configuration file and script files make it easy for you to complete the configurations during the What-if flow.

To generate the configuration file and script files, perform the following steps:

1. In the *Switching Analysis Report* frame, use the **File -> Export What-if Configuration File** command to invoke the *Export What-if Configuration File* form.
2. Specify the FSDB file, scope, and time period in the corresponding fields.
3. Click the **OK** button.



The following configuration file and script files are generated accordingly:

- `wi_config_file`
  The configuration file for the What-if flow.
- `vcs_wi_compile.rc`
  The VCS compilation script used in the What-if flow.
- `vcs_wi_run.rc`
  The VCS simulation script used in the What-if flow.
- `ius_wi_compile.rc`
  The IUS compilation script used in the What-if flow
- `ius_wi_run.rc`
  The IUS simulation script used in the What-if flow

The following is the example of the configuration file and VCS script files:

```
//wi_config_file
   set FSDB          = ./srsn.fsdb
   set Scope         = tb
   set Map           = <Required>
   set Begin_Time    = 200
   set End_Time      = 249
   set Time_Unit     = 1s
   set Simulation_Compile_Script= ./vcs_wi_compile.rc
   set Simulation_Run_Script= ./vcs_wi_run.rc
   #For IUS user=====================================
   #set Simulation_Compile_Script= ./ius_wi_compile.rc
   #set Simulation_Run_Script= ./ius_wi_run.rc


//vcs_wi_compile.rc
   setenv VCS_HOME <cad tool path>/Synopsys/VCS_vE-2011.03-3
   setenv VERDI_HOME <Verdi_Install_path>
   setenv PATH ${VCS_HOME}/tools/bin:${VERDI_HOME}/bin:${PATH}
   setenv tab ${VERDI_HOME}/share/PLI/VCS/LINUX64/novas.tab
   setenv pli ${VERDI_HOME}/share/PLI/VCS/LINUX64/pli.a
   setenv LD_LIBRARY_PATH ${VERDI_HOME}/share/PLI/VCS/
   LINUX64:${VERDI_HOME}/share/PLI/lib/LINUX64
   alias vcs "\vcs -P $tab $pli +vcsd +memcbk +cli+4 -full64 -debug"
   vcs -sverilog -f wi_run.f


//vcs_wi_run.rc
   setenv VCS_HOME <cad tool path>/Synopsys/VCS_vE-2011.03-3
   setenv VCSI_HOME $VCS_HOME
   setenv LM_LICENSE_FILE 27005@sps403:$LM_LICENSE_FILE
   setenv VERDI_HOME <Verdi_install_path>
   setenv tab ${VERDI_HOME}/pliProd/PLI/VCS/LINUXAMD64/novas.tab
   setenv pli ${VERDI_HOME}/pliProd/PLI/VCS/LINUXAMD64/pli.a
   setenv LD_LIBRARY_PATH ${VERDI_HOME}/pliProd/PLI/VCS/
   LINUXAMD64:${VERDI_HOME}/pliProd/PLI/lib/LINUXAMD64
   alias vcs "\vcs -P $tab $pli +vcsd +memcbk +cli+4 -full64 -debug"
   ./simv
```