

# BugHunter Pro and VeriLogger Pro

User's Manual

[www.syncad.com](http://www.syncad.com)

## **BugHunter Pro and VeriLogger Pro Manual (rev 10.0) copyright 1999-2005 SynaptiCAD**

### Trademarks

- Timing Diagrammer Pro, WaveFormer Pro, TestBench Pro, VeriLogger Pro, DataSheet Pro, GigaWave Viewer, BugHunter Pro, Reactive TestBench Generation Option and SynaptiCAD are trademarks of SynaptiCAD Inc.
- Verilog-XL is a trademark of Cadence Design Systems, Inc.
- PinPort is a trademark of Exsent, Inc.
- Windows, Windows NT, Windows 95, Windows 98, and Windows 2000 are registered trademarks of Microsoft.

All other brand and product names are the trademarks of their respective holders.

Information in this documentation is subject to change without notice and does not represent a commitment on the part of SynaptiCAD. The software and associated documentation is provided under a license agreement and is the property of SynaptiCAD. Copying the software in violation of Federal Copyright Law is a criminal offense. Violators will be prosecuted to the full extent of the law.

No part of this document may be reproduced or transmitted in any manner or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the written permission of SynaptiCAD.

For latest product information and updates contact SynaptiCAD at:

web site: <http://www.syncad.com>

email: [sales@syncad.com](mailto:sales@syncad.com)

phone: (800)804-7073 or (540)953-3390





## Table of Contents

|   |           |
|---|-----------|
| <b>Table of Contents .....</b>  | <b>5</b>  |
| <b>Introduction .....</b>   | <b>7</b>  |
| <b>Chapter 1: A Quick Start to BugHunter Pro .....</b>                  | <b>8</b>  |
| Step 1: Setting Up Simulators .....                                     | 8         |
| Step 2: Creating a Project .....  | 8         |
| Step 3: Add Source Files to the Project .....                           | 9         |
| Step 4: Unit Level Testing (Drawing Stimulus) .....                     | 9         |
| Step 5: Build the Project .....   | 9         |
| Step 6: Debug Syntax Errors in Report Window .....                      | 10        |
| Step 7: Watch Signals and Components .....                              | 10        |
| Step 8: Simulate the Project .....                                      | 10        |
| Step 9: Debug With Breakpoints, Stepping and Inspecting Values .....    | 11        |
| Step 10: Save the Project, Code and Waveform Files .....                | 12        |
| <b>Chapter 2: Project Functions .....</b>                               | <b>13</b> |
| 2.1 Opening, Saving, and Starting New Projects .....                    | 13        |
| 2.2 Adding Files to the Project .....                                   | 13        |
| 2.3 Using the Project Window .....                                      | 14        |
| 2.4 Watching Signals and Components .....                               | 15        |
| 2.5 Simulator and Compiler Settings Dialog .....                        | 15        |
| 2.6 Project Simulation Properties Dialog .....                          | 16        |
| <b>Chapter 3: Simulation and Debugging Functions.....</b>               | <b>20</b> |
| 3.1 Build and Simulate .....  | 20        |
| 3.2 Simulation Button Bar .....   | 21        |
| 3.3 Displaying Errors and Simulation Results in the Report Window ..... | 22        |
| 3.4 Breakpoints .....   | 22        |
| 3.5 Inspect Values .....  | 23        |
| 3.6 Interactive Debugging in the Console Window .....                   | 24        |
| 3.7 Simulation Control Commands .....                                   | 25        |
| 3.8 Interpreted Verilog Simulators .....                                | 25        |
| <b>Chapter 4: Editor Functions .....</b>                                | <b>27</b> |
| 4.1 Opening, Saving, and Creating New Source Code .....                 | 27        |
| 4.2 Displaying or Finding a Specific Line of Code .....                 | 27        |
| 4.3 Using the Editor/Report Preferences Dialog .....                    | 28        |
| 4.4 Editor Cursor Commands .....  | 29        |
| 4.5 XEmacs Integration .....  | 30        |
| 4.6 Using an External Editor .....                                      | 32        |
| <b>Chapter 5: Waveforms and Test Bench Generation.....</b>              | <b>33</b> |
| 5.1 Stimulus and Results Diagrams .....                                 | 33        |
| 5.2 Drawing Waveforms for Stimulus Generation .....                     | 34        |
| 5.3 Working with the Diagram Window .....                               | 34        |
| 5.4 Waveform Comparisons .....  | 35        |
| 5.5 Generating and Reading VCD Files .....                              | 37        |
| <b>Chapter 6: VeriLogger Pro Command Line Simulator .....</b>           | <b>38</b> |
| 6.1 Preparing Verilog Source Files .....                                | 38        |


|   |           |
|---|-----------|
| 6.2 Using the Command Line Simulator .....                  | 38        |
| 6.3 Command Line Simulation Options .....                   | 39        |
| 6.4 Predefined Plus Options .....                           | 40        |
| 6.5 Simulator Control Commands .....                        | 41        |
| <b>Chapter 7: VeriLogger Pro SDF Support .....</b>          | <b>43</b> |
| 7.1 Using a Standard Delay File (SDF) .....                 | 43        |
| <b>Chapter 8: VeriLogger Technology Background.....</b>     | <b>44</b> |
| 8.1 Compilation Process .....                               | 44        |
| 8.2 User-Defined Primitives and Memory Usage .....          | 44        |
| 8.3 Notes on Using Specify Blocks .....                     | 44        |
| 8.4 IEEE-1364 LRM Standardization .....                     | 44        |
| 8.5 Implementation Differences from Verilog-XL .....        | 46        |
| <b>Appendix A: BugHunter System Tasks .....</b>             | <b>47</b> |
| <b>Basic Verilog Simulation .....</b>                       | <b>53</b> |
| Part 1: Project Management and Simulation .....             | 53        |
| 1.1) Add Files to the Project .....                         | 53        |
| 1.2) Build the Tree and Use the Editor Window .....         | 54        |
| 1.3) Simulate the Project .....                             | 54        |
| 1.4) Watch and View Internal Signals .....                  | 55        |
| 1.5) Save the Project, Waveforms and Source Code .....      | 55        |
| Part 2: Graphical Test Bench Generation .....               | 56        |
| 2.1) Remove TestBench Model and Clean Results Diagram ..... | 56        |
| 2.2) Build the Project and Examine the Black Signals .....  | 56        |
| 2.3) Use the Debug Run and Simulation Mode .....            | 57        |
| 2.4) How to Draw Waveforms .....                            | 57        |
| 2.5) How to Edit Waveforms .....                            | 57        |
| 2.6) Draw the Stimulus Waveforms .....                      | 58        |
| 2.7) Simulate Using the Auto Run Simulation Mode .....      | 58        |
| 2.8) Import and Generate Waveforms .....                    | 59        |
| 3) Breakpoints, Stepping and Tracing .....                  | 60        |
| <b>Index .....</b>  | <b>61</b> |

# Introduction

Thank you for purchasing BugHunter Pro or VeriLogger Pro with the BugHunter interface. BugHunter Pro is a graphical debugging system for Verilog, VHDL, and C++ simulators. BugHunter supports all the major HDL simulators. Additional simulators can be added to your BugHunter license by purchasing a simulator support package.

Chapters 1-5 cover the BugHunter graphical debugger interface. Chapters 6-8 cover the command line features specific to VeriLogger Pro.


# Chapter 1: A Quick Start to BugHunter Pro

This chapter will cover all the basic steps involved in creating a project and simulating in BugHunter. Please note that this is just a general guide for using the product. For more detailed information, refer to chapters 2-5. Also, several dialogs that you will use contain a context help feature that will allow you to learn more about the specific actions you are performing. This form of help is very useful in allowing you to have a hands-on introduction, as well as allowing you to jump directly to more in-depth information in the appropriate manual. The dialogs with context help have a small button with a question mark  in the upper right hand corner of the dialog (next to the close button). To use context help, click the question mark, and then click the item in the dialog about which you would like to see help.

## Step 1: Setting Up Simulators

BugHunter Pro needs to know where your VHDL/Verilog simulator or C++ compiler is located. If you are using VeriLogger Pro you can skip this section because the simulator was setup during installation.

The *Simulator/Compiler Settings* dialog contains the path settings for external tools. These settings are saved in the `syncad.ini` file each time the program is closed. To specify the path for each simulator or compiler that you will use:

- Choose the **Options > Simulator / Compiler Settings** menu option to open a dialog of that name.
- In the **Tools** drop-down choose your simulator or compiler.
- In the **Simulator Path** edit box either type in the path name or use the **browse button**  to search for the path.
- Continue to setup the paths for each tool that you are interested in using. When you are done click **OK** button to close the dialog.

Each of the main simulation languages has a default tool and program settings that are stored in the Project file. When you create a new project, the *project language* will determine which tools are used. Specify which tool to use and its' default settings:

- Choose **Project > Project Simulation Properties** menu option to open the *Project Simulation Properties* dialog.
- Choose the **Settings Template** radio button to indicate that you will be editing the default project settings for all future projects. These settings are saved in the INI file.
- Click on the language tab for the external tool that you are setting up.
- From the **Simulator Type** drop-down, choose the external tool.
- Choose the **Diagram Settings** radio button and edit the simulator that is used to simulate individual transactions (simulated signals in a *Diagram* window). By default this is setup to use an internal Verilog simulator, but if you are simulating in a different language set the simulator to your external simulator.
- Press the **OK** button to close the dialog.

## Step 2: Creating a Project

BugHunter uses a project file to control simulation settings and to list the set of files to be simulated. Projects are created, saved, and re-opened using the menu commands under the **Project** menu.

To create a project:

- Choose the **Project > New Project** menu item to open the *New Project Wizard* dialog.
- Specify the *project name* and *project directory*, and choose a language from the **Project Language** drop-down.
- Press the **Finish** button to create the project and populate the project tree.



### Step 3: Add Source Files to the Project

Once the project is created you can either create new model files using the built in editors or just add source files to the project.

To create a new source file:

- Choose the **Editor > New HDL File** menu option to open an editor window. Type in your source code and then save the file. Usually you will save the file in the project directory, but it is not required.

Add file to the project:


- Right click in the editor window and choose **Add to Project** from the context menu. The file name will be listed in the *User Source Files* folder of the *Project* window.
- Files can also be added by using the *Project* window context menu. Right click the **Source Code Files** folder, and select the **Add HDL File(s)** context menu option to open a *File* dialog. Choose your source code files and close the dialog.

The **Editor** menu contains functions that act on the editor windows. Double clicking on a source file name in the *Project* window automatically launches an editor window. For more information about the *Editor* window read *Chapter 4*.

### Step 4: Unit Level Testing (Drawing Stimulus)

If your top-level module has input ports, BugHunter can take drawn waveforms and generate a test bench model that can be used to test your model. The *VeriLogger Basic Verilog Simulation* tutorial (part 2) demonstrates this feature.

The basic steps for unit level testing:


- Press the **Parse MUT** button  to extract the port signal names and sizes and put them in the **Stimulus and Results** diagram.
- Draw waveforms on the black input signals. The gray signals are outputs of the MUT and will turn purple after the simulation begins. If you have the **Reactive Test Bench** option then all of the signals will come in as black so that you can draw expected response from the MUT (which will draw in blue).
- Each time a simulation is run (see *Step 8: Simulate the Project*), BugHunter will create a test bench module from the drawn waveforms. A wrapper module that hooks up the test bench module to the design model is created at the same time.

BugHunter has two simulation modes, **Auto Run** and **Debug Run** that determine when a simulation is performed. The current simulation mode is displayed on the leftmost button on the simulation button bar. In the **Debug Run** simulation mode, simulations are started only when the user clicks the **Run** or **Single Step** buttons (similar to a standard HDL simulator). In the **Auto Run** simulation mode, the simulator will automatically run a simulation each time a waveform is added or modified in the *Diagram* window. This mode makes it easy to quickly test small modules and do bottom-up testing. Click the mode button to toggle between the two simulation modes.

### Step 5: Build the Project

When files are first added to the project, you can see the file name but you cannot see a hierarchical view of the modules inside the files. To view the internal modules on the project tree you must first **build** or **run** a simulation. The **build** command compiles the user source files and builds the model tree. It does not run a simulation. For large projects **build** allows you to quickly construct the tree without having to wait for a simulation to run.

There are three ways to **build** a project:

- Click the yellow **Build** button  on the simulation button bar, select the **Simulate > Build** menu option OR press the <F7> key.

After you build the project, the signals or the ports in the top-level module are automatically added to the *Stimulus and Results* window. In the *Project Window* you can also view all the modules, signals, ports, and components in the user source files. One module name will be surrounded by brackets <<<name>>>. This is the top-level module for the project. It is the highest-level instantiated component. All sub-modules can be viewed by descending the top-level module's tree.

## Step 6: Debug Syntax Errors in Report Window

The simulation status indicator in the lower right corner displays success/failure information about the last simulation or build. If **Simulation Error** or **Compile Error** are displayed on the button, then there is an error in the user source code files in the project.

If your build fails, there are two different files in the *Report* window you can check to find out why the simulation failed. The first file is the **simulation.log** file. This file displays all the available information about the current simulation. The second place is the **Error** tab in the *Report* window. This tab displays the errors in a more concise manner.

To jump to a particular error:

- Click the **Error** tab in the *Report* window to display the error data.
- Double-click on an error. This will open the offending file in the *Editor* window and place the cursor at the error line.

If you have used the automatic test bench features, such as direct entry of Boolean equations in the *Signal Properties* dialog, then a third file, **waveperl.log**, will help you locate any errors related to code generation. This file is covered in *Section 12.5: Simulation Errors of the Diagram Editor & Universal Features* documentation.

## Step 7: Watch Signals and Components

BugHunter has two main output displays for the results of a simulation. The **simulation.log** file in the *Report* window captures simulator messages and any output from display statements embedded in the code. The *StimulusAndResults* diagram displays the waveform results of the simulation.

To view the log file:

- Click on the **simulation.log** tab in the *Report* window. If you cannot see the *Report* window, select the **Window > Report** menu option to bring the window to the front.

The watched signals are displayed in the Stimulus and Results diagram window. To add new watch signals to the diagram:

- Click on the plus sign to the left of the top-level module <<<name>>>. This will expand the top-level tree.
- Continue to open the top-level tree until you are able to see a signal or component that you would like to watch.
- Right-click on the signal or component to open the context menu for that item.
- Select the **Watch Connection** or **Watch Component** menu option. This will add the signals with a full path reference to the *Diagram* window. If you are watching a component, then all the top-level signals of that component will be added to the *Diagram* window.

The Stimulus and Results diagram saves the list of watched signals and simulation results. *Chapter 5: Waveforms and Test Bench Generation* has more information about switching and archiving stimulus and results diagrams.

## Step 8: Simulate the Project

To simulate the project:

- Click the **Run** button  on the simulation button bar,

- Select the **Simulate > Run** menu option,
- OR, press the <F5> key.

Once a project is simulated the waveforms will be displayed in the *Stimulus and Results* diagram. If you do not want to continue to watch a particular signal, click on the signal name in the *Diagram* window and press the <Delete> key.

## Step 9: Debug With Breakpoints, Stepping and Inspecting Values

BugHunter supports graphical breakpoints, single step debugging and the inspection of signal values while stepping through a simulation.


To create a source line breakpoint:


- Open the Verilog file in an editor window, either by double-clicking in the *Project* window or by selecting the **Editor > Open HDL Source** menu option.
- In the *Editor* window, notice the gray strip to the left of the source code. This is the breakpoint window.
- Click on the breakpoint window at the line of code that you wish to stop the simulation. This will do two things: first, a red dot will be added to the breakpoint editor window; second, the breakpoint will be listed in the **breakpoint** tab in the *Report* window.

To create a time breakpoint:

- Open the **Breakpoints** tab in the *Report* window.
- Right click and choose **Add Breakpoint** from the context menu.
- Change *Type* to **Time** and enter the time you wish to break at.


There are several single-step buttons:


 The **Step Into** and **Step Into With Trace** buttons have a small triangle between two medium size triangles to indicate that you are stepping to the next line of code to execute. The trace function logs the values of the executed line in the *Report* window.


 The **Step Over** button has two medium size triangles to indicate that it will not step into a function call or task.

It is best to use the single step buttons in conjunction with breakpoints, so that if you step into a loop you can use the **Run** button to simulate to the next breakpoint.

There are also three other buttons that will help you debug your simulation:

 The **End** button exits the current simulation.

 The **Goto** button opens an *Editor* window and displays the line that will be executing next.

 The **Pause** button temporarily pauses simulation.

The *Console Window*, on the simulation button bar, accepts simulator commands that are supported by your simulator. For interpreted simulators you can also enter SystemTasks or blocks of code to execute.

The *Inspect Values* dialog can be used to view signal and variable values for the current and past simulation times.

- Choose **Simulate > Inspect Values** menu option to open the dialog.
- Type in a signal or variable name and the value will be displayed to the right.

- The variables values are by default shown for the current scope. You can use the “S” top-level scope button and “s” local scope buttons to change the scope of the inspected values.

The *Editor* window can also be used to inspect values. During a simulation you can use the mouse to hover over a variable and a flyout will appear that shows the value of the variable.

