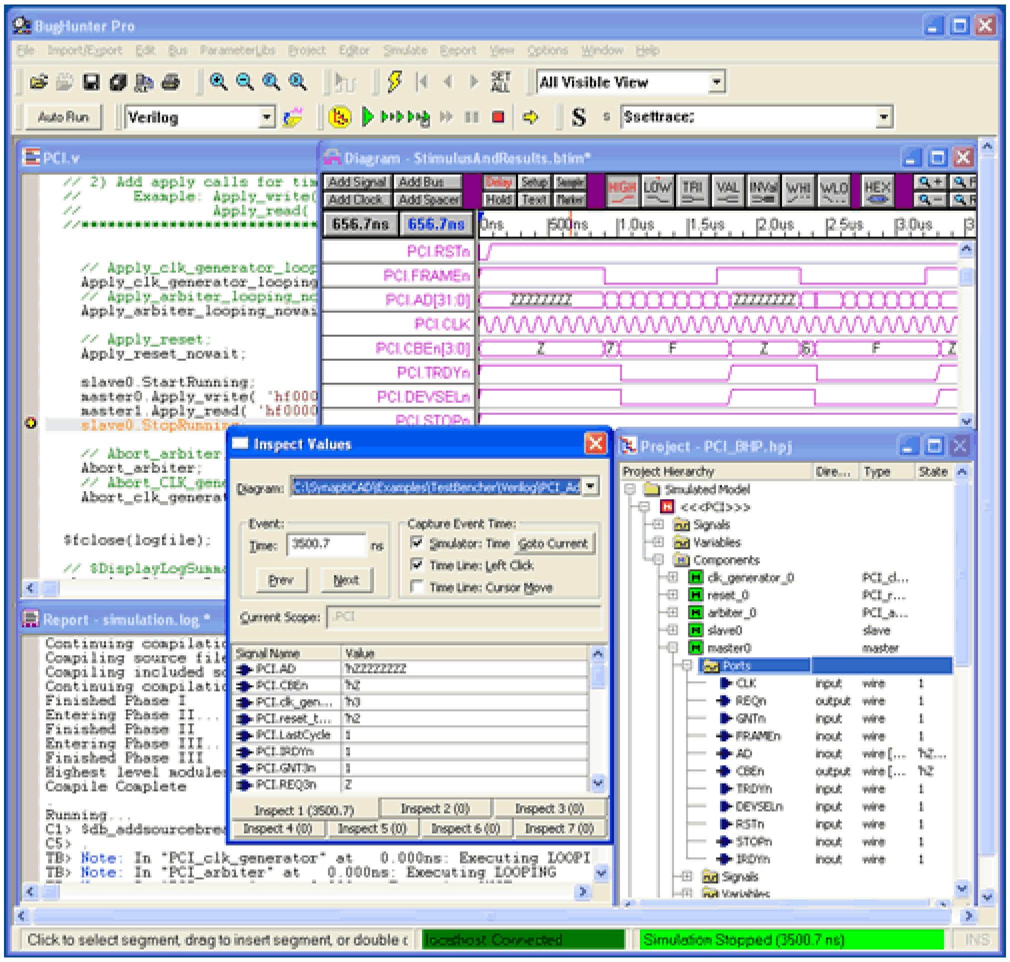
BugHunter Pro and the VeriLogger Simulators


BugHunter Pro
File Import/Export Edit Bus ParameterLibs Project Editor Simulate Report View Options Window Help
All Visible View
Auto Run
Verilog
$settrace;
PCI.v
Diagram - StimulusAndResults.btim*
Add Signal Add Bus Add Clock Add Spacer
Delay Setup Sample Hold Text Marker
HIGH LOW TRI VAL INVal WHI WLO HEX
656.7ns 656.7ns
0ns 500ns 1.0us 1.5us 2.0us 2.5us 3.0us
PCI.RSTn
PCI.FRAMEn
PCI.AD[31:0]
PCI.CLK
PCI.CBEn[3:0]
PCI.TRDYn
PCI.DEVSELn
PCI.STOPn
Inspect Values
Project - PCI_BHP.hpj
Report - simulation.log *

```
// 2) Add apply calls for ti
//      Example: Apply_write(
//               Apply_read(
//*****************************

  // Apply_clk_generator_loop
  Apply_clk_generator_looping
  // Apply_arbiter_looping_nc
  Apply_arbiter_looping_nowai

  // Apply_reset;
  Apply_reset_nowait;

  slave0.StartRunning;
  master0.Apply_write( 'hf000
  master1.Apply_read( 'hf0000
  slave0.StopRunnin

  // Abort_arbiter
  Abort_arbiter;
  // Abort_CLK_gene
  Abort_clk_genera

$fclose(logfile);

// $DisplayLogSumm
```