## Step 10: Save the Project, Code and Waveform Files

In BugHunter there are three types of files associated with a project.

- Project files have an extension of **hpj** and are saved by using the **Project > Save HDL Project** menu option. This saves the file list and simulation options. It does not save the watch settings.
- HDL Source code files usually have an extension of **v**, **vhd** or **cpp** (depending on the language) and are saved by selecting the editor window and choosing the **Editor > Save HDL Code** menu option.
- Stimulus and Results diagram files have an extension of **BTIM** and are saved using the **File > Save Timing Diagram** menu option. This file saves any watched signals.

Saving watched signals in separate diagram files allows you to build several different test cases so you can compare and contrast future simulation results.

## Chapter 2: Project Functions

BugHunter Pro uses a project file to store simulation options and the list of files to be simulated. Most of the project level features like saving, opening, creating, and editing the settings are accessed through the **Project** menu options. The *Project* window right-click context menus give access to functions that can be applied to a specific node in the tree like setting watches on signals and viewing source code files.

### 2.1 Opening, Saving, and Starting New Projects

Projects are opened and saved using the **Project** menu options.

To open an existing project:

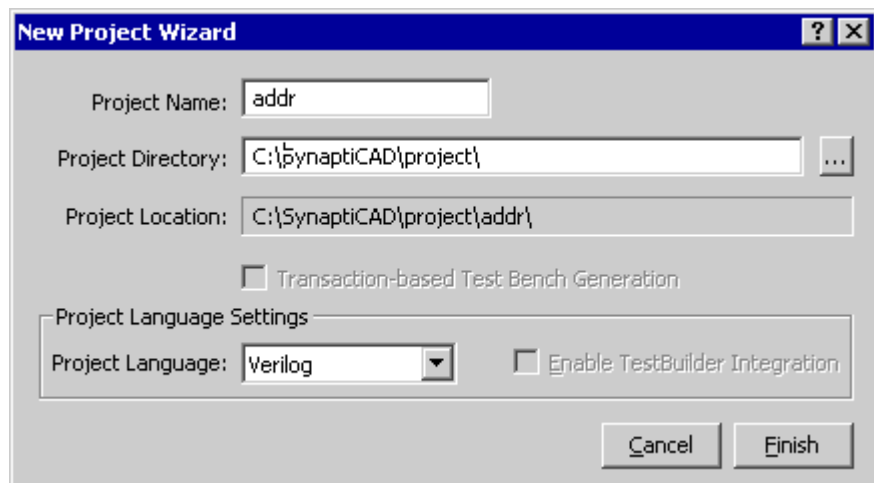
- Select the **Project > Open Project** menu option. This opens the *Open Project File* dialog where you can select a project file to open.

To save an open project:

- Select the **Project > Save Project** menu option to open a *Save* dialog. By default, project filenames have an extension of ".hpj".

To clear the current project and start a new project:

- Select the **Project > New Project** menu option. This will bring up the *New Project Wizard*.
- Type the name of the project in the **Project Name** edit box.
- Type the base path for the new project in the **Project Directory** edit box. Note that the **Project Location** displays the full path to the project. BugHunter will create a directory that is named after the project at the end of the path specified in the *Project Directory* edit box.
- Click the **Finish** button. This will create a new project.



### 2.2 Adding Files to the Project

The first step in setting up a project is to add user source files to the project.

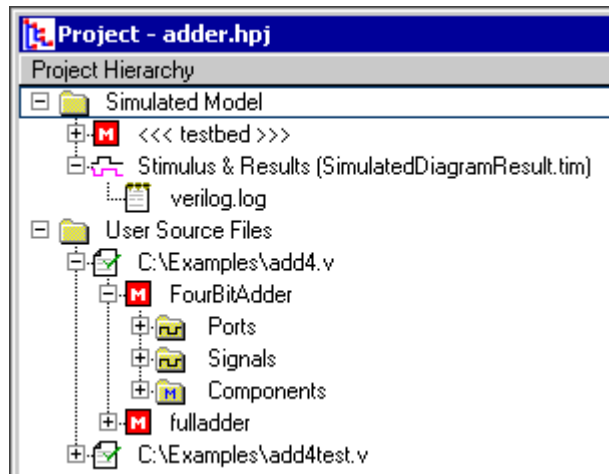
To add a file to the project:

- Right-click the *User Source Files* folder in the *Project* window to open the pop-up context menu, and select the **Add HDL File(s) to User Source File Folder** menu option,
- OR, select the **Project > Add User Source File(s)** menu option.
- Both of these functions open the *Add File* dialog. Select one or more files to add to the project and click the **Open** button to close the dialog.

When files are first added to the project, you can see the filename but you cannot see a hierarchical view of the modules inside the files. To view the internal modules on the project tree you must first **build** the project or **run** a simulation as described in *Chapter 3: Simulation and Debugging Functions*.

## 2.3 Using the Project Window

The *Project* window can be used to open source code editors, set watches on signals, and set the Stimulus and Results diagram. After a project is built as described in *Chapter 3: Simulation and Debugging Functions*, the *Project* window can be used to investigate the hierarchical structure of the design. Each node in the tree has a context sensitive pop-up menu that can be opened by right clicking on the node.



Expanding or hiding branches of a tree:

- Click + or - to expand or close a node. This will not open sub-nodes.

Viewing source code:

- Double-click on a filename to open the file in a new *Editor* window
- Double-click on a signal or component to open an *Editor* window scrolled to the declaration in the HDL source code.

### The Stimulus & Results Diagram

The diagram under the *Stimulus&Results* folder is the diagram that will be used to display the waveform results of the simulation. This diagram can also be used to generate a unit level test bench as described in *Chapter 5: Waveforms and Test Bench Generation*.

To set a new diagram as the Stimulus & Results diagram:

- Right-click on the *Stimulus & Results* folder of the *Project* window and select **Replace Current Result Diagram** from the context menu. This will open a *File* dialog where you can choose the new file.

OR

- Open the diagram you wish to use.
- Right-click in the *Label* window and select **Set Diagram as Stimulus and Results** from the context menu.

## 2.4 Watching Signals and Components

After compiling the project, use the *Project* window to pick signals to be watched and place them in the Stimulus and Results diagram. To maximize simulation speed, simulators do not automatically store signal transition times unless a signal is specifically tagged as one to watch.

To set a signal, component, or port to be watched:

- Expand the tree until you locate the signal or component that you would like to watch.
- Right-click on the component to open the context menu for that item.
- Choose one of the **Watch** menu options. The watch menu options vary according to what type of object is selected.
- Notice that one or more signal names have been added to the *Diagram* window. The full path name of the signal is used in the *Diagram* window. The waveform data will be displayed during the next simulation run.

**Watching a signal in a specific instance of a module:** Expand the tree starting from the top-level module <<<name>>>. By expanding from the top-level tree you will be able to watch signals in specific instances of a module.

**Automatic Watches in the "Simulate Diagram With Project" mode:** After you build the project, the signals or the ports in the top-level module are automatically added to the *Diagram* window. If the top-level module does not have port signals, the internal signals of the module are viewed. If the top-level module has port signals, the output ports are viewed as purple signals and input ports are viewed as black signals. You can edit the black input signals to provide stimulus to the top-level module. The waveform drawing functions are covered in Chapter 1 of the *Diagram Editor & Universal Features* manual of the on-line help (select the **Help > Timing Diagram Editor Help** menu option to view the on-line help).


The automatic port watching feature can be turned off by unchecking the **Grab top level signals** checkbox located in the *Project Simulation Properties* dialog (select the **Project > Project Simulation Properties** menu option to open this dialog).

To stop watching a signal:

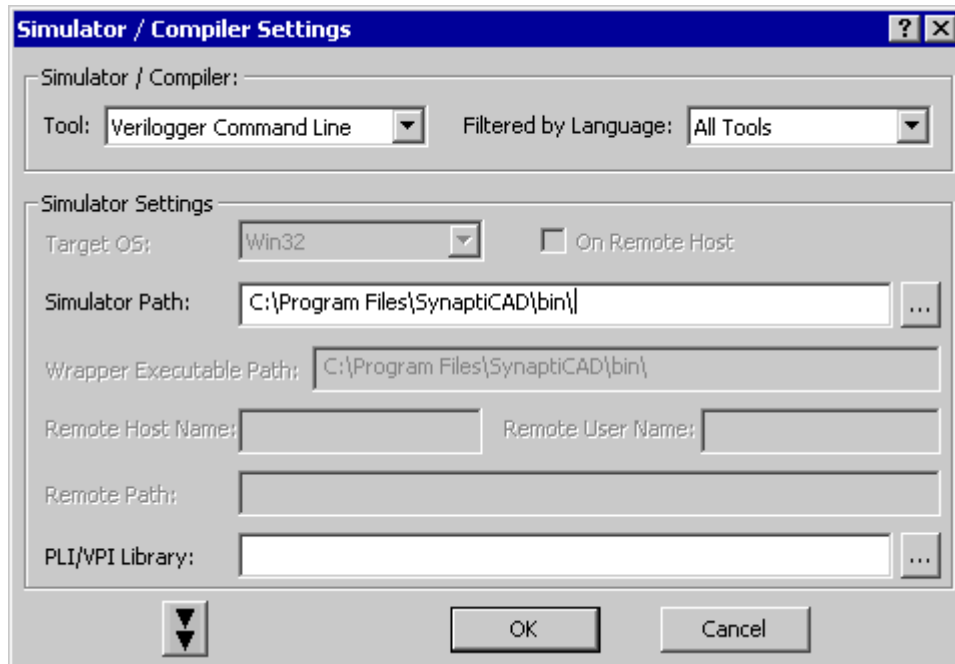
- In the *Diagram* window, double-click on the signal name to open the *Signal Properties* dialog.
- Change the signal type from *Watch* to **Drive** or **Compare**.

## 2.5 Simulator and Compiler Settings Dialog

The external simulator and compiler paths are set using the *Simulator / Compiler Settings* dialog. These settings are saved in the syncad.ini file. Before you simulate, you will also need to set which tool to use for a particular project by using the *Project Simulation Properties* dialog covered in *Section 2.6: Project Simulation Properties Dialog*. To change the path information for external tools:

- Choose the **Options > Simulator / Compiler Settings** menu option to open the *Simulator/Compiler Settings* dialog.
- In the **Tools** drop-down choose your simulator or compiler.
- In the **Simulator Path** edit box either type in the path name or use the **browse button**  to search for the path.

- Continue to setup the paths for each tool that you are interested in using. When you are done click the **OK** button to close the dialog.

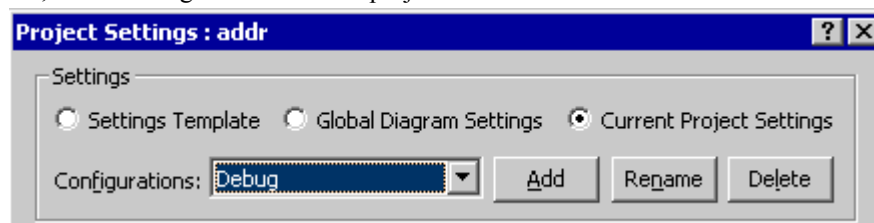


## 2.6 Project Simulation Properties Dialog

The *Project Simulation Properties* dialog determines the simulator run time options and which simulator to use for projects and diagrams. This information is stored inside the project HPJ file and the INI file. To open the Project Simulation Properties dialog:

- Select the **Project > Project Simulation Properties** menu option to open the dialog.

The top half of the dialog determines if you are editing the default settings that affect new projects, the settings for simulating diagrams, or the settings for the current project.



- If the **Settings Template** radio button is selected, then you are editing the default settings that are used by new projects. These are stored in the INI file each time the program is closed. The **Restore Default Templates** button at the bottom of is used to reset the INI file to the factory default settings for this dialog.
- If the **Global Diagram Settings** radio button is selected, then you are editing the options for how transactions are simulated (simulated signals in a *Diagram* window). These are stored in the INI file.
- If the **Current Project Settings** radio button is selected, then you are editing the project settings for the current project. These settings are stored in the Project HPJ file when you save the project.
- By default the *Settings Template* and the *Current Project Settings* use the **Debug Configuration**. If you are moving projects to different machines or if you want to have different settings for debugging and releasing a



project you may want to create a new configuration to store the different settings. If you need to define a new configuration:

- Press the **Add** button to open the *Add New Configuration* dialog, that lets you specify a name and the default configuration to copy the settings from.
- **Rename** button lets you change the name of the current configuration.
- **Delete** removes the current configuration.
- Use the **Configurations** drop-down to choose which configuration you will be editing.

The **General** tab contains simulation options that are standard across all of the simulators.

- The **Grab Top Level Signals** check box turns on the automatic monitoring of ports or internal signals in the top-level module.

- The **Capture and Show Watched Signals** check box enables the display of waveform results from a simulation run.

- The **Dump Watched Signals** check box will generate a dump file for any watched signals in the diagram. The generated file will have the same name as the .btim file, only with an extension of .VCD.

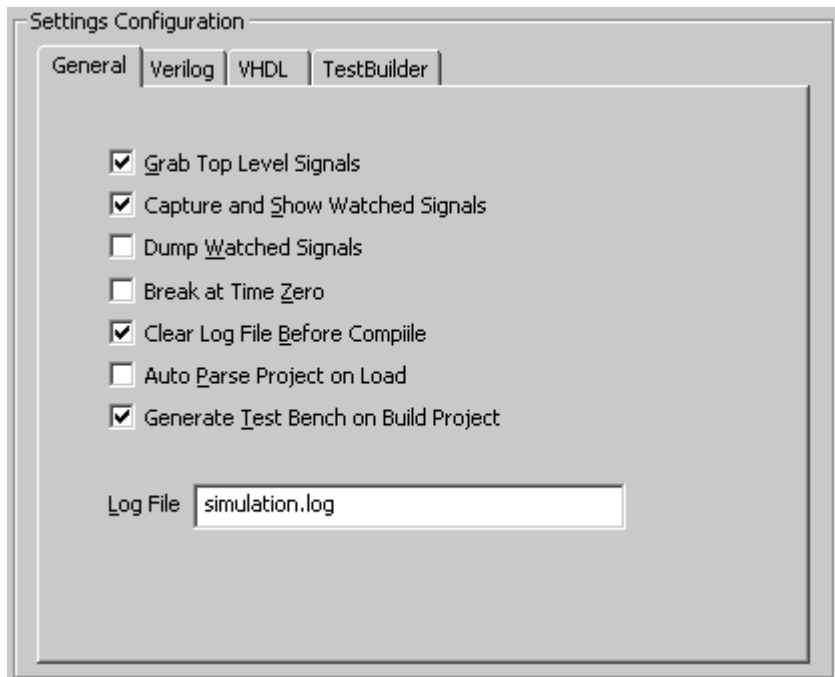
- The **Break at Time Zero** check box is the equivalent of setting a breakpoint at time zero. This starts the simulator and allows you to enter commands into the console window that will be executed during simulation.

- The **Clear Log File Before Compile** checkbox clears the simulation log just prior to a new compilation being performed. This log maintains compilation notes, as well as some simulation notes. Note that in this dialog you can also change the name of this log (see Logfile below).

- When the **Auto Parse Project on Load** box is checked, user source files are automatically parsed and built when the project is loaded. The top-level module is the first module that is not included by another for Verilog; it is the first entity/architecture pair parsed for VHDL. This is mainly used by Actel Libero customers with WaveFormer Lite.

- The **Generate Test Bench on Build Project** automatically updates the test bench for changes to timing diagrams. Turn this off if you want to temporarily change some of the generated source code manually or to avoid updating the test bench on diagram changes.

- The **Log File** specifies the name of the log file that receives all the simulation results and information. By default BugHunter uses *simulation.log*.



The **Verilog** tab specifies the simulator and simulation options used for Verilog projects.

- The **Simulator Type** drop-down determines the simulator.

- The **Simulator Settings** button opens the *Simulator / Compiler Settings* dialog where you can edit the simulator paths.

- **Include Directories** edit box specifies the directories where BugHunter searches for included files. The following is a Windows example (Unix users should use / slashes):

```
C:\design\project;c:\design\library
```

- The **Library Directories** edit box lists the path and directories where the program searches for library files. BugHunter will try to match any undefined modules with the names of the files that have one of the file extensions listed in the *Lib Extensions* edit box. The simulator does not look inside a file unless the undefined module name exactly matches a file name. The simulator does not look at any files unless there are file extensions listed in the Lib Extensions edit box. The following is a Windows example (Unix users should use the / slashes):

```
C:\design\project;c:\design\library
```

- The **Lib Extensions** edit box specifies the file name extension used when searching for library files in the library directory. Each library extension should begin with the period character followed by the extension name. Use a semicolon to separate multiple file extensions.