Diagram: C:\SynaptiCAD\Examples\TestBencher\Verilog\PCI_Ad

Event: Time: 3500.7 ns

Capture Event Time: [✓] Simulator: Time [✓] Time Line: Left Click [ ] Time Line: Cursor Move

Current Scope: .PCI

Signal Name	Value
PCI.AD	'hZZZZZZZZ
PCI.CBEn	'hZ
PCI.clk_gen...	'h3
PCI.reset_t...	'h2
PCI.LastCycle	1
PCI.IRDYn	1
PCI.GNT3n	1
PCI.REQ3n	Z

Inspect 1 (3500.7) Inspect 2 (0) Inspect 3 (0) Inspect 4 (0) Inspect 5 (0) Inspect 6 (0) Inspect 7 (0)

Project Hierarchy	Dire...	Type	State
Simulated Model			
<<PCI>>>			
Signals			
Variables			
Components			
clk_generator_0		PCI_cl...	
reset_0		PCI_r...	
arbiter_0		PCI_a...	
slave0		slave	
master0		master	
Ports			
CLK	input	wire	1
REQn	output	wire	1
GNTn	input	wire	1
FRAMEn	inout	wire	1
AD	inout	wire [...	'hZ...
CBEn	output	wire [...	'hZ
TRDYn	input	wire	1
DEVSELn	input	wire	1
RSTn	input	wire	1
STOPn	inout	wire	1
IRDYn	inout	wire	1
Signals			
Variables			

```
Continuing compilatio
Compiling source file
Compiling included so
Continuing compilatio
Finished Phase I
Entering Phase II...
Finished Phase II
Entering Phase III...
Finished Phase III
Highest level module
Compile Complete

Running...
C1> $db_addsourcebre
C5> .
TB> Note: In "PCI_clk_generator" at    0.000ns: Executing LOOPI
TB> Note: In "PCI_arbiter" at   0.000ns: Executing LOOPING
```

Click to select segment, drag to insert segment, or double c | localhost Connected | Simulation Stopped (3500.7 ns) | INS

BugHunter Pro and the VeriLogger Simulators

Printed: December 2007 in (whereever you are located)

BugHunter Pro and Verilog Simulators

BugHunter Pro, VeriLogger Extreme, VeriLogger Pro

Welcome to the BugHunter Pro, VeriLogger Extreme, and VeriLogger Pro manual. This manual covers using the BugHunter Pro graphical simulation and debugging interface. This is the interface for both VeriLogger Extreme and VeriLogger Pro and can also be used with most commercial simulators.

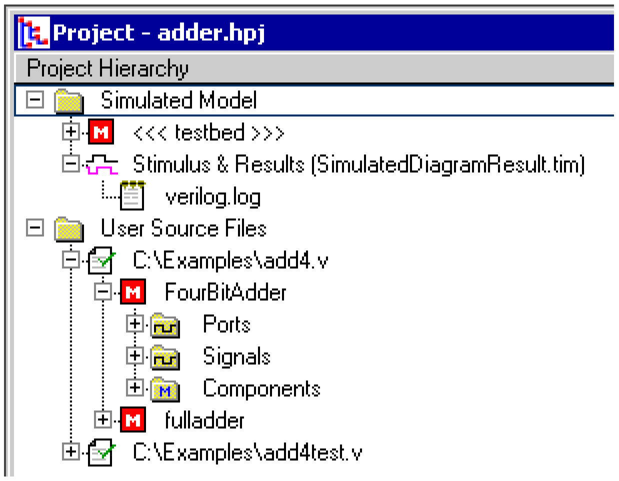
BugHunter uses the SynaptiCAD graphical environment and supports all major HDL simulators. It has the ability to launch the simulator, provide single step debugging, unit-level test bench generation, streaming of waveform data, project management, and a hierarchy tree. The unit-level test bench generation is unique in that it lets the user draw stimulus waveforms and then generates the stimulus model and wrapper code and launches the code. It is one of the fastest ways to test a model and make sure that everything is working correctly. The debugger also has exceptional support for VCD waveform files.

With an integrated debugging environment you can graphically build a project, launch a simulation, and view the results in just a few minutes. The interface also manages the test bench interface so that it is easy to create a set of regression tests to run the design through

Table of Contents

Chapter 1: A Quick Start to VeriLogger & BugHunter

This chapter covers the basic steps involved in setting up BugHunter to work with your simulator and how to create and debug a project. Each step is listed in the suggested order that you will want to perform the functions.



Step 1: Setup the Simulator Path

Step 2: Setup the Simulator Options

Step 3: Create a Project

Step 4: Add Source Files to the Project

Step 5: Draw a Test Bench (optional)

Step 6: Build the Project and Set the top

Step 7: Simulate and Debug

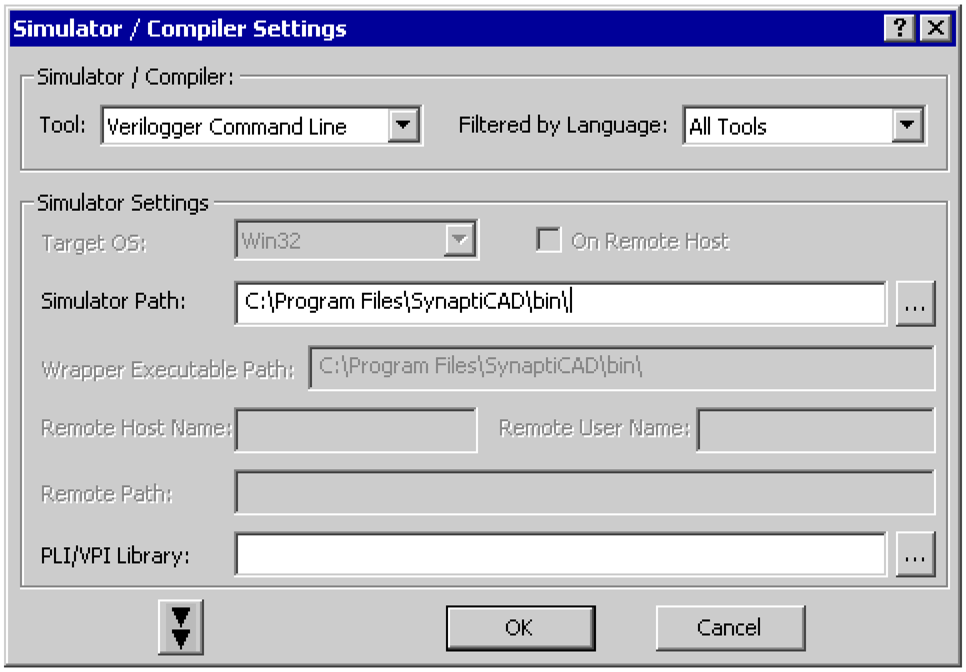
Step 8: Save the Project, Code and Waveform Files

Step 1: Setup the Simulator Path

BugHunter Pro needs to know where your VHDL/Verilog simulator or C++ compiler is located. If you are using VeriLogger Extreme or VeriLogger Pro you can skip this section because the simulator was setup during installation. BugHunter saves the paths for each external simulator or compiler in the syncad.ini file each time the program is closed.

Set the Path to the Simulator:

- 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.