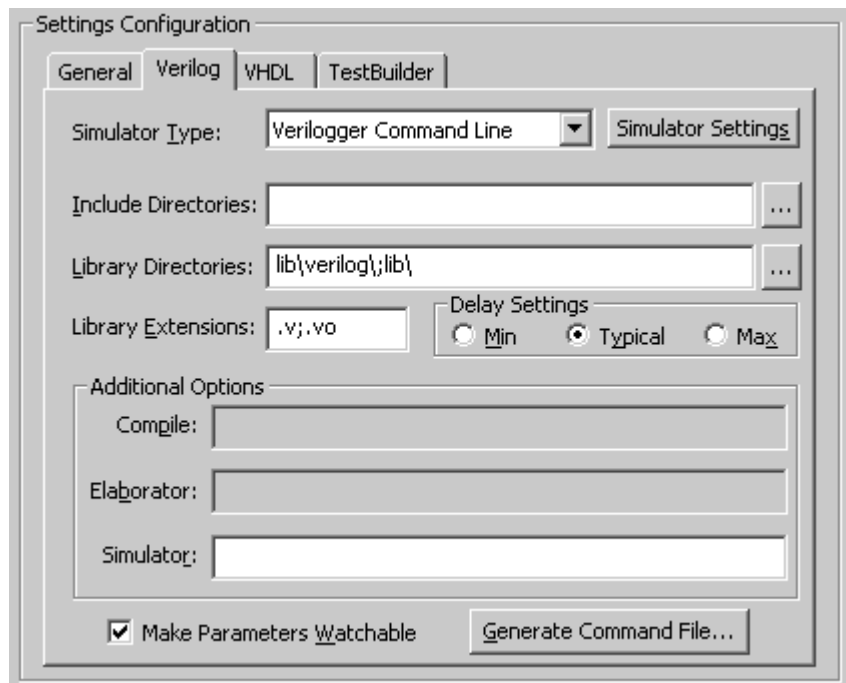
```
.v;.vo
```

- The **Delay Settings** radio buttons determines which delay value is used in min:typ:max expressions. These settings are output as either the **+maxdelays**, **+mindelays**, or **+typdelays** command line simulator option.

- **Compile, Elaborator, and Simulator** option edit boxes allow you to write additional command line options that will be passed to the tool when it is run. Most simulators do not support all three phases of command line options.

- When the **Generate Command File** button is pushed, the text contained in the Simulator Options edit box along with the list of Verilog files specified in the *Project* window are written to a Command File. This file can then be used with the Command Line version of your simulator to run a simulation without the BugHunter GUI.

- The **Make Parameters Watchable** determines whether or not parameters will be included with the automatic monitoring of ports and internal signals in the top-level module.

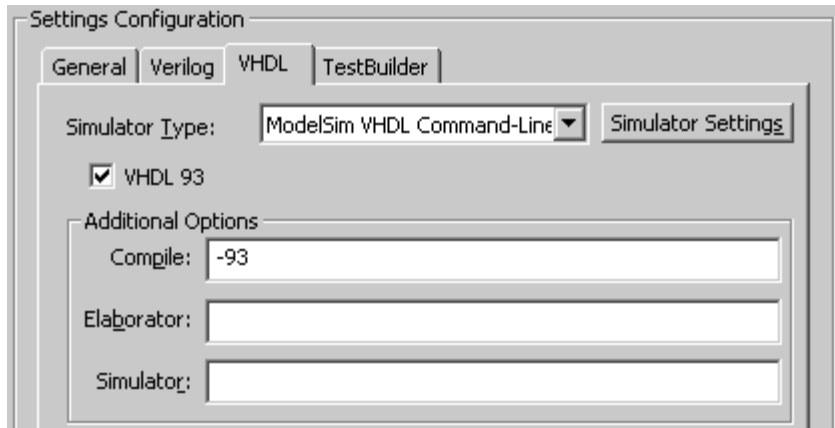


The **VHDL** tab contains the simulation options and simulator used for VHDL projects.

- The **Simulator Type** drop-down determines the simulator.

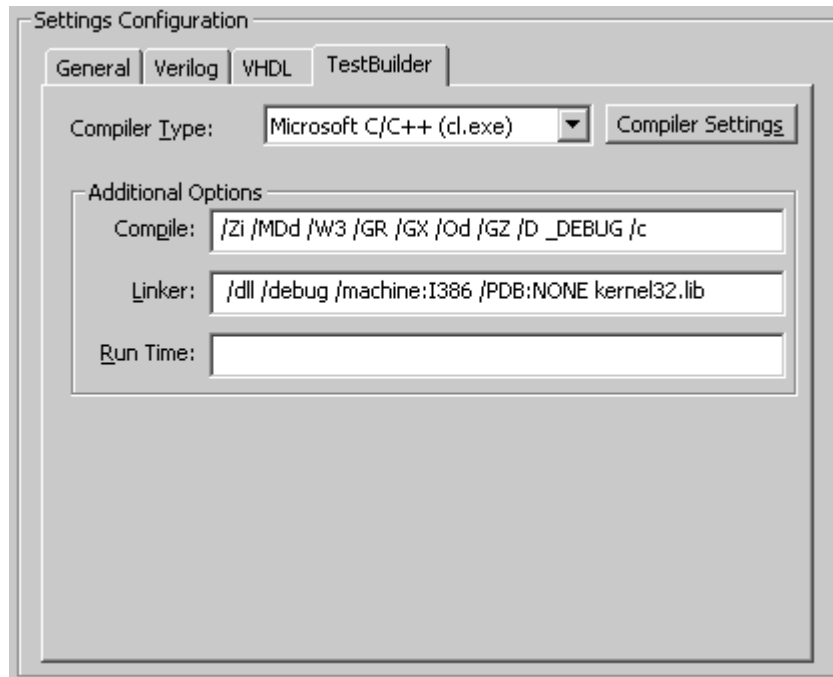
- The **VHDL 93** checkbox specifies that the project dialect for the generated files is **VHDL 93**.

- The **Simulator Settings** button opens the *Simulator / Compiler Settings* dialog where you can edit the simulator paths.
- The **Compile**, **Elaborator**, and **Simulator** options edit box allow you to write additional command line options that will be passed to the tool when it is run. Most simulators do not support all three phases of command line options.



The **TestBuilder** tab contains the compiler options and compiler used for C++ projects.

- The **Compiler Type** drop-down determines the C++ compiler.
- The **Compiler Settings** button opens the *Simulator / Compiler Settings* dialog where you can review and edit the compiler paths.
- The **Compile**, **Linker**, and **Run Time** options edit box allow you to write additional command line options that will be passed to the tool when it is run.



# Chapter 3: Simulation and Debugging Functions

BugHunter Pro uses the simulation button bar to compile, simulate, and debug the design. The simulation button bar also has an interactive command console window that can be used to enter simulator commands to observe and control variables and models during simulation. For interpreted simulators, the command console accepts system task calls and behavioral statements that can be used within *initial* and *always* blocks.


The *Report* window manages several tab windows - three of which are important to simulation and debugging: **simulation.log**, *Errors*, and *Breakpoints*. The **simulation.log** file displays the default log file for the simulator. The *Breakpoints* tab displays all of the breakpoints that are set in the project files. And the *Errors* tab displays any compile or simulation bugs that are found in the design.

BugHunter also provides mouse over inspection and an *Inspect Values* window for inspecting the values of variables and signals during a paused simulation. In an editor window, place the mouse over a variable to see the value at the current simulation time. The *Inspect Values* window can be used to inspect current and previous values of a variable.

## 3.1 Build and Simulate

When files are first added to the project, you can see the filename but you cannot see a hierarchical view of the modules inside the files. To view the internal modules on the project tree you must first build or run a simulation. The build command compiles the source files and builds the hierarchical tree. It does not run a simulation. For large projects, build lets you quickly construct the tree without having to wait for a simulation to run.

There are three ways to build a project:

- Click the yellow Build button  on the simulation button bar,
- Select the **Simulate > Build** menu option,
- OR, press the <F7> key


The status of the build is reported in the lower right hand corner of the screen:

**Simulation Started** means that the compile succeeded and you are ready to simulate

**Compile Error** means that the compile failed and the syntax errors will be listed in the *Error* tab of the *Report* window (see *Section 3.3: Displaying Errors and Simulation Results in the Report Window*).

After the project is successfully built, the *Project* window displays a hierarchical view of all the modules, signals, ports, and components in the source files. One module name will be surrounded by brackets <<<name>>>. This is the top-level module for the project. The top-level module is automatically set to the first *non*-instantiated component found. You can select another component as the top-level by right-clicking on it in the project tree. All sub-modules can be viewed by descending the top-level module's tree.

There are three ways to run the simulation:

- Click the Run button  on the simulation button bar.
- Select the **Simulate > Run** menu option, OR
- Press the <F5> key.

During Simulation the status of the simulation is displayed in the bottom right hand corner of the BugHunter Pro main window. When the simulation is complete, **Simulation Good** will be displayed in the status bar and the following information will be displayed:

- The simulation waveforms will be displayed in the *Stimulus and Results* diagram window (See *Section 5.1: Stimulus and Results Diagram*).
- The simulation log file, **simulation.log**, is shown in the *Report* window. Any notes, warnings and errors reported by the simulator will appear in this log.

In the *Project* window, the *Simulated Model* folder contains the compiled top-level module, the MUT and the Stimulus & Results diagram. If there are any archived stimulus/results files, they will be in the Stimulus & Results Archive folder. If any extra files are necessary for BugHunter to build the project, they will be automatically added to the project, and contained in the *Compiled Library Files* folder.











### 3.2 Simulation Button Bar

The simulation and debugging functions in BugHunter are accessed from the simulation button bar located at the top of the main window.



One of the unique features of BugHunter is that it has two simulation modes: **Auto Run** and **Debug Run**. The active simulation mode is displayed on the left most button on the simulation button bar. In **Debug Run** mode, simulations are started only when the user clicks the **Run** or **Single Step** buttons (similar to a standard simulator). In **Auto Run** mode the simulator will automatically run a simulation each time a waveform is drawn or changed in the *Diagram* window. This mode makes it easy to quickly test small modules and do bottom-up testing. Click the mode button to toggle between the two simulation modes.



-  **Parse MUT** - parses the source code files and extracts the MUT and port information into the current diagram.
-  **Build** - compiles the project files, builds a testbench if necessary and builds the hierarchical tree and populates the Stimulus and Results diagram. It does not run a simulation.
-  **Run/Resume** - compiles the files (if necessary) and runs a simulation. This button also continues a simulation when it is currently paused.
-  **Step Over calls** - steps to the next line of code. It does not step into function calls.
-  **Step Into** - steps to the next line of code and will also step into function calls.
-  **Step Into With Trace calls** - steps to the next line of code and also sends a trace statement to the **simulation.log** file. This button will also step into function calls.
-  **Pause** - stops the simulation and places the simulator into interactive debugging mode. This button is only active during a simulation.
-  **End** - exits the simulation.
-  **Goto** - opens an editor at the line that will execute next. Use this button when the simulation is stopped.
-  **Top Scope** - changes the interactive scope for console commands and *Inspect Values* window to the scope of the top-level module.

**S Local Scope** – changes the interactive scope for console commands and *Inspect Values* window to the scope of the line that will execute next.

**Console Window** (the drop-down edit box) accepts simulation commands and special simulator commands. Most of this chapter is dedicated to describing the types of commands accepted by the console window.

### 3.3 Displaying Errors and Simulation Results in the Report Window

The *Report* window manages several tab windows, three of which are important to simulation and debugging, **simulation.log**, **Errors** and **Breakpoints**.



The first important tab is the *simulation.log* tab. This tab contains the default log file for BugHunter. All information generated by the simulator, such as compiler messages, and all user-generated messages from \$display tasks and traces are sent to this file. During a simulation run you should watch the simulation.log file for important messages. Click on the **simulation.log** tab to view this file.

The second important tab is the *Errors* tab. If the compiler discovers an error, a message is returned to the *simulation.log* tab and the *Errors* tab. The *simulation.log* tab displays all simulation information and finding a specific error can be difficult. Using the *Errors* tab enables you to quickly view all simulation errors. Double-clicking on an error in the *Errors* tab will open an editor starting at the line of source code where the error was found.

| Report - Errors |                                  |        |                                   |
|-----------------|----------------------------------|--------|-----------------------------------|
|                 | File Name                        | Line # | Error                             |
| 0               | C:\Program Files\SynaptiCAD\E... | 1      | Files\SynaptiCAD\Examples\Verilog |
| 1               | C:\Program Files\SynaptiCAD\E... | 148    | C:\Program Files\SynaptiCAD\Exar  |
| 2               | C:\Program Files\SynaptiCAD\E... | 150    | C:\Program Files\SynaptiCAD\Exar  |

The third important tab is the *Breakpoints* tab, which is discussed in *Section 3.4: Breakpoints*.

### 3.4 Breakpoints

BugHunter supports both source code and time based breakpoints that can be used to pause the simulation. A complete list of the Project's Breakpoints is displayed in the *Breakpoints* tab in the *Report* window. Source code Breakpoints are also displayed in the *Editor* windows to the left of the source code.

To add a source code breakpoint to a line in the source code using an *Editor* window:

- In an *Editor* window, click on the gray line on the left side of the window. This adds a breakpoint, indicated by the red circle on the line. It also adds a breakpoint listing to the *Breakpoints* tab in the *Report* window.

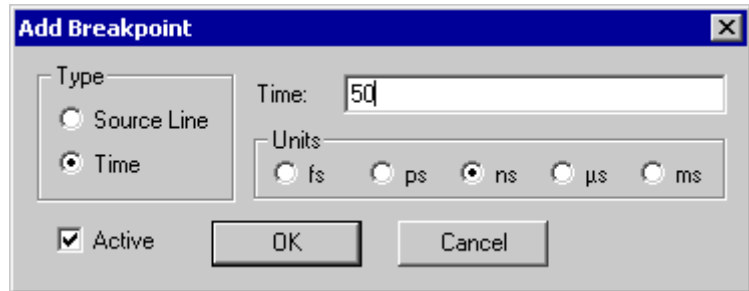
```
integer i;
initial                                     //initialize all
begin
  DBUS_driver = 8'bzzzzzzzz;
  for (i=0; i < 4095; i = i + 1)
    ram[i] = 0;
end
```

To add a time breakpoint or source code breakpoint using the *Breakpoints* tab.

- Right-click anywhere in the *Breakpoints* tab window and select the **Add Breakpoint** option from the pop-up menu to the *Add Breakpoint* dialog.

- For time based breakpoints, choose the **Time** radio button and enter a time to stop the simulation.
- For source code breakpoints, choose the **Source Line** radio button and enter the full path name of the file and the line number to enter the breakpoint.

There are several mouse-oriented commands that you can do inside the *breakpoints* tab:



- Double-clicking on a source code breakpoint will open an editor starting at that line in the source code.
- Right-clicking anywhere in the tab window will open a menu allowing you to add, edit, or delete breakpoints.
- Clicking on a red breakpoint button will toggle it between active and inactive states. An inactive breakpoint is displayed as a small red circle and is ignored in a simulation.

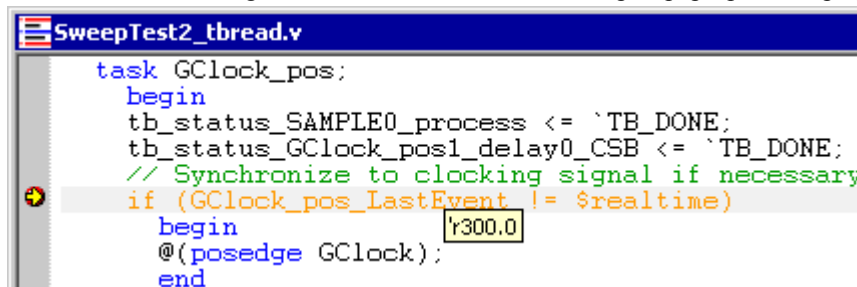
|   | Type          | File Name            | Line # | Time(ns) | Expression |
|---|---------------|----------------------|--------|----------|------------|
| 0 | ● Time        |                      |        | 50       |            |
| 1 | ● Source Line | C:\SynaptiCAD\sram.v | 50     |          |            |
| 2 | ● Source Line | C:\SynaptiCAD\sram.v | 20     |          |            |

### 3.5 Inspect Values

During a paused simulation, BugHunter supports inspecting values of variables and signals in both the *Editor* windows and in the *Inspect Values* dialog. The *Editor* window displays only the current values for the simulation. The *Inspect Values* dialog can be used to inspect both the current and past values.

To inspect with an *Editor window*.

- Put the mouse over a variable or signal name. This will cause a tool tip to pop-up and display the value.



To inspect with the *Inspect Values* dialog:

- Choose **Simulate > Inspect Values** menu option to open the dialog.
- Under the **Signal Name** column, type in the name of a variable or signal name OR select a signal in the diagram and then drag-and-drop the green bar at the top of signal name into the dialog.

- The **Event** section changes the value display to different simulation times. As you go back in time the icons will change from blue OR gates to waveforms to indicate whether if the information is coming from the simulator or from the previous results stored in the diagram window. The **Prev** and **Next** move the simulation time to the closest event in the diagram window.