Tool: Verilogger Command Line
Filtered by Language: All Tools
Target OS: Win32
On Remote Host
Simulator Path: C:\Program Files\SynaptiCAD\bin\
Wrapper Executable Path: C:\Program Files\SynaptiCAD\bin\
Remote Host Name:
Remote User Name:
Remote Path:
PLI/VPI Library:

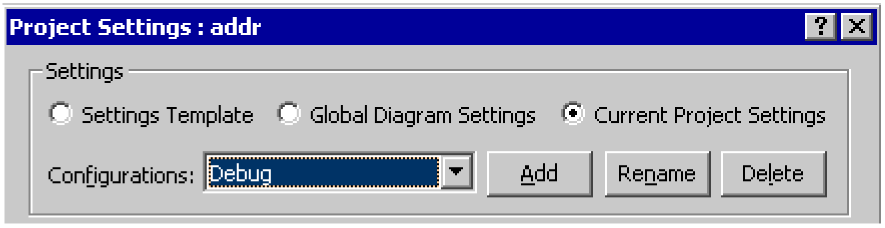
- Continue to setup the paths for each tool that you are interested in using. When you are done click **OK** button to close the dialog.

Step 2: Setup the Simulator Options

Generally the default settings for each simulator will be sufficient to properly simulate a project, so you can skip this section. However, if you you are moving projects back and forth between different machines and simulators you may wish to create **configuration** templates for each machine. Also you may wish to have different settings for different debug setups. The *Project Simulation Properties* dialog determines the simulator run time options and which simulator to use for projects and diagrams.

Open the Project Simulation Dialog:

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



( ) Settings Template ( ) Global Diagram Settings [✓ Current Project Settings]
Configurations: Debug

Global versus Project Settings

- Select **Settings Template** to edit 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 is used to reset the INI file to the factory default settings for this dialog.
- Select **Global Diagram Settings** to edit the options for how transactions are simulated (simulated signals in a *Diagram* window). These are stored in the INI file.
- Select **Current Project Settings** to edit the project settings for the current project. These settings are stored in the Project HPJ file when you save the project.

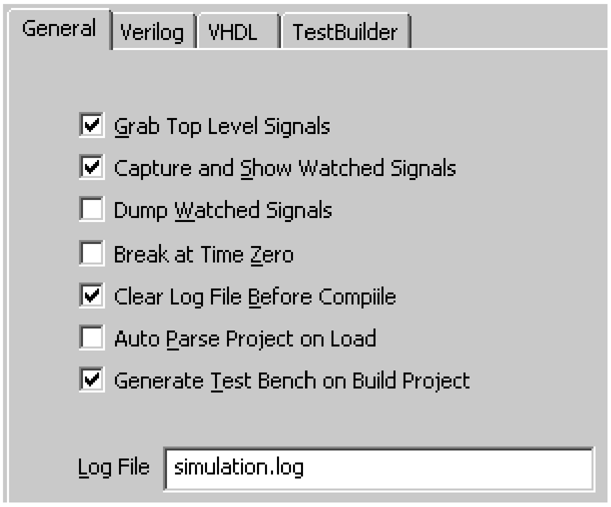
Configurations:

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. The ***Debug* Configuration** holds the default 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:

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



- [✓] Grab Top Level Signals
- [✓] Capture and Show Watched Signals
- [ ] Dump Watched Signals
- [ ] Break at Time Zero
- [✓] Clear Log File Before Compiile
- [ ] Auto Parse Project on Load
- [✓] Generate Test Bench on Build Project

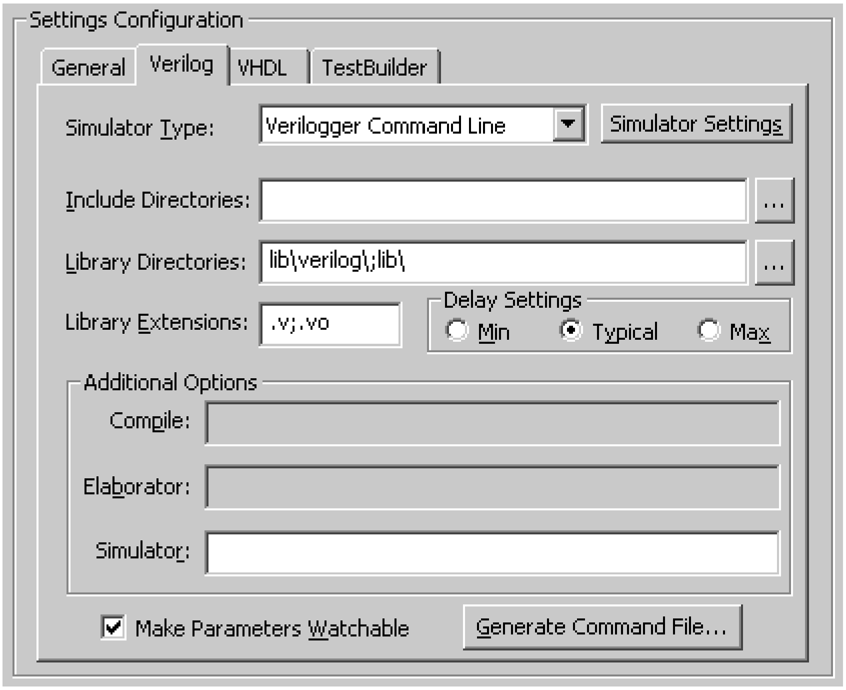
Log File: simulation.log

- **Grab Top Level Signals** causes signals in the top-level component to be automatically added as Watch signals in the stimulus and results diagram whenever the project is rebuilt.
- **Capture and Show Watched Signals** enables the display of waveform results from a simulation run.
- **Dump Watched Signals** generates a dump file for any watched signals in the diagram. The generated file will named ***diagramName* .VCD**.
- **Break at Time Zero** 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.
- **Clear Log File Before Compile** 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 component is the first component 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.
- **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.
- **Log File** specifies the name of the log file that receives all the simulation results and information. By default BugHunter uses *simulation.log*.

Verilog Tab:

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

- **Simulator Type** specifies the simulator
- The **Simulator Settings** button opens the *Simulator / Compiler Settings* dialog where you can edit the simulator paths.
- **Include Directories** specifies the directories where BugHunter searches for included files. The following is a Windows example (Unix users should use the / slashes):



Simulator Type: Verilogger Command Line

Include Directories:

Library Directories: lib\verilog\;lib\

Library Extensions: .v;.vo

Delay Settings: Min [✓ Typical] Max

Compile:

Elaborator:

Simulator:

[✓] Make Parameters Watchable

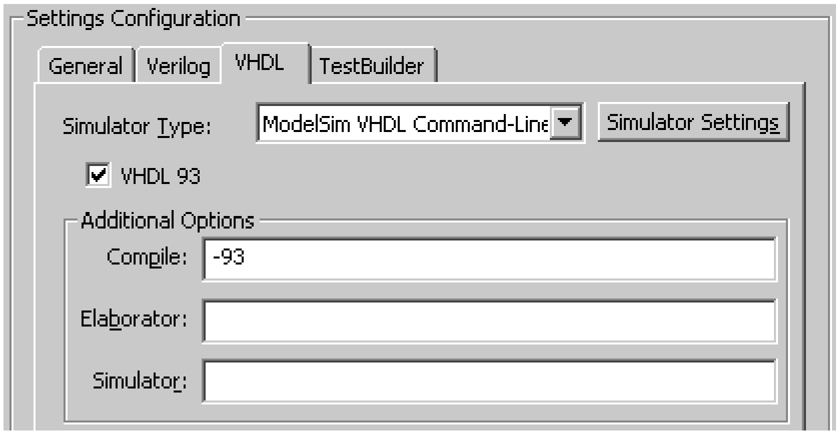
```
C:\design\project;c:\design\library
```

- The **Library Directories** box lists the path and directories where the program searches for library files. BugHunter will try to match any undefined components 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 component 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** 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 component.

VHDL Tab:

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

- **Simulator Type** determines the simulator.