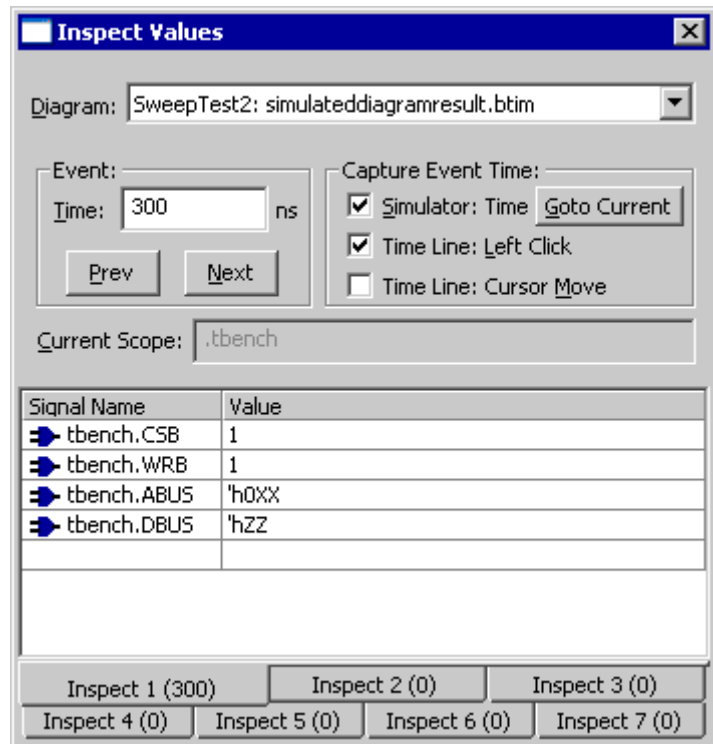
- The “S” top-level scope and “s” local scope buttons, on the simulation button bar effect the scope of the variables in the dialog.

- There are 7 different inspect tabs that let you group related variables together for easier debugging.

- When the **Simulator: Time** check box is checked each tab will continue to update its values to the current simulation time. The **Goto Current** button updates a tab to the current simulation time.

- The **Time Line: Left click** and **Time Line: Cursor Move** effect how the mouse

changes the **Time** for the dialog. Left click down in the time line on the top of the diagram window, causes a value display to appear across waveforms. If these check boxes are checked then the corresponding mouse action cause the values in the dialog to change.



### 3.6 Interactive Debugging in the Console Window

BugHunter has an interactive command console that can be used to enter simulator console commands to observe, control, and debug the simulation. The interactive command console is the drop-down combo box located on the simulation button bar. For example, during a Verilog simulation you can enter a Verilog command such as **\$finish;** (to control the simulation) or **\$display;** (to display the value of a variable). The command console is used to enter commands that are not available in a graphical environment. The types of commands that are supported are dependent on your particular simulator. BugHunter takes the commands and hands them directly to the simulator.

To use the command console window:

- Stop the simulator during a simulation run. When a simulation is stopped, the simulation display on the status bar turns bright green and reports **Simulation Stopped**. There are four ways to stop the graphical simulator during a simulation run:

- 1) Single-step into a simulation using the green single-step buttons.
- 2) Insert a source code or time breakpoint.
- 3) Press the **PAUSE** button (useful for long simulations).
- 4) In one of your source code files, embed a **\$stop** system task simulator command.


- Type a command into the console window and press the **<Enter>** key.

The next two sections describe the simulation control commands and the types of commands that can be entered into the console window.



### 3.7 Simulation Control Commands

This section describes some of the types of commands that you can enter into the console window. The specific syntax of the commands depends on the simulator. VeriLogger and VerilogXL support commands entered in the console window that perform the same functions as the simulation buttons. One of the advantages of console commands is that several commands can be entered at the same time.

For example:  would cause the simulator to execute five lines of code.

VeriLogger and VerilogXL support the following commands:

- To **continue** the simulation, type the period (.) character, or press the green **Run** button.
- To **step** to the next statement in the code, type the semicolon (;) character, or press the **Step Over** button.
- To **step-and-trace** (step to the next statement in the code and generate a trace message in the **simulation.log** file) type the comma (,) character, or press the **Step Into** button.
- To **display the current code-line** execution, (open an editor window and display the currently executing line of HDL code) type the colon (:) character, or press the **Goto** button (the magnifying glass).
- To **terminate** the simulation, type the **\$finish;** command or press the red **STOP** button.

### 3.8 Interpreted Verilog Simulators

Some interpreted Verilog simulators such as VeriLogger Pro and VerilogXL can execute lines of code entered into the console window. Generally, any behavioral statement used within an **initial** or **always** block can be entered into the console window. Statements that affect the project structurally, such as instantiating a model, are not allowed. All system tasks are accepted in the console window. Compiled code simulators can not do this because all code must be compiled before simulation begins.

Because all Verilog commands require a terminating semicolon, the semicolon must be entered in the console window. Below are some examples of useful interactive commands:

- **Displaying Variables:** Use the **\$display(...);** system task to view a variable's current value. Make sure that the scope is correct. A common mistake is to view a trace, pause the simulation, and type **\$display;** without realizing that the variable may not be in the current scope. In interactive mode, the current scope is set using the scope buttons or the **\$scope** system task. By default, the scope is set to the top-level module, not the scope at the current execution line. For example, the following statement could be used to view the variable **ireg**:

```
$scope (top.cpu1.iunit);
$display (ireg);
// OR, this can be expressed as a single statement
$display (top.cpu1.iunit.ireg);
```

All the variables in a given scope can be displayed using the **\$showvars** system task. **\$showvars** also displays the information about when the variable was last modified, specifically, the simulation time, the filename, and the line number of the reference.

- **Changing Variables:** Use an assignment statement to change a variable's value.

```
ireg = 4 * bar;
```

- **Variable Watches (breakpoints):** Interactive statements can be used to stop the simulation when a particular variable, or combination of variables, changes. For example:

```
@(top.cpu1.iunit.ireg) $stop;
```

This code will continue the simulation until the variable changes. However, this statement will not necessarily be the first statement executed after the variable changes. Due to the non-determinacy of Verilog code exe-

cution, other statements scheduled to execute at the same time unit may execute before the **\$stop** statement is performed.

- **Timed simulations:** A simulation can be set to run for a certain length of simulation time using a delay and the **\$stop** directive. The following statement suspends and waits for 1000 simulation time units to pass. After 1000 time units, the simulation is stopped.

```
#1000 $stop;
```

# Chapter 4: Editor Functions

BugHunter's editor windows are an integrated part of the simulation environment. Double-clicking in the *Project*, *Errors*, or *Breakpoints* windows will open an editor and display the relevant source code. The editor windows are also used to display the current execution line for **single-step** debugging.

All editor windows provide color-syntax highlighting, search, single-click breakpoint placement, goto lines, and font control. The simulator automatically recognizes when a file is modified in an editor window, and will warn you when it needs to be saved.

## 4.1 Opening, Saving, and Creating New Source Code

Source code files are opened and saved using the **Editor** menu options. When BugHunter starts a new simulation, it checks for any unsaved files and automatically prompts you to save them.

To open an existing source code file use one of the following methods:

- Double-click on the *filename* in the *Project* window,
- OR, use the **Editor > Open HDL file** menu option.

To create a new source code file:

- Select the **Editor > New HDL file** menu option.

To save an open source code file:


- Select the **Editor > Save HDL file** menu option to open a *Save* dialog. By default, Verilog filenames have an extension of **v**, VHDL files have **vhd** and C++ files have **cpp**.

To close the editor window and save the source code:

- Select the **Editor > Close** menu option. If the file has been altered, you will be prompted to save the file.

## 4.2 Displaying or Finding a Specific Line of Code

Most BugHunter display windows are linked directly to an editor window, making it easy to view relevant source code. Below is a list of windows and buttons that can be used to jump directly to a particular line of code.

- The *Errors* tab in the *Report* window displays compilation errors. Double-click on an error to open an editor and display the line where the error was found.
- The *Breakpoints* tab in the *Report* window displays all the breakpoints in the current project. Double-click on a breakpoint to open an editor and display the line where the breakpoint is located.
- The **Goto** button  opens an editor starting at the last line executed. This button is only active during a simulation.
- The *Project* window displays all signals, ports, and modules used in the project.

There are also several ways to search for line numbers or character strings in a file. Use the following keyboard combinations inside an active editor window to locate source code.

To move to a specific line in your code:

- Press **<Ctrl> + G** to open the *Jump To* dialog. Enter the line number to view.

To search for a character string:

- Press **<Ctrl>+F** to open the *Search* dialog. Enter the character string to locate.

- To perform another search, press the <F3> key.

### Find in Files

The Find in Files feature allows you to search through a specific directory, with or without its subdirectories, in search of a particular text string in a file. This feature can be used to search all files in the directory, or files with a specific file extension.

To use the Find in Files feature:

- Select the **Editor > Find in Files...** menu option. This will open the *Find In Files* dialog.
- Type the test phrase that you are searching for in the **Find what:** combo box. **Note:** the drop down list can be used to select a search that was previously performed. If you want to redo a search or modify a previous search you can select that search from the drop down menu.
- Type the name filter for the type of file that you want to search in the **In files/file types:** combo box. You can select from a value entered in a previous search here by using the drop-down list.
- Type the directory or folder that you want to search in the **In folder:** combo box, or select a directory used in a previous search from the drop down list. The default value for this is the current working directory.
- If you want to search subfolders (or subdirectories), make sure that the **Look in subfolders** checkbox is enabled. If you only want to search a specific directory, disable this checkbox.
- Click the **Search** button to perform the search.

The results of the search will be displayed in the *Report* window on the **Grep** tab. Double-click any reported instance to open the appropriate file and jump to that line number in the file.

## 4.3 Using the Editor/Report Preferences Dialog

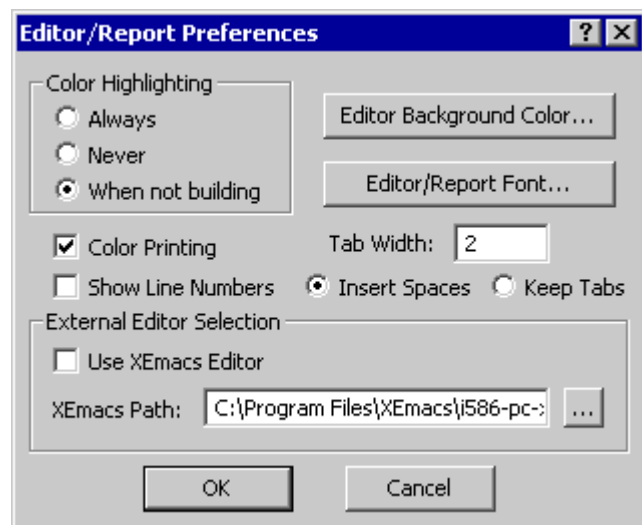
The *Editor/Report Preferences* dialog controls the editor options. This information is stored inside the project **HPJ** file.

To open the *Editor/Report Preferences* dialog:

- Select the **Editor > Editor/Report Preferences** menu option.

There are several sections in the *Editor/Report Preferences* dialog:

- The **Color Highlighting** radio buttons determine when color syntax editing is active. By default, the **When not building** option is selected so that the color syntax editing does not slow the build time of large projects.
- The **Background Color** button opens the Color dialog. From this dialog you can set the background color of the *Editor* and *Report* windows.
- The **Font** button opens the Font dialog. In the Font dialog you can set the font type, size, and color of the text in the *Editor* and *Report* windows.
- The **Color Printing** checkbox prints the source code in color. If unchecked, all code is printed in black.
- The **Show Line Numbers** checkbox determines whether or not line numbers are displayed in the editor window.



- The **Tab Width** edit box sets the number of spaces that the tab key will generate. The default setting is two spaces, but it can be set to match the tab width of an external editor.
- The **Insert Spaces** and **Keep Tabs** radio buttons determine whether spaces or tab characters are inserted when the <Tab> key is pressed.
- If the **Use XEmacs Editor** box is checked, BugHunter will use the XEmacs text editor to edit HDL files instead of its internal editor. For more information on this feature see *Section 4.5: XEmacs Integration*.
- The **XEmacs Path** edit box contains the location of the XEmacs executable to use (when XEmacs is enabled).

#### 4.4 Editor Cursor Commands

The *Report* window displays are full-featured editor windows. Listed below are the keyboard and mouse commands supported by the editor window.

| <u>Key</u>       | <u>Purpose</u>   |
|------------------|--|
| Left/right arrow | Moves the cursor one space left or right   |
| Up/down arrow    | Moves the cursor one line up or down   |
| Page Up          | Moves the cursor one page up   |
| Page Down        | Moves the cursor one page down   |
| Home             | Move to the beginning of the current line  |
| End              | Move to the end of the current line  |
| Backspace        | Deletes the character to the left of the cursor<br>OR deletes the selected text                    |
| Delete           | Deletes the character to the right of the cursor<br>OR deletes the selected text                   |
| Shift+Leftt      | Selects text one character at a time to the left   |
| Shift+Right      | Selects text one character at a time to the right  |
| Shift+Down       | Selects one line of text down  |
| Shift+Up         | Selects one line of text up  |
| Shift+End        | Selects text to the end of the line  |
| Shift+Home       | Selects text to the beginning of the line  |
| Shift+Page Down  | Selects text down one window<br>OR, cancels the selection if the next window is already selected   |
| Shift+Page Up    | Selects text up one window<br>OR, cancels the selection if the previous window is already selected |
| Ctrl+Shift+Left  | Selects text to the previous word  |

|                  |  |
|------------------|--|
| Ctrl+Shift+Right | Selects text to the next word                  |
| Ctrl+Shift+Up    | Selects text to the beginning of the paragraph |
| Ctrl+Shift+Down  | Selects text to the end of the paragraph       |
| Ctrl+Shift+End   | Selects text to the end of the document        |
| Ctrl+Shift+Home  | Selects text to the beginning of the document  |
| Ctrl+A           | Selects all of the text in the document        |
| F1               | Opens help for the editor                      |
| F4               | Print from window                              |
| Shift+F4         | Print options                                  |
| Tab              | Inserts Tab                                    |
| Ctrl+F           | Search and/or Replace Dialog                   |
| Ctrl+X           | Cut  |
| Ctrl+C           | Copy   |
| Ctrl+V           | Paste  |
| Ctrl+Z           | Undo   |
| Ctrl+Y           | Redo   |
| Ctrl+G           | Jump to line#                                  |

## 4.5 XEmacs Integration

BugHunter supports complete editing and debugging integration with the popular XEmacs text editor.

### Enabling XEmacs Integration

To enable BugHunter's XEmacs Integration feature:

- Install XEmacs onto the computer that is running BugHunter.
  - Note:** XEmacs Integration requires version 21.2 or later of XEmacs.
- Select the **Editor > Editor/Report Preferences** menu option to open the *Editor/Report Preferences* dialog.
- Check the **Use XEmacs Editor** checkbox.
- Enter the path to the XEmacs editor in the **XEmacs Path** edit box, or click the **Browse (...)** button to locate the XEmacs files.
- Click the **OK** button to enable XEmacs integration and close the *Editor/Report Preferences* dialog.

For information on XEmacs, including installation information, see the official XEmacs website at <http://www.xemacs.org/>. All the files needed to install XEmacs are available by anonymous FTP from <ftp.xemacs.org/>.

Windows users will only need to install the basic XEmacs package. Unix users will also need to install two libraries, *annotations* and *derived*, available from <ftp.xemacs.org/>. Unix users will also need to make sure that global support

for the XPM image format is installed before attempting to configure XEmacs. The most recent version of the global XPM support library can be obtained from <ftp.x.org/contrib/libraries/>. Consult your system administrator if you have any questions.

### Using XEmacs with BugHunter

The XEmacs Integration feature allows you to control project functions and simulate the project from within XEmacs. The use of Breakpoints in HDL files is also supported. For more information on running a VeriLogger simulation, see *Section 3.2: Simulation Button Bar*. For more information on Breakpoints, see *Section 3.4: Breakpoints*.

To add an HDL file created in XEmacs to the active BugHunter project:

- Select the **Syncad > Add to Project...** menu option.

To run a BugHunter simulation from within XEmacs:

- Press the <F5> key. (This is identical to selecting **Simulate > Run** from BugHunter's **Simulate** menu.)

To single-step through a BugHunter simulation from within XEmacs:

- Press the <F10> key. (This is identical to selecting **Simulate > Step Over** from BugHunter's **Simulate** menu.)

To single-step through a BugHunter simulation from within XEmacs and send a **trace** statement to the **simulation.log** file:

- Press the <F11> key. (This is identical to selecting **Simulate > Step Into** from BugHunter's **Simulate** menu.)

Note that simulation from XEmacs can also be carried out by means of the XEmacs simulation bar located at the bottom of the XEmacs editor window. These buttons function identically to the buttons on the Simulation Button Bar in BugHunter (see *Section 3.2: Simulation Button Bar*).



To add or remove a Breakpoint in XEmacs:

- Click in the margin of the XEmacs editor to the left of the line that the Breakpoint should be added to or removed from.

OR

- Right-click in the margin of the XEmacs editor to the left of the line that the Breakpoint should be added to or removed from.

- Select **Insert/Remove Breakpoint** from the context menu.

OR

- Place the cursor in the line that the breakpoint should be added to or removed from.
- Press the <F9> key.