Simulator Type: ModelSim VHDL Command-Line

☑ VHDL 93

Compile: -93

Elaborator:

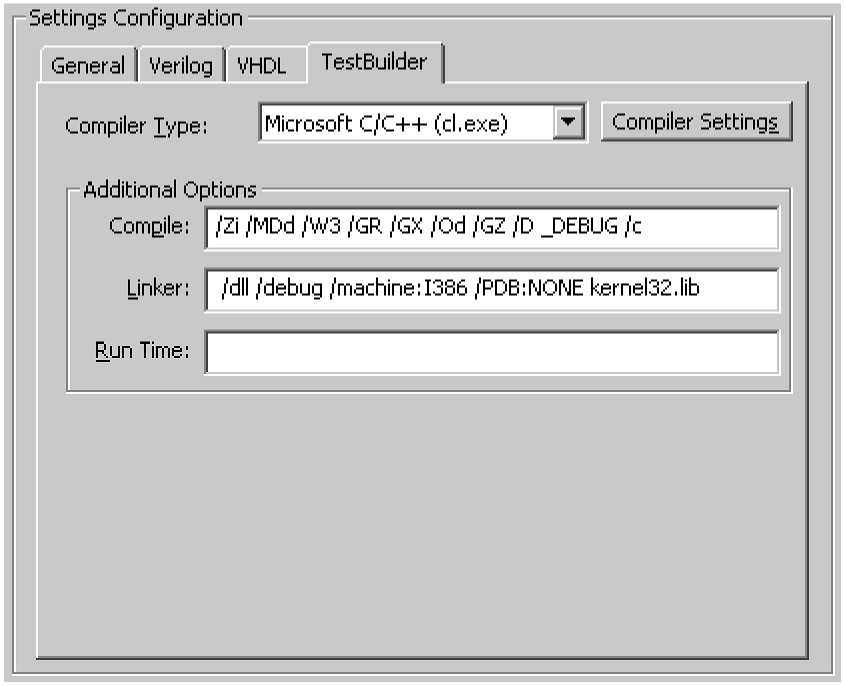
Simulator:

- The **Simulator Settings** button opens the *Simulator / Compiler Settings* dialog where you can edit the simulator paths.
- The **VHDL 93** checkbox specifies that the project dialect for the generated files is **VHDL 93**.
- 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.

TestBuilder Tab:

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

- **Compiler Type** specifies 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.



Compiler Type: Microsoft C/C++ (cl.exe)

Compile: /Zi /MDd /W3 /GR /GX /Od /GZ /D _DEBUG /c

Linker: /dll /debug /machine:I386 /PDB:NONE kernel32.lib

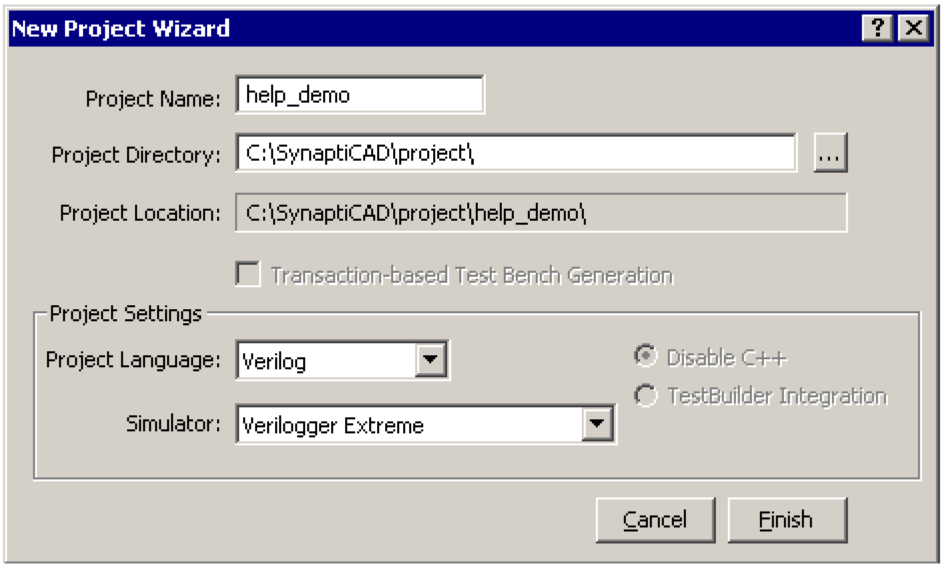
Run Time:

Step 3: Create a Project

BugHunter Pro uses a project file to store the list of files to be simulated and the simulation 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.