To enable or disable a Breakpoint in XEmacs:

- Right-click in the margin of the XEmacs editor next to the Breakpoint that should be enabled or disabled.
- Select **Enable/Disable Breakpoint** from the context menu.

OR

- Place the cursor in the same line as the Breakpoint that should be enabled or disabled.
- Press the <Ctrl> and <F9> keys.

An enabled Breakpoint is represented by a red dot in the margin, and a red circle represents a disabled Breakpoint.

```
begin
Control.example1.tb_trigger <= TB_DONE;
Control.example1.tb_status <= TB_DONE;
tb_serialToparallel_output_files.init_files;
end tb_initialize_serialToparallel;
```

## 4.6 Using an External Editor

External editors can be used with BugHunter. If you use an external editor, make sure it is configured to detect when other programs externally modify a file. While simulating and debugging in BugHunter, you will want to use the internal editors to make quick fixes to the code so you can continue simulating. If your editor does not detect that you have modified a file, it may overwrite your fixes.



# Chapter 5: Waveforms and Test Bench Generation

BugHunter uses the current *Stimulus and Results* diagram to list the simulation watch signals, display simulation results, and graphically generate stimulus vectors for the simulation. The *Stimulus and Results* diagrams can be archived to separate directories and used for regression testing.

If your top-level module has input ports, BugHunter can take the drawn waveforms in the stimulus and results file and create a stimulus module and a wrapper module that hooks the stimulus up to the model under test. This makes it very fast to test individual models and small designs.

If you have purchased the Waveform Comparison Option for BugHunter, then you can perform automated comparisons between different Stimulus and Results diagrams. Also if you have purchased the Reactive Test Bench Generation option you can create test benches that check the simulation output and create pass/fail logs.

If your simulations are big enough to start slowing down because of the memory limitations of your system there are several methods for reducing the memory requirements for a simulation. The first method is to dump the waveform data to a VCD file and turn off the graphical display of waveform data during simulation. After the simulation you can load in the VCD file and view it graphically. Also if your project contains many modules the hierarchical Project window display can be turned off to save memory. And finally you can also run Verilog simulations from the command line as described in *Chapter 6: VeriLogger Pro Command Line Simulator*.

## 5.1 Stimulus and Results Diagrams

The *Stimulus and Results* diagram lists the watched signals for the simulation and displays the waveform results for these signals after simulation. This diagram is also the place where you can draw the stimulus waveforms to create unit-level test benches as described in *Section 5.2: Drawing Waveforms for Stimulus Generation*.

The *Project* window folder *Stimulus & Results* defines the current stimulus and results diagram. It is often useful to have several different diagrams for a design that define different sets of watched signals and different unit-level test benches. Each diagram can be used to test a different aspect of your design. Once you have created stimulus and results diagram that you want to keep for regression testing you can save it to a *Stimulus and Results Archive* folder.

Each time an Archive folder is made, a new subdirectory is created in the project directory. The *stimulus and results* diagram and the **simulation.log** file are copied into the archive directory. The archived files are displayed in the *Project* window under the *Stimulus & Results Archive* folder.

There are several ways to set a new *Stimulus & Results* diagram:

- In the *Project* window, right click on the *Stimulus & Results* folder and choose **Replace Current Result Diagram** from the context menu. This opens a file dialog that lets you choose a timing diagram.

OR

- Open a timing diagram and right click in the label window and choose **Set Diagram as Stimulus and Results** from the context menu.

OR

- Right click on a timing diagram name listed in *Stimulus & Results Archive* folder and choose **Set Diagram as New Stimulus and Results** from the context menu. This option will copy the archived diagram and log file back the main Project directory (the original archive files will not be changed by subsequent simulations).

To archive the current Stimulus & Results diagram:


- In the *Project* window, right click on the *Stimulus & Results* folder and select the **Save Current Result Diagram in an Archive** from the context menu. This will open the *Save the current Stimulus & Results file in an Archive* dialog.
- Enter the name of the archive in the edit box. This will become the name of the directory that the archived files are saved in.
- Click **OK** to copy the *Stimulus & Results* diagram and log file to the new directory and close the dialog. You may be asked to save the diagram and log file before they are archived.

## 5.2 Drawing Waveforms for Stimulus Generation

BugHunter can generate stimulus code for signals drawn in the *Stimulus and Results* diagram and use that code as inputs to the project. At the beginning of a compile, BugHunter will take the drawn waveforms and create a stimulus module. It will also create a top-level module that will hook up the stimulus module to your design models.

**Input Signals:** After you build the project, the input and output ports of the top-level module are automatically added to the *Stimulus & Results* diagram. You can draw the waveforms for these signals using the cursor and the waveform buttons on the diagram window.

The basic steps for creating a stimulus test bench are:

- Press the Parse MUT button  to populate the *Stimulus and Results* diagram with the module's input and output ports.
- Draw waveforms on the input signals. *Timing Diagram Editor Help - Chapter 1: Signals and Waveforms* has information on drawing waveforms. The gray signals are outputs of the MUT and will turn purple after the simulation begins. If you have the **Reactive Test Bench** option then all of the signals will come in as black so that you can draw expected response from the MUT (which will draw in blue).
- Verify that the **Simulate > Simulate Diagram With Project** menu item is checked. This option lets BugHunter create the stimulus and wrapper modules.
- Run a Simulation.
- The Auto Run /Debug Run button determines if simulations are automatically rerun each time you change the drawn waveforms or if you will be required to start a simulation with the green simulation button.

**Internal Signals:** BugHunter can generate stimulus for internal signal nodes (Verilog only). To do this, add the signal (using the full hierarchical name) to the *Stimulus & Results* diagram. The best way to add a signal is to first find the signal in the *Project* window under the <<<top-level>>> tree branch and use the context menu to set a watch on the signal (See *Section 2.4: Watching Signals and Components*).

**Inout Signals:** BugHunter can graphically generate stimulus for inout ports and simultaneously watch the port's simulated output using another signal with the same name. To do this, add two signals with the same full hierarchical name. Make one signal a watch signal and the other a drive signal. Remember to draw tri-state values on the drive signal when that signal should not be driving the inout port.

## 5.3 Working with the Diagram Window

The *Diagram* window in BugHunter is the same timing diagram editor that is the basis of SynaptiCAD's WaveFormer Pro and Timing Diagrammer Pro products. Because of this, BugHunter users have access to many of the timing diagram editor drawing features. Some features of the Timing Diagram editor such as Timing Analysis, Reactive Test Bench Generation, and GigaWave require optional feature licenses. The timing diagram editor is described in the **Help > Timing Diagram Editor Help** menu item.

There are several features that are particularly useful for viewing lengthy simulated signals. We have listed them here for your convenience.


### Viewing all signal values

- Click in the time line in the *Diagram* window. This displays a marker line that shows the value of each signal at that particular time.

### Zooming in the Drawing window


- **Click-and-drag** in the time line to zoom. Click and hold inside the time line and drag the mouse to indicate the range in which to zoom. When you release the mouse, the diagram will zoom to show the range you selected. This provides a quick way to graphically specify the zoom level and range for a section in a large timing diagram.
- The **Zoom In** and **Zoom Out** buttons, located on the right of the button bar, change the zoom level in the timing diagram.
- The **Zoom Range** button opens a dialog that lets you specify the starting and ending times displayed in the Diagram window.
- The **Zoom Full** button displays the entire timing diagram on the screen.

### Scrolling to a specific time or offset position:

- The two buttons directly above the signal label window provide an absolute time readout and a relative time readout. The **Time Button** (left), with the black writing, displays the current position of the mouse cursor in the drawing window. The **Delta Button** (right), with the blue writing, displays the difference between the mouse cursor and the delta mark (an upside-down, blue triangle) on the timeline above the drawing window. These buttons can also be used for quick scrolling in very long timing diagrams.
 
- Clicking on either button opens an edit box that accepts time values.
- Entering a value in the **Time** button causes the drawing window to scroll to that exact time.
- Entering a value in the **Delta** button causes the drawing window to scroll that amount from its current position.

## 5.4 Waveform Comparisons

Waveform comparisons graphically display the differences between compared waveforms for two timing diagrams or individual signals. This feature is exceptionally useful comparing two different simulation runs, as well as for comparing logic analyzer data to a simulation run. The specific regions where waveforms differ will turn red when the two waveforms are compared. By using the navigation buttons on the **compare** toolbar, you will be able to jump to the first difference and browse to each subsequent difference. The **tolerance** range can be set using the compare signal settings in the *Signal Properties* dialog. These settings appear in the dialog after you set the signal as a compare signal.

The **compare** toolbar contains five buttons. The **compare** button, with the yellow lightning bolt, performs the waveform comparison. The next three buttons are used to browse through the regions of difference on the signals. The last button on the toolbar, labelled **SET ALL**, is used to open the signal properties dialog with all of the compare signals selected so that matching tolerance ranges (and other signal properties) can be set. The selection is performed automatically.
 

When comparing two waveforms, the signal names must match. This can be accomplished by changing the names of the compare signals in the *Signal Properties* dialog (see below) or by ensuring that the signal names in the two files match.

### Comparing two timing diagrams

The timing diagrams that are being compared can be the result of two different simulation runs, or one or both could contain the data from a logic analyzer. To compare two timing diagrams:

- Load the first timing diagram. Either use the **File > Open Timing Diagram** menu option to load a new file or use the current timing diagram.
- Select the **File > Compare Timing Diagram** menu option to load in the signals to be compared. This opens a *File* dialog which lets you select the file to be compared. Closing this dialog loads the second set of signals and sets their signal type to compare. Any two signals that have matching names will automatically be compared. The compare signals will appear in red under the corresponding original signals.

### Comparing two signals in the same timing diagram

A signal can be changed to a compare signal to be used for comparison. This method works for both unmatched signals in files that you are comparing, or for signals that you have created in this file. To compare two signals, the signals must have the same name and one signal must have a signal type of compare.

- Double-click on the signal name to open the *Signals Properties* dialog.
- Change the name of the signal to match the signal you want to compare with.
- Select the **Compare** radio button located in the top part of the dialog. This makes the *Signals Properties* dialog display the tolerance controls used to define how the compare will be done.
- Click the **Compare** button to run a comparison.

### Finding Compare Errors

Once you have made modifications, you can rerun the compare by clicking the **Compare** button on the compare toolbar (far right toolbar; the Compare button has a lightning bolt icon). Other buttons on the compare toolbar allow you to quickly find and move to the differences that are located. The three buttons are:

- **First**, which moves to the first difference that has occurred,
- **Previous**, which moves back one, and
- **Next**, which moves forward one difference.

### Adjusting Comparison Tolerances

Ranges may also be used for comparisons. Instead of looking for comparisons directly on an edge, you can allow a tolerance within a set range of the edge. To set the tolerance on a compare signal:

- Open the *Signal Properties* dialog by double clicking left on the compare signal name. Note: the tolerance can only be set on the compare signal, not on the original signal.
- Specify the tolerance range previous to the edge by typing a value into the **-Tol:** textbox. (The value will be in the Display Time Unit.)
- Specify the tolerance range after the edge by typing a value into the **+Tol:** textbox. (This value will also be in the Display Time Unit.)
- Click the **OK** button to close the dialog. Now when you run the comparison a tolerance will be provided as specified.

The tolerance for multiple signals can be set simultaneously with the following steps:

- If the *Signal Properties* dialog is open, close it.
- Click the names of the compare signals in the signal window to select the signals to be edited. Notice that each signal can be selected individually.
- Right-click one of the highlighted signal names and select **Edit Selected Signal(s)** from the pop up menu.
- Proceed with steps outlined above for editing signals.

## 5.5 Generating and Reading VCD Files

The VCD format is a standard Verilog file format that can be used with external waveform viewers, static timing analyzers, or VeriLogger's graphical display. Watched signals in VeriLogger are displayed graphically, and by default are NOT dumped to a VCD file. Two check boxes in the *Project Simulation Properties* dialog control the output of data gathered by watched signals.

To determine the output of watched signals:

- Select the **Project > Project Simulation Properties** dialog to open a dialog of the same name.
- Check the **Capture and Show Watched Signals** checkbox to view watched signals in the *Diagram* window.
- Check the **Dump Watched Signals** checkbox to return data from watched signals to a VCD file.

It is less memory intensive to dump files than it is to actively view the data in the *Diagram* window. To speed up large simulations, turn off the waveform display and dump the watched signals. After the simulation, import the VCD file:

- Select the **Export > Import Timing Diagram From** menu option. This *Open* dialog is special in that it remembers the file type of the last file imported.
- Type the name of the VCD file you wish to open in the **File name:** edit box.
- Select the **Open** button to load the file. The waveforms are now visible in the *Diagram* window.

If you are using Verilog, VCD files can also be generated by using the Verilog system tasks **\$dumpvars**, **\$dumpfile**, **\$dumpall**, **\$dumpon**, and **\$dumpoff** to save waveform data. See the **Verilog Language Overview** for more information on the syntax of these statements.

# Chapter 6: VeriLogger Pro Command Line Simulator

This chapter describes how to launch the VeriLogger Pro command line simulator and the command line options to control the simulation. You can also enter most of the command options directly into BugHunter's *Command Console* window on the simulation button bar (see *Chapter 3.6 Interactive Debugging in the Console Window*). If you are using BugHunter as the graphical interface for another simulator, you can glance through the chapter to get an idea of what kinds of features that might be available in your simulator. The syntax for the commands depends on the simulator.

## 6.1 Preparing Verilog Source Files

Before using the command line simulator you may want to add statements to the Verilog source code to generate simulation display statements. Signals that were watched in the graphical simulator will not automatically generate output in the command line simulator.

There are several Verilog statements that will generate output:

- The **\$monitor** system task is used to continuously monitor a signal and produce an output message every time the signal changes.

```
$monitor("Counter = %d", count);
```

- The **\$display** system task is used to print text messages and look at values on signals. The **\$display** statements write the results to the **simulation.log** file. This statement is similar to a debug statement used to debug program flow in a standard programming language. See the **Verilog Language Overview** for more information on the syntax. An example of a display statement used inside a module is:

```
$display("Counter = %d", count);
```

- The **\$dumpvars**, **\$dumpfile**, **\$dumpall**, **\$dumpon**, and **\$dumpoff** system tasks are used to save waveform data in to a value change dump (VCD) file. The VCD format is a standard Verilog file format that can be used with external waveform viewers, static timing analyzers, or VeriLogger's graphical display. See the **Verilog Language Overview** for more information on the syntax of these statements.

## 6.2 Using the Command Line Simulator

To run the command line simulator:

- Open a command line window on your operating system. Windows users should open a DOS prompt.
- Navigate to the VeriLogger directory.
- Next, invoke the command line simulator with one or more source files and any desired simulation options. The following example starts the simulator, and executes the source file *model.v*:

```
vlogcmd model.v
```

If there is more than one file, then each file needs to be specified on the command line. The order that the files are entered in the command line is the order in which they are compiled. In most cases the order is irrelevant, but there are some cases where it is significant, particularly when using the same macros (**define**) across files.

```
vlogcmd cpu.v memory.v io.v
```

To avoid retyping the same source files and simulation options every time you perform a simulation, you can create a command file. A command file is a simple text file that contains a list of source files and simulation options used in the simulation. To call a command file, use the **-f** simulation option (all simulation options are listed in *Section 6.3*:

*Command Line Simulation Options*) followed by the name of the command file. The use of a command file is demonstrated below:

```
vlogcmd -f command.vc
```

A complete list and description of the commands available for the command files can be obtained by entering `vlogcmd` at the command prompt without any options.

Command files are user-created text files with a `*.vc` file extension. They consist of Verilog source files, simulator options, and other command files. When creating a command file, list only one file or simulation option per line. The following is an example of a command file with three Verilog source files and two simulation options:

```
cpu.v
memory.v
io.v
-s
-t
```

To automatically generate a command file that contains all project settings project files:

- Select the **Project > Project Simulation Properties** menu option to open the *Project Simulation Properties* dialog.
- Click the **Generate Command File** button. This takes all the project commands and file names contained in the **Command Line Options** edit box and creates a command file.

### 6.3 Command Line Simulation Options

The VeriLogger command line simulator supports several simulation options that can be used to control and debug simulations. The simulation options may be displayed in any order and anywhere on the command line. To the simulator, the following statements are identical:

```
vlogcmd -t -s cpu.v memory.v io.v
vlogcmd cpu.v memory.v io.v -s -t
vlogcmd cpu.v -t memory.v -s io.v
```

Listed below are the simulation options supported by VeriLogger:

- Use a command file, **-f <command filename>** runs the simulator with the designated command file. All of the following simulation options can be used in a command file.

```
vlogcmd -f commfile.vc
```

- Stop, **-s** compiles the source code then enters the interactive mode before the execution begins.

```
vlogcmd -s cpu.v memory.v io.v
```

- Trace, **-t** enables a tracing mode that returns a trace history of each line executed into the log file.

```
vlogcmd -t cpu.v memory.v io.v
```

- Compile only, **-c** compiles the source code and exits without performing a simulation.

```
vlogcmd -c cpu.v memory.v io.v
```

- Key filename, **-k <key filename>** changes the name of the key file that contains a log of all keystrokes entered during the simulation run. By default, the key file is called **verilog.key**.

```
vlogcmd -k mykey.key -c cpu.v memory.v io.v
```

- No Key, **-k nokey** disables the key file.

```
vlogcmd -k nokey -c cpu.v memory.v io.v
```

- **Log filename**, **-l <log filename>** changes the name of the log file that contains all output generated during a simulation. By default, the log file is called **simulation.log**.

```
vlogcmd -l mylog.log -c cpu.v memory.v io.v
```

- **No log**, **-l nolog** disables the log file.

```
vlogcmd -l nolog -c cpu.v memory.v io.v
```

- **Library filename**, **-v <filename>** specifies the name of a library file. If this option is used, VeriLogger will try to match any undefined modules to modules inside the library files.

- **Library directory**, **-y <directory>** specifies the directory path where searches for library files are made. If this option is used, the simulator will attempt to match any undefined modules with files that have one of the file extensions set with the **+libext** option. The simulator does not look inside a file unless the undefined module name exactly matches the filename. The simulator will not look at any files unless file extensions have been set using the **+libext** option. The following examples show how to specify a directory path, a directory path with spaces, and how to use the **+libext** option (UNIX users should use a backslash):

```
vlogcmd model.v -y\mylibs +libext+.v
```

```
vlogcmd model.v -y"\My Libraries" +libext+.v
```

- **Interactive Command filename**, **-i <filename.vi>** allows the simulator to accept interactive commands from a file. Any legal interactive mode command can be included in the interactive command file. The file is submitted to the simulator before the simulation begins and starts to execute as soon as the simulator enters an interactive simulation mode. Therefore the **-i** command must be paired with a statement that stops the simulator and enters the interactive mode. There are two ways to do this:

- Use the **-s** option to stop VeriLogger and enter the interactive mode before execution begins,

- OR, embed the **\$stop** system task into a Verilog source code file and use it with a delay to stop the system at a later time. For example, assume the file **cpu.v** contains the following code fragment to stop the system 1000 time units after the simulation begins:

```
#1000 $stop;
```

This file is submitted with the following command:

```
vlogcmd -i interactive.vi cpu.v memory.v io.v
```

## 6.4 Predefined Plus Options

The VeriLogger simulator supports the following Verilog run time simulation options:

- +maxdelays** | **+mindelays** | **+typdelays** determines which delay used in the **min:typ:max** expressions. In the graphical simulator this command is set using the **Project > Project Simulation Properties** menu option. In the command line simulator add the option to the command line:

```
vlogcmd cpu.v memory.v io.v +mindelays
```

- +define+<macro name>+<macros name> ...** defines macro names from the command line, generally for use with conditional compilation directives. Any number of macros can be defined by adding another **+<macro name>** to the list. For example, the *count.v* Verilog source code file had the following code fragment:

```
'ifdef EXTEND
    $display("Using extended mode");
'else
    $display("Using normal mode");
```

Then the following command will execute the first **display** statement:

```
vlogcmd count.v +define+EXTEND
```

- +synopsys** displays warnings for constructs that are either not supported or ignored by the Synopsys HDL Compiler.



**+noshow\_var\_change** disables the tracking of variable changes. By default, VeriLogger keeps track of the location and simulation time where variables are last written. This information can be displayed using the **\$show-vars** directive. This feature may cause a slight performance degradation, so it can be disabled with this option.

**+libext+<ext>+<ext> ...** specifies the filename extension used when searching for libraries in the library directory. This is most often used with the **-y** option. The following example will search the directory `\design\libs` for libraries whose filename ends with `.vl` and `.vv`:

```
vlogcmd cpu.v -y \design\libs +libext+.vl+.vv
```

**+incdir+<directory1>+<directory2>+...** specifies the directories that VeriLogger will search for included files. All the characters between the pluses are used in the directory name.

```
vlogcmd cpu.v +incdir+\design\project1+ -y \design\libs +libext+.vl+.vv
```

**+loadpli1=<pli\_library\_name.dll>:<register\_function1>, <register\_function2>, ...** specifies the PLI library name that contains a list of PLI tasks and functions to execute. VeriLogger is expecting the library to contain a `s_tfc` array called **veriusertfs** that contains a list of PLI user tasks and functions. You can also group related PLI commands into register functions so that you can partially load commands from the PLI library. The register function should contain a **veriusertfs** array and return a pointer to that **veriusertfs** array. Here are some examples of using the option:

```
vlogcmd +loadpli1=myplilib.dll
```

```
vlogcmd +loadpli1=myplilib.dll:register_my_tasks
```

```
vlogcmd +loadpli1=myplilib.dll:register_my_tasks1,register_my_tasks2
```

Here is a code example of a register function containing the **veriusertfs** array:

```
s_tfc* register_syncad_tasks()
{
static s_tfc veriusertfs[30] =
{
/** Template for an entry:
{ usertask|userfunction, data, checktf(), sizetf(), calltf(), misctf(),
"$tfname", forwref?, Vtool?, ErrMsg? },
Example:
{ usertask, 0, my_check, 0, my_func, my_misctf, "$my_task" },
***/
/** final entry must be 0 ***/
{0}
}
return veriusertfs;
}
```

## 6.5 Simulator Control Commands

In addition to the Verilog commands, there are also several VeriLogger specific commands that can be used to control the simulation:

- To *continue* the simulation, type the period (.) character.
- To *step* to the next statement in the code, type the semicolon (;) character.
- To *step-and-trace* to step to the next statement in the code and generate a trace message in the **simulation.log** file type the comma (,) character.
- To *display the current code line* execution, type the colon (:) character.

- To *terminate* the simulation, type the **Sfinish;** command, or press <Ctrl>+C.

# Chapter 7: VeriLogger Pro SDF Support

In the initial stages of your design you will be performing "functional" analysis simulations to ensure that the logic in your circuit operates correctly. After your FPGA or ASIC tools generate a layout for your gate-level design, you may want to perform a final simulation with back-annotated timing information generated during the layout process to account for real world interconnect and gate delays. This "timing" simulation is often used as a final check to ensure that unexpected delays generated during the layout process don't create timing violations in your design. The layout tools will create a Standard Delay File (SDF) that includes this timing information. By including this timing information, the model can be tested based upon these propagation delays. This chapter will tell you how VeriLogger can be used to include the information in the SDF in your model. Note: This type of timing simulation is often unnecessary if you use a static timing analysis tool to verify the critical paths in your design meet the timing constraints of your design.

## 7.1 Using a Standard Delay File (SDF)

An SDF can be produced for any module in the hierarchy of your project. For example, if you are modeling a board-level design that contains an FPGA, your FPGA tools will probably produce an SDF file for the laid out gate level module of the FPGA. To include the timing information from this file in your design, add an `$sdf_annotate` command in the FPGA module whose timing is to be modified. Include the bold lines of code in the example FPGA module shown below to tell the simulator to read the SDF timing information:

```

module MyFPGA(ports...)
  // port declarations...
  initial
    begin
      $sdf_annotate("mydesign.sdf");
    end
  // ...other code...
endmodule

```

Note that if you have an `initial` block already in the module to be annotated, you can include the `$sdf_annotate` line in the existing block. Also note that "mydesign.sdf" shown above should be replaced with the filename your tool generated for the SDF. The file extension `.sdf` should be used.

# Chapter 8: VeriLogger Technology Background

VeriLogger is an interpreted simulator. This means that when VeriLogger starts, it reads the source models, compiles them into internal data structures, and then executes them from these structures. The structures contain enough information that the source can be reproduced from the structures (minus formatting and comments.)

While the model is running, the simulation can be interrupted at any time by pressing <Ctrl>+C. This puts the simulator in an interactive command mode. From here VeriLogger commands and Verilog statements can be entered to debug, modify, or control the course of the simulation.

## 8.1 Compilation Process

VeriLogger uses a three-phase compilation process:

**Phase 1:** The files are read and converted into an internal data structure. Syntax errors and semantic errors regarding undeclared variables or illegal use of variables are reported in this phase.

**Phase 2:** In this phase the model hierarchy is built, module ports are connected, and storage for variables is allocated. If any module is instantiated more than once, its structure is copied as many times as needed. Also, module parameters are propagated. Errors reported in this phase deal with missing modules, irregularities of the parameters, and out-of-memory errors during the allocation. The project tree is built during this phase.

**Phase 3:** The entire structure is re-parsed during which time-forward references to tasks and functions are resolved, hierarchical names are resolved, and expression sizes are determined. Errors detected in this phase include semantic errors dealing with hierarchical references that could not be detected in Phase 1, illegal references to functions and tasks, port size discrepancies, and illegal expression sizes.

**Note:** Most memory is allocated in the first two phases of the compilation.

## 8.2 User-Defined Primitives and Memory Usage

User-Defined Primitives (UDPs) are used to define combinatorial primitives and two-state devices. In VeriLogger, UDPs are optimized for performance. This is accomplished by creating a table in memory for each UDP definition. Only one such table is used for each UDP definition; every instance of the definition uses the same table. When there are more than six inputs, the size of the table is very large. For this reason, the maximum number of inputs is ten (nine for state UDPs). The maximum table size is approximately 256K.

## 8.3 Notes on Using Specify Blocks

Specify blocks are used to define pin-to-pin timings and setup-and-hold checks. In VeriLogger, specify blocks function similarly to those described in the Verilog Language Reference Manual. However, there are some differences. In VeriLogger, there is no concept of expanded nets. Nets that are defined as vectors are not split into individual nets and cannot have their own timing information. Therefore, certain combinations of timing specifications will be ignored. Specifically, there are two ways to describe module paths. One is the parallel case ( $\Rightarrow$ ), and the other is the full case ( $*\Rightarrow$ ). In VeriLogger, both cases are treated as if they were defined as the parallel case. This does not pertain to scalar nets. VeriLogger supports all of the defined setup and hold systems tasks.

## 8.4 IEEE-1364 LRM Standardization

Except for the following discrepancies, VeriLogger behaves exactly as specified by the IEEE-1364 LRM and Verilog-XL standards.

### Port Collapsing

In certain versions of Verilog, if two nets are connected together via a port, the port is **collapsed** (combined to form one net). In VeriLogger, module ports are connected using transparent continuous assignments. If a register is connect-

ed to a net, then the port propagation does not occur immediately after the port changes. Instead, the port propagation is scheduled for later in the same simulation time. However, when a net is connected to a net, then a collapsed port is emulated by forcing the propagation to occur instantly. The effect of this implementation does not affect the functionality of the model being simulated, but becomes visible during trace.

### Port Connections of Different Net Types

VeriLogger does not check the legality of the connections of different net types in the hierarchy. For example, if a parent module instantiates a child module, and the net on the parent's side of a port is a **tri1** while the net on the child's side of a port is a **tri0**, an oscillation will result. To use **tri1**, **tri0**, **triand**, and **trior** as ports effectively in VeriLogger, they should be declared only in the top-most level in the hierarchy. All lower-level connections should be declared as **wire** or **tri**.

### Working Around Pullup/Pulldown Gates

When modeling an open-collector bus, a common technique is to have a **pullup** or **pulldown** gate drive a **wire** net and have drivers pull the bus in the opposite direction with a greater strength when asserting a signal. In VeriLogger, drive strengths are not implemented. Therefore, this technique will generate an unknown value (X) when a driver attempts to drive a signal in the opposite direction as the pull. The preferred method for modeling open-collector buses is to use the **triand** nets for pullup buses, and the **trior** nets for pulldown buses. This net type should only appear in the highest level of the hierarchy in which the bus exists.

### Using Trace Implementation

Trace is an indispensable tool for debugging Verilog programs. It displays each statement as it is executed, as well as any results returned from the statement. There are three ways to enable a trace:

- 1) Specify the **-t** option at the command line.
- 2) Execute the system task **\$settrace** from either the program or the interactive command line.
- 3) Execute and trace a single statement by entering a comma (,) at the interactive command line. Enter multiple commas to execute the respective number of statements.

If a model uses continuous assignments or ports, VeriLogger displays the activation of these as part of the trace as soon as the activation occurs. For example, given the continuous assignment **assign test = bar;** when **bar** changes, the continuous assignment is executed immediately and displayed in the trace. The continuous assignment represents one of possibly many drivers to the net **test**; the net itself is scheduled for updating for sometime later in the current simulation time unit.

Because port connections are implemented as continuous assignments it may take several steps for a signal to propagate from an output port to an input port, especially in cases where there are several ports connected to a net. Trace shows part of this propagation. A signal emanating from an output port travels upward to its parent module; it then travels back down to other connected ports. Each time a signal reaches a new port, the net connected to that port is evaluated and the results are displayed in the trace.

### Predefined Macro `__VERIWELL__`

The macro `__VERIWELL__` is predefined so that statements such as:

```
`ifdef __VERIWELL__
```

are used for VeriLogger-specific code, such as for waveform display.

### Simulation Statistics

The non-standard system task, **\$showstats**, displays statistics about the current simulation, including the amount of memory used and available. Some of the information is provided for diagnostic purposes only.

### Displaying the Location of the Last Value Edited

The **\$showvars** system task optionally displays the current location in the module and the simulation time at which the module variables were last changed. This information is updated even if the value did not change (that is, the new

data is the same as the old data). Tracking this update information may affect the performance of the simulation slightly. If this is a problem, this feature can be disabled with the `+noshow_var_change` command line option.

### User Interrupt

Pressing `<Ctrl>+C`, `COMMAND+C`, or `<Ctrl>+BREAK` (in MS-DOS) during simulation will put the command line version of VeriLogger into interactive mode. Pressing any of these during compilation will halt the process and exit to the operating system.

## 8.5 Implementation Differences from Verilog-XL

This section describes the differences between VeriLogger and Verilog-XL. Note that these differences are subtle and will not affect the execution of well-written Verilog models.

### Event Ordering

The order of event scheduling and execution is consistent with Verilog-XL in every extent possible. The reason for doing this is not so that models are guaranteed to work under both VeriLogger and Verilog-XL but rather because VeriLogger was designed so that users can trace models in it and in Verilog-XL with little noticeable difference. However, it should be noted that models that depend on the order of execution are considered to be unwisely written because they reflect race conditions and may perform unpredictably in other vendors' Verilog, or even in future releases of VeriLogger (or Verilog-XL). In some cases, the order of net scheduling may be different. This is because Verilog-XL schedules nets differently depending on the type of net, whether it is sourced by a continuing assignment, a net assignment, or a port, and whether a port is collapsed. In most cases, net scheduling will track that of Verilog-XL.

### Module Ports and Port Collapsing

Port connections are implemented as continuous assignments in VeriLogger. Rules for port connections are similar to those of Verilog-XL, but there are some differences. In Verilog-XL, under certain circumstances, ports are **collapsed**, that is, if each side is a net, then one of the nets disappears and only one is used. This is a performance enhancement. VeriLogger emulates port collapsing by immediately propagating values across ports that have been **collapsed**. This is unlike Verilog-XL, which actually combines nets that have been collapsed. Verilog-XL will expand vector nets into arrays of scalar nets if a port connects two different sized nets, or if one or both sides are concatenations or part selects. VeriLogger does not implement expansion of nets, so it could not handle these cases by building continuous assignments. VeriLogger will **collapse** a port if both sides of a port are scalar nets or if both sides are vector nets. Therefore, there are some cases when VeriLogger will not collapse a port, but where Verilog-XL will. This may cause a disparity in the way nets are scheduled in the two simulators.

### Control Expressions are Limited to 32 Bits

Expressions used by VeriLogger for control are limited to 32 bits. This includes repeat counts, delay values, part- and bit-select and array index expressions, and shift counts. A compile-time error will result if the expression attempts to evaluate a number greater than 32 bits.

### The \$monitor System Task

Unlike Verilog-XL, the \$monitor system task will be triggered if any variable in the argument list changes. In Verilog-XL, \$monitor changes only when an argument expression changes. For example, in Verilog-XL the following statement will not be triggered if both a and b change, and the sum stays the same. In VeriLogger, the statement will be triggered if both a and b change in this case.

```
$monitor (a + b);
```

- Verify that only one file, `add4.v`, is listed on the project tree, and that the Diagram window is empty.

# Appendix A: BugHunter System Tasks

This Appendix describes PLI based BugHunter System Tasks. If you are using a Verilog simulator, you can call these system tasks in your source code by prefixing the function with a \$ symbol, for example: \$btim\_dumpfile("my-file.btim"); . If you are running from Bug Hunter's graphical environment, you can execute these system tasks from the *console window* (see *Section 3.6: Console Window for Interactive Debugging*) on the simulation button bar. See your simulator's documentation for how to execute user-written system task using this method, a few simulators do not support this capability. If you are running your simulator in command line mode, you can link in the BugHunter dll for your specific simulator and get access to these system tasks (the graphical environment does this automatically when it launches the simulator).

**Table 1: System Task Availability by Simulator**

|                               | VeriLogger<br>vlogcmd | VerilogXL | ModelSim | NC               | ActiveHdl<br>(Verilog) |
|-------------------------------|-----------------------|-----------|----------|------------------|------------------------|
| init_syncad                   | Yes                   | Yes       | Yes      | Yes              | Yes                    |
| btim_dumpfile                 | Yes                   | Yes       | Yes      | Yes              | Yes                    |
| btim_closedumpfile            | Yes                   | Yes       | Yes      | Yes              | Yes                    |
| btim_AddDumpSignal            | Yes <sup>1</sup>      | Yes       | Yes      | Yes              | Yes                    |
| db_getcurrenttime             | Yes                   | Yes       | Yes      | Yes              | Yes                    |
| db_printinteractivescope      | Yes                   | Yes       | Yes      | Yes              | Yes                    |
| db_finish                     | Yes                   | Yes       | Yes      | Yes              | Yes                    |
| db_addtimebreak               | -                     | Yes       | Yes      | Yes              | -                      |
| db_removeimebreak             | -                     | Yes       | Yes      | Yes              | -                      |
| db_enabletimebreak            | -                     | Yes       | Yes      | Yes              | -                      |
| db_disableimebreak            | -                     | Yes       | Yes      | Yes              | -                      |
| db_getbasictype               | Yes                   | Yes       | -        | Yes <sup>2</sup> | -                      |
| db_getvalue                   | Yes                   | Yes       | Yes      | Yes              | Yes                    |
| db_printinternaltimeprecision | Yes                   | Yes       | Yes      | Yes              | -                      |
| db_setinteractivescope        | -                     | Yes       | Yes      | Yes              | -                      |

1. You must specify the full path to the signal name
2. Currently available for NC Verilog but not NC VHDL

## 1) init\_syncad

Initializes the necessary global variables, etc necessary to run the other SynaptiCAD BugHunter simulator tasks. None of the other SynaptiCAD tasks can be executed before this task is called. The BugHunter PLI automatically calls this task.

Syntax:

```
$init_syncad( )
```

**2) btim\_dumpfile**

Creates a timing diagram (btim, binary timing diagram) with the specified file name that can be used to dump signal data. Use **btim\_AddDumpSignal** to specify which signals to dump.

Syntax:

```
$btim_dumpfile( filename )
```

Arguments:

```
char* filename // filename is the name of the btim file to receive the signal data
```

Related Functions:

```
btim_AddDumpSignal, btim_closedumpfile
```

**3) btim\_closedumpfile**

If there was a dump file created by calling **btim\_dumpfile**, then this command will write the btim to disk and close it.

Syntax:

```
$btim_closedumpfile( )
```

Related Functions:

```
btim_dumpfile
```

**4) btim\_AddDumpSignal**

Adds a signal to the timing diagram that was specified by calling **btim\_dumpfile**. This will dump the specified signal to the timing diagram during simulation. Once a dump signal has been added using this command, it can not be removed.

Syntax:

```
$btim_AddDumpSignal( signalname )
```

Outputs an error if the signal doesn't exist.

Arguments:

```
char* signalname // relative or full signal name
```

Related Functions:

```
btim_dumpfile, btim_AddDumpSignal
```

**5) db\_getcurrenttime**

Outputs the current simulation time as a floating-point number and time unit.

Syntax:

```
$db_getcurrenttime()
```

Outputs a string in the following format:

```
"Current simulation time: <time> <s,ms,us,ns,ps,fs>"
```

Related Functions:

```
db_printinternaltimeprecision
```



**6) db\_printinternaltimeprecision**

Outputs the current internal time precision (resolution) of the simulator. This is the unit that **db\_addtimebreak()** expects the time argument to be in.

Syntax:

```
$db_printinternaltimeprecision()
```

Outputs a string in the following format:

```
"Internal time precision: <1,10,100> <s,ms,us,ns,ps,fs>"
```

Related Functions:

```
db_getcurrenttime
```

**7) db\_finish**

Finishes the current simulation.

Syntax:

```
$db_finish()
```

**8) db\_addtimebreak**

Adds a break point at the absolute time specified.

Syntax:

```
$db_addtimebreak( id, time, unit )
```

Outputs an error if the time specified is less than or equal to the current time.

Arguments:

```
int id;      // Breakpoint ID
time int64;  // absolute time
char* unit;  // time unit (TUnit string)
```

Related Functions:

```
db_removeimebreak, db_enableimebreak, db_disableimebreak
```

**9) db\_removeimebreak**

Removes the time break point that was added previously with **db\_addimebreak** using given id of the break point.

Syntax:

```
$btim_removeimebreak( id )
```

Outputs an error if it cannot find the time break point.

Arguments:

```
int id; //Breakpoint ID
```

Related Functions:

```
db_addimebreak, db_enableimebreak, db_disableimebreak
```

**10) db\_enabletimebreak**

Enables the time break point that was added previously with the given id.

Syntax:

```
$db_enabletimebreak( id )
```

Outputs an error if the time break point was never added.

Arguments:

```
int id; // Breakpoint ID
```

Related Functions:

```
db_addtimebreak, db_removeimebreak, db_disableimebreak
```

**11) db\_disableimebreak**

Disables the time break point that was added previously with the given id.

Syntax:

```
$db_disableimebreak( id )
```

Output an error if the time break point was never added.

Arguments:

```
int id; // Breakpoint ID
```

Related Functions:

```
db_addimebreak, db_removeimebreak, db_enableimebreak
```

**12) db\_getbasicitype**

Prints the basic type of the given object.

Syntax:

```
$db_getbasicitype( objectname )
```

Outputs an error if the object cannot be found or if the type cannot be retrieved.

Otherwise, if successful, it prints a string with the following format, where the basic type is either **variable**, **net**, **port**, **reg**, or **other**.

```
"FullObjectName : basicitype"
```

Arguments:

```
char* objectname; // path and name of the object
```

**13) db\_getvalue**

Outputs the value of a specified signal. This command will output a TState or TExState depending on the type. The signal name can be relative to the current interactive scope (retrieved by calling **printcurrentscope**), or an absolute path from the top of the hierarchy. It will first attempt to find the signal name relative to the current interactive scope. “simple” will be output in parenthesis if the state represents a TState. “exstate” will be output in parenthesis if the state represents a TExState.

Syntax:

```
$getvalue( signalname )
```

Outputs an error if the signal cannot be found or if the value cannot be retrieved.

Otherwise, if successful, it outputs a string in the following format:

```
"FullSignalName = value <simple|exstate>"
```

Arguments:

```
handle signalname; // signal name with path
```

#### 14) **db\_printinteractivescope**

Outputs the current interactive scope to the command line.

Syntax:

```
$db_printinteractivescope( )
```

Outputs a string in the following format:

```
Current scope: interactive scope
```

Related Functions:

```
db_setinteractivescope
```

#### 15) **db\_setinteractivescope**

Sets the interactive scope to the specified scope name. The scope name can be relative to the current interactive scope or a full scope path.

Syntax:

```
$db_setinteractivescope( scopename )
```

Output depends on the simulator.

Arguments:

```
const char* scopename; // scope name
```

Related Functions:

```
db_printinteractivescope
```

**Command Line Simulator Support**

BugHunter uses the following DLLs for the following simulators:

**Table 2: Simulator DLLs**

| <b>SynaptiCAD DLL Name</b> | <b>Simulator</b>        | <b>Interface Support</b>   |
|----------------------------|-------------------------|--|
| vlogpli                    | VeriLogger Command Line | PLI 1.0  |
| vxlpmi                     | VerilogXL and NCVerilog | PLI 2.0 (VPI)  |
| vpimsim                    | ModelSim (Verilog)      | PLI 2.0 (VPI)  |
| syncadfli                  | ModelSim(VHDL)          | FLI (Foreign Language Interface)   |
| syncadfmi                  | NC VHDL                 | uses the following libraries:<br>FMI (Foreign Model Import)<br>CFC (C Function Call)<br>VDA (VHDL Design Access) |

# Basic Verilog Simulation

This tutorial demonstrates the basic simulation features of VeriLogger Pro. It teaches you how to create and manage a project and how to build, simulate, and debug your design. It also demonstrates the graphical test bench generation features that are unique to VeriLogger Pro. This is a stand alone tutorial which you should be able to complete without reading any of the other tutorials. However, if you plan to make extensive use of the graphical stimulus generation features then you may also want to perform the *Basic Drawing and Timing Analysis* tutorial and *Waveform Generation and Bus* tutorial which cover the time-saving features of the timing diagram editor.

In this tutorial, you will compile and simulate a 4-bit adder and a test bench module contained in files `add4.v` and `add4test.v`. Figure 1 shows a schematic of the circuit. Later in the tutorial you will learn to graphically enter the stimulus vectors instead of using a test bench module. Also you will get to practice using the basic debugging features of breakpoints, single stepping, and viewing different signals in the file.

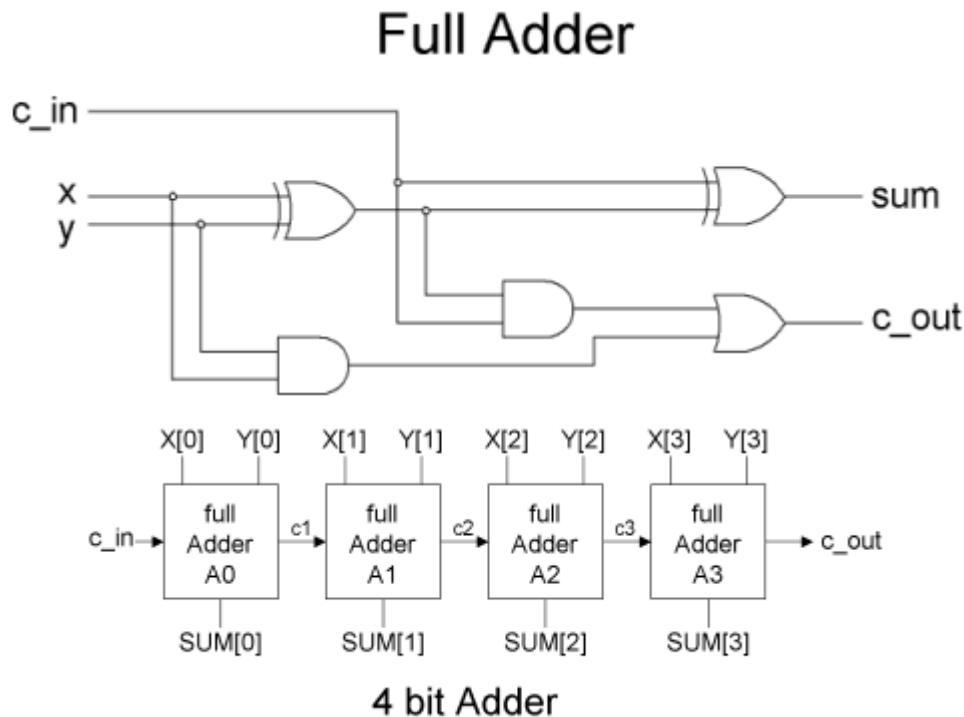


Figure 1: Schematic of the 4-bit adder simulated in this tutorial.

## Part 1: Project Management and Simulation

In this section, you will create, build, and simulate a project. VeriLogger uses a project to control all aspects of simulation and design including specifying the files to be simulated, controlling simulation options, and setting watches on signals. The project also stores the hierarchical structure of the Verilog components contained in the design and displays this information on the tree control in the Project window.

### 1.1) Add Files to the Project

VeriLogger Pro uses a project to store information about the simulation settings and the list of files to be simulated. First you will create a project and add the Verilog model files.

1. Run VeriLogger Pro and select the **Project > New Project** menu option to open the *New Project Wizard* dialog.
2. Type **add4test.hpj** into the *Project Name* edit box and press the **Finish** button to create a new project and project directory.
3. Right click the *User Source Files* folder in the *Project* window to open the context menu and choose the **Add HDL File(s) to Source File Folder** menu option. This opens the *Add Files* dialog.
4. Select the **add4.v** and **add4test.v** files located in the **SynaptiCAD\Examples\TutorialFiles\VeriloggerBasicVerilogSimulation** directory. To select multiple files at the same time, select the first file then hold down the **<CTRL>** key while using the mouse to select any additional files.
5. Press the **Open** button to add the files to the project. Both file names should be visible on the project tree. If you do not see both files then repeat instructions 3 and 4 to add the missing file to the project.


VeriLogger Pro ships with a built in editor that can be used to view and edit source code. The built in editor can be replaced with your favorite editor as described in *Section 4.6: Using an External Editor* of the BugHunter Pro and VeriLogger Pro User's Manual.

In the *Project* window:

1. Double click on the **add4.v** file to view the source code. Scan the source code and see how the modules model the schematic for the 4-bit adder. Close the editor window when you are finished.
2. Click the **Editor** menu. Notice the **Save HDL File**, **Open HDL File**, and **Editor/Report Preferences** menu options. You will probably be using these options the most.

## 1.2) Build the Tree and Use the Editor Window

In this section we will build the project tree and use the tree to view the internal modules. When files are first added to the project, you can see the file name but you cannot see a hierarchical view of the modules inside the files. To view the internal modules on the project tree you must first **build** or **run** a simulation. The **build** command compiles the Verilog files and builds the Verilog tree. It does not run a simulation. For large projects **build** lets you quickly construct the tree without having to wait for a simulation to run. To **build** a project:


1. Press the yellow **Build** button  on the simulation button bar. This will populate the **Stimulus and Results** diagram and fill out the **Simulated Module** in the *Project* window.

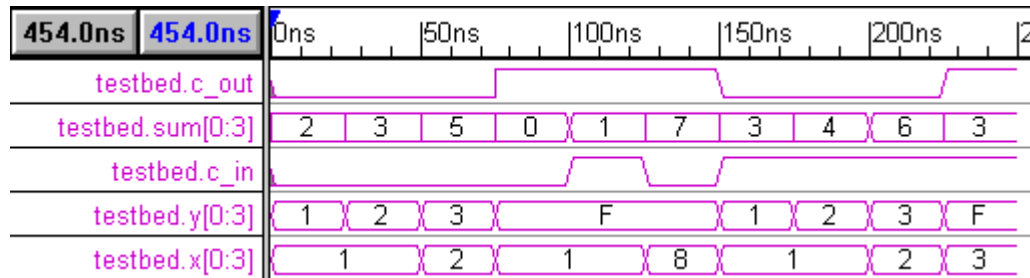
One module, **testbed**, is placed in the *Simulated Model* folder and surrounded by brackets to indicate that it is the top-level module (the highest-level instantiated component). All sub-modules can be viewed by descending the top-level module's tree. When the tree is expanded it can display the signals, ports, and components contained in each module. Expand the tree by using the + buttons:

1. Press the + button to the left of <<<**testbed**>>> to expand the project tree. Explore the sub nodes using the + buttons until you open the components folder of **A1**.
2. Double click on the **fa0** component. This will open an editor window scrolled to the instantiation of **fa0** and there is a yellow arrow to the left of the editor screen indicating the correct line. This feature lets you very quickly view component code in a large design. Close the editor when you are done.

## 1.3) Simulate the Project

When we built the project in the last section, the names of the internal signals in the top-level module were automatically added to the **Stimulus and Results** timing diagram window. This feature allows you to quickly set up a project and start simulating and debugging without having to stop and specify a set of signals. For large projects you may want to turn off this feature by choosing the **Project > Project Settings** menu and un-checking the **Grab top level signals** check box. For small projects the automatic signal watches save a lot of time so we will leave it on for the tutorial. First, let's simulate with the default signals:

1. Click the green **Run** button  on the simulation button bar. This causes a simulation to start and run until the end of the simulation time or until a breakpoint is reached. The *Diagram* window should contain purple waveforms.
2. Verify that the **sum** and **c\_out** are correctly being computed as  $x + y + c\_in$ .



## 1.4) Watch and View Internal Signals

With VeriLogger you can watch any combination of signals listed under the top-level module tree in the *Simulated Model* folder. To demonstrate this we will set watches on the **sum** outputs for the *full adders* sub-modules that make up the 4-bit adder:

1. In the *Project* window, expand the top-level module tree of the *Simulated Model* and find the **fa0** component.
2. Right click on the **sum** port for **fa0** to open a context menu and choose the **Watch Connection** menu option. This adds the **testbed.A1.fa0.sum** signal to the *Diagram* window.
3. Press the green **Run** button to run another simulation. Verify that the **testbed.A1.fa0.sum** signal is the 0 bit of the **testbed.sum[3:0]** signal.

Signals can be removed from the watch list by selecting the signal name in the *Stimulus and Results* diagram and pressing the delete key.

Next we will experiment with different ways to view waveforms in the *Diagram* window:

1. In the time line above the signals in the *Diagram* window, left click down and hold to show a marker that displays the value of each signal. Release the mouse button without dragging.
2. Left click and drag the marker about 50ns in the time line window. When you release the mouse button, the window will zoom to display the time range that the mouse was dragged over.
3. Right click in the time line to zoom out on the waveforms.
4. Press the **Zoom Full** button on the *Diagram* window to return the zoom level to the entire simulation range.

## 1.5) Save the Project, Waveforms and Source Code

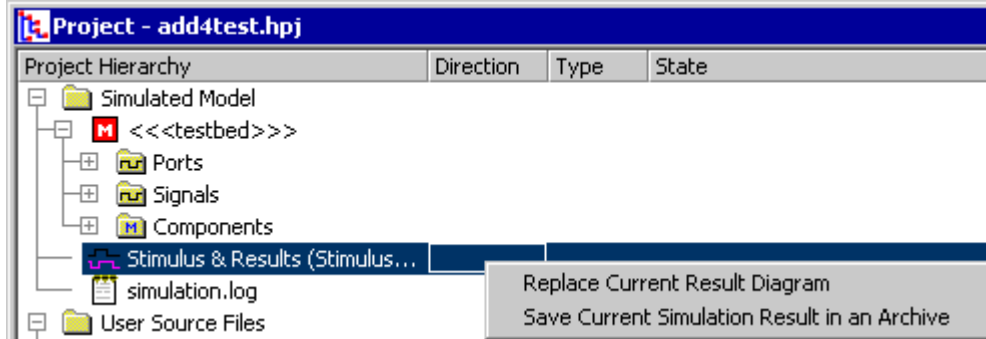
Next we will learn to save the project, waveforms, and source code. The project saves the simulation options and the names of the files contained on the project tree. It does not save the source code or the watched signals. To save the project:

1. Choose the **Project > Save Project** menu option.

The watch signals and simulation results are saved in the correct stimulus and results file. By making the watched signals separate from the project file, VeriLogger lets you set up different sets of watched signals so that you do not have to watch your entire design each time you simulate. Also watching small sections of your design makes it easier to

detect bugs in a particular section and speeds up simulation execution. In the evaluation version of VeriLogger you cannot save the waveforms, however in the full version you can save using the following menu command:

1. Select the **File > Save Timing Diagram** menu option to save the active timing diagram window.
2. In the *Project* window, right click **Stimulus & Results** to open the context menu. These functions allow you to change the current Stimulus and Results diagram.



Each time you simulate, every open editor is queried to determine if the source code needs to be saved before the simulation starts. If you need to save the code before you are ready to perform a simulation, use one of the following menu options:

1. The **Editor > Save HDL File** menu option to save the source code in the editor with the focus.
2. The **Editor > Save All** menu option to save the source code in all opened editors.

To re-open a VeriLogger Project, first open the project and then load the timing diagram files.

## Part 2: Graphical Test Bench Generation

In this section you will draw and simulate a test bench using the timing diagram editor.


### 2.1) Remove TestBench Model and Clean Results Diagram

Now we will set up the project for this section by removing the test bench file and saving the project under a different name.

1. Select the **Project > Save Project As** menu option and save the project under the name of **add4wave.hpj**.
2. In the *Project* window *User Source File Folder*, right click **add4test.v** and select the **Remove Selected File from the Folder** from the context menu.
3. Delete all of the signals in the Stimulus and Results diagram by selecting the signal names and clicking the delete key.
4. Verify that only one file, **add4.v**, is listed on the project tree, and that the *Diagram* window is empty.

### 2.2) Build the Project and Examine the Black Signals

In the previous section, all the signals were purple to indicate that they were simulated signals that were generated by the Verilog code. In this section we have deleted the testbed module and the new top level module has input port signals that are not being driven by any other module in the project. To verify this:

1. Verify that the **Simulate > Simulate Diagram With Project** menu option is checked. This option lets the simulator compile both the drawn waveforms and the Verilog source code files together.
2. Press the **Extract the MUT Ports into Diagram** button  on the simulation button bar.



3. Notice that the *Diagram* window now has two purple signals and three black signals. The purple signals are "simulated" signals whose values will be determined during the next simulation (once they are simulated they will turn purple). The black signals are input signals that need to be defined before a non-trivial simulation can take place.
4. Use the Project tree to verify that the black signals are input ports of the <FourBitAdder> module.

### 2.3) Use the Debug Run and Simulation Mode

VeriLogger has two simulation modes: **Auto Run** and **Debug Run**. The simulation mode is displayed on the left most button on the simulation button bar. In the **Debug Run** mode, simulations are started only when the user presses the **Run** or **Single Step** buttons (similar to a standard Verilog simulator). In **Auto Run** mode the simulator will automatically run a simulation each time a waveform is edited in the Waveform window. This mode makes it easy to quickly test small modules and perform bottom-up testing. While drawing the original test bench we will set the simulator to **Debug Run** mode:

1. Press the simulation mode button to toggle the display to **Debug Run**.



### 2.4) How to Draw Waveforms

If you are already familiar with SynaptiCAD's timing diagram editing environment, skip ahead to Section 2.6 where you will draw stimulus vectors and use the *Virtual State* edit box to define the values for the x and y busses.

If this is your first time using a SynaptiCAD timing diagram editor then we will first draw several random waveforms to familiarize you with the drawing environment.

1. Notice the buttons with the waveforms drawn on them. These are the state buttons. The active button is colored red and indicates the state of the next segment drawn. In this case, the **HIGH** state button is probably active.



2. Move the mouse cursor to inside the drawing window at the same level as the signal name **c\_in**, and at about 40ns.
3. Left click to draw a waveform segment from 0ns to the cursor. Notice that a **HIGH** signal was created.
4. A different state button is now activated. The state buttons automatically toggle between the two most recently activated states. The small red **T** above the state name denotes the toggle state.
5. Move the cursor to about 80ns on the same signal and left click. Now a **LOW** segment is drawn from the end of the **HIGH** signal to the location of the cursor.
6. Left click on the **VAL** button to activate the valid state button and draw another waveform segment.
7. Draw more segments, using all the states except the HEX button. We will use this button later to define the state values for the multi-bit signals. For now, experiment with the graphical states on each of the black signals (the purple signals are outputs of the simulation and cannot be drawn on).

Your drawing should be a mess, or at least look nothing like Figure 2 located in Section 2.6.

### 2.5) How to Edit Waveforms

There are four main editing techniques used to modify existing signals (Note: these techniques will not work on clocks and simulated signals). The most commonly used technique is the dragging of signal transitions to adjust their location. The other three techniques all act on signal segments (the waveforms between two consecutive signal transitions). The segment waveform can be changed, deleted, or a new segment can be inserted within another segment. Use each of the following techniques:

1. **Move a signal transition:** Left click and hold on a signal transition. A green bar will appear that follows the mouse cursor. Release the mouse button when the green bar is at the desired location.
2. **Change the state of a segment:** A segment is the waveform between two consecutive signal transitions. Left click on the segment to select it (a selected segment has a highlighted box drawn around it). Then left click on the state button of the new state desired.
3. **Delete a segment:** Select a segment, then press the **<delete>** key.
4. **Insert a segment:** Inside a large segment, left click down and drag to the right, then release. A new segment will be added in the middle of the original segment. For this operation to work, the original segment must be wide enough to be selected.

More waveform generation techniques are covered in the *Timing Diagram Editor - Chapter 1: Signals and Waveforms* on-line help.

## 2.6) Draw the Stimulus Waveforms

Now use the above techniques to edit the signals so they have roughly the same transitions and graphical states as the signals in the figure below. This is not the normal way to create a timing diagram, but it will teach you how to use the editing features of SynaptiCAD's timing diagram editor. Make sure you try all the editing techniques.



**Figure 2: Stimulus vectors for the 4-bit adder circuit**


Next, edit the virtual bus states of the valid segments on the x and y buses:

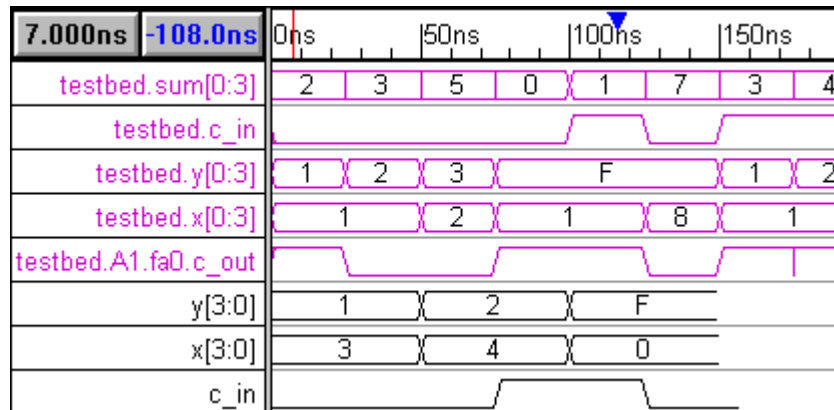
1. Double click on the first segment on the **x** signal to open the *Edit Bus State* dialog.
2. Verify that the default radix is **hex**.
3. Enter **1** into the *Virtual* edit box.
4. Press the **ALT-N** keys or **Next** button to move to the next segment on the signal.
5. Continue to enter values into each segment so that it matches Figure 2 and press the **OK** button to accept the last value.
6. Repeat the above instructions for the **y** signal.

At this point, the **c\_in**, **x[3:0]**, and **y[3:0]** signals should look like Figure 2. Exact placement of edges is not required for this tutorial.

## 2.7) Simulate Using the Auto Run Simulation Mode

Currently the simulator is in **Debug Run** mode, so simulations are started only when the Run button is pressed. Start a simulation:

1. Press the yellow **Compile Model and Testbench** button  on the simulation button bar. This will generate the test bench and compile the MUT and test bench together.
2. Press the green **Run** button on the simulation button bar.
3. Verify that the **sum** and **c\_out** are correctly being computed as  $x + y + c\_in$ .



4. Next, drag-and-drop an edge on the **x[3:0]** signal. Notice that the simulated signals do NOT change values because the simulator is in **Debug Run** mode.
5. Press the green **Run** button to update the simulation values.

Next we will demonstrate the **Auto Run** mode which allows interactive debugging of modules. This mode is especially useful for debugging small modules.

1. Press the simulation mode button to toggle the display to **Auto Run**.
2. Drag-and-drop an edge on the **x[3:0]** signal. Notice that the simulated signals change values as soon as you drop the edge.
3. Experiment with dragging edges and changing the values of the virtual states. If this was a low-level module that you just designed, you could quickly check the functionality of the module without having to design a formal test bench.

## 2.8) Import and Generate Waveforms

The most difficult and tedious part of designing test benches is accurately entering the waveform data. VeriLogger accelerates this process by accepting waveform data via four different methods: Verilog code, drawing, simulator output, and equation generation. So far we have demonstrated the drawing of waveforms and the use of standard Verilog code which are excellent choices for designing small test benches. However, for large test benches it is easier to use automated techniques to generate your data. The equation-based generation of waveforms is covered in *Chapter 11: Waveform Equation Generation*. If you purchase the “Waveform Import” option then you can also import waveform data from Agilent and Tektronix logic analyzers, spreadsheets, and SPICE simulators.

Let us quickly demonstrate the waveform equation features using the following steps:

1. Double click on the **x[3:0]** signal name to open the *Signal Properties* dialog box.
2. Notice the drop-down edit box to the right of the **Wfm Eqn** button. This box is where temporal equations are entered. The default equation contains the syntax for all the possible states. If you start by editing this equation you will not have to look up the syntax for writing the temporal equation.

```
8ns=Z (5=1 5=0)*5 9=H 9=L 5=V 5=X
```

3. Press the **Wfm Eqn** button to apply the above equation to the signal.

4. Look at the generated waveform and compare it to the equation. Notice that the equation is a list of the form *time\_duration=state\_of\_segment* elements. To repeat parts of the list use the syntax *(list)\*repeat\_number*.

You can also automatically label waveforms by using the **Label Waveform Equation** functions. These are more complex than the waveform equations so you will have to read *Chapter 11* in the TestBencher and WaveFormer manual to get the full benefit of these features.

1. Double click on the **x[3:0]** signal name to open the *Signal Properties* dialog box.
2. Notice the drop-down edit box to the right of the **Label Eqn** button. This box is where label waveform equations are entered.
3. Enter the following equation into the drop-down edit box: **Hex( Inc ( 0 , 1 , 16 ) )**
4. Press the **Label Eqn** button to label the signal for **x[3:0]**. This equation generates increments from 0 in steps of 1 for 16 times and outputs a hexadecimal value.
5. Notice how the labels have changed on the signal (you may need to *Zoom In* to clearly see all the segments). Also notice how the simulation output changed for the valid segments but it did not change for the non-valid segments. This is because the **virtual state** values are only used to define the state of the valid segments.

### 3) Breakpoints, Stepping and Tracing

If you would like to practice debugging, first read the *Getting Started* and *Chapter 3: Simulation and Debug Functions* chapters in the on-line VeriLogger Help. Next, introduce a syntax error into the **add4.v** file and attempt to find it using the Errors tag in the Report window. Fix the syntax error. Then introduce a semantic error in the full adder code so that it does not handle the carry correctly. Use breakpoints and single-step debugging to locate the error.

# Index

\$display 25, 38  
 \$dumpall 37, 38  
 \$dumpfile 37, 38  
 \$dumpoff 37, 38  
 \$dumpon 37, 38  
 \$dumpvars 37, 38  
 \$finish 25  
 \$monitor 38, 46  
 \$scope 25  
 \$showstats 45  
 \$showvars 25, 45  
 \$stop 25

## A

### Add

- breakpoint 22
- files to a project 9, 13
- signal to be watched 15

Auto Run simulation mode 21  
 Automatic Watches 15

## B

Back-annotated simulation 43  
 Breakpoints
 

- adding 22

 btim\_AddDumpSignal 47, 48  
 btim\_closedumpfile 47, 48  
 btim\_dumpfile 47, 48  
 Build button 21  
 Building a project 9

## C

Changing variables 25  
 Color Highlighting 28  
 Command file 38  
 Command line simulator
 

- creating a command file 39
- options 39
- predefined options 40
- preparing the source files 38
- running 38

- simulator control commands 41

Comparing waveforms 35  
 Console window 22
 

- entering commands 25
- interactive debugging in 24
- using 24

 Continuing a simulation 25

## D

db\_addtimebreak 47, 49  
 db\_disabletimebreak 47, 50  
 db\_enabletimebreak 47, 50  
 db\_finish 47, 49  
 db\_getbasicitype 47, 50  
 db\_getcurrenttime 47, 48  
 db\_getvalue 47, 50  
 db\_printinteractivescope 47, 51  
 db\_printinternaltimeprecision 47, 49  
 db\_removevetimebreak 47, 49  
 db\_setinteractivescope 47, 51  
 Debug Run simulation mode 21  
 Debugging
 

- using the Console window 24

 define (simulation option) 40  
 Diagram window usage 34  
 Differences from Verilog-XL 46  
 Displaying
 

- breakpoints 27
- errors 27
- the current code-line 25
- variables 25

## E

Editor keyboard commands 29  
 Editor Preferences 28  
 Editor Preferences dialog 29  
 Editors - external 32  
 Errors tab in the Report window 22  
 Expanding/hiding tree branches 14

## F

Finding a line of code 27  
 Font 28

**G**

Goto button 21  
 Grouping waveforms  
   Stimulus & Results Diagram  
   using 33

**I**

incdir (simulation option) 41  
 init\_syncad 47  
 Interactive Verilog commands 25

**J**

Jump To dialog 27

**K**

Keyboard shortcuts in editor windows 29

**L**

libext (simulation option) 41  
 Line Numbers 28  
 Local Scope button 22

**M**

maxdelays (simulation option) 40  
 Memory Usage 44  
 mindelays (simulation option) 40

**N**

noshow\_var\_change (simulation option) 41

**O**

Open  
   existing source code 27  
   grouped signals 33  
   project 13  
   Verilog file 14

**P**

Port collapsing 44  
 Port connections of different net types 45  
 Project  
   adding files to 9, 13  
   building 9  
   opening 13  
   saving 13

simulating 10

  starting a new project 13  
   using the tree control 14

Project window 14

Pullup/Pulldown gates 45

**R**

Report Preferences 28

Report Window 22

Run/Resume button 21

**S**

Save

  existing source code 27

  grouped signals 33

  project 13

Scrolling to a time or offset position 35

SDF 43

Searching for a character string 27

Simulating the project 10

Simulation

  back-annotated 43

  command controls 25

  continuing 25

  displaying the current code-line 25

  stopping 24

Simulation button bar 21

Simulation modes 21

Specify blocks 44

Standard Delay File

  see SDF

Step and trace 25

Step Into and Trace Calls button 21

Step Over Calls button 21

Stimulus & Results Diagram

  changing 14

Stop button 21

Stopping a simulation run 24

syncadfmi 52

synopsys (simulation option) 40

Syntax Highlighting 28

System Tasks 47

**T**

Tab Width 28  
Top Scope button 21  
Trace implementation 45  
Tree control

- expanding/hiding branches 14

typdelays (simulation option) 40

**U**

User interrupt 46  
User-Defined Primitives 44

**V**

Variables

- changing 25
- displaying 25

VCD files

- saving waveform data 38

Verilog statements that generate output 38  
verilog.log file 22  
VeriLogger compilation process 44  
VERIWELL - predefined macro 45  
View

- signal declarations 14
- source code 14

Viewing all signal values 35  
vlogpli 52  
vpimsim 52  
vxlpmi 52

**W**

Watching a signal

- determining output 37
- in a specific instance of a module 15
- selecting the signal 10, 15
- stop watching a signal 15

Waveform comparison 35

- adjusting tolerances 36
- comparing diagrams 35
- comparing signals 35

**X**

XEmacs 30

- enabling 28

**Z**

Zooming techniques 35