Create a New Project:

- Choose the **Project > New Project** menu to open the *New Project Wizard* dialog.

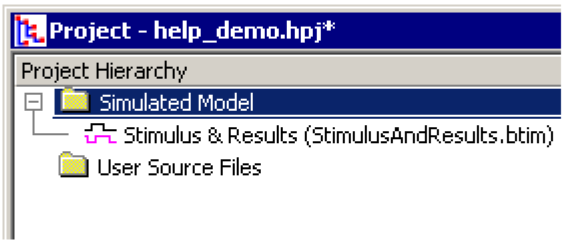


New Project Wizard

Project Name: help_demo
Project Directory: C:\SynaptiCAD\project\
Project Location: C:\SynaptiCAD\project\help_demo\
[ ] Transaction-based Test Bench Generation

Project Settings
Project Language: Verilog
Simulator: Verilogger Extreme
(●) Disable C++
( ) TestBuilder Integration

- In the **Project Name** box, enter the name of the project file.
- Enter 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.
- If you are running VeriLogger Extreme or VeriLogger Pro the **Project Language** and a **Simulator** will already be set, otherwise set these properties.
- Press the **Finish** button to create a new project with several empty folders and a default Stimulus and Results timing diagram.



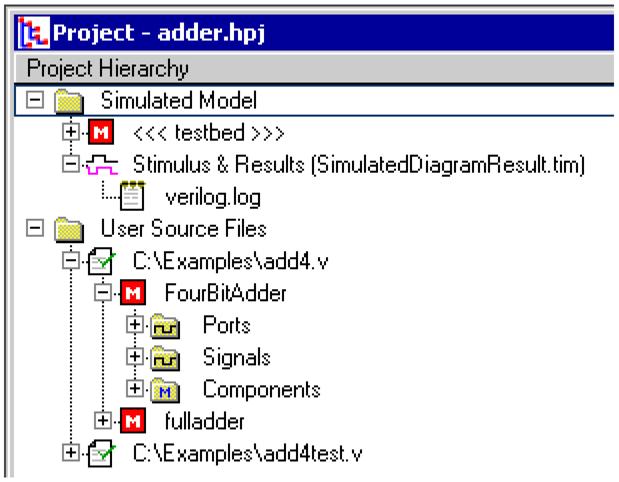
Project - help_demo.hpj*

Project Hierarchy
- Simulated Model
  - Stimulus & Results (StimulusAndResults.btim)
- User Source Files

Working with the Project Window:

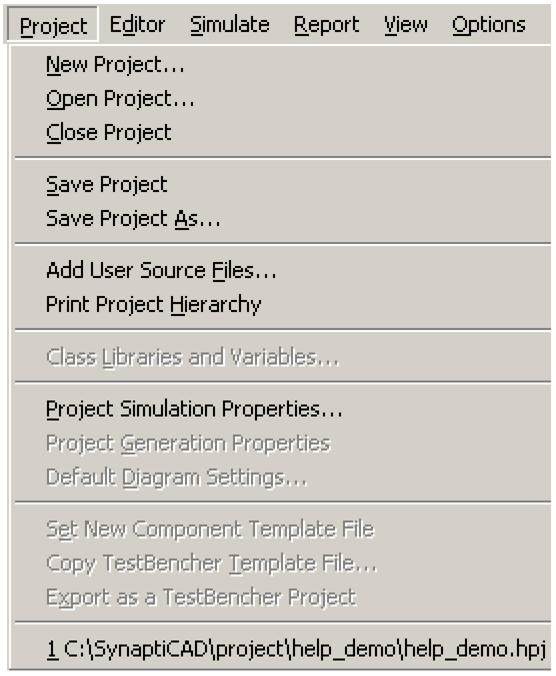
The *Project* window can be used to open source code editors, set watches on signals, and set the Stimulus and Results diagram. After a project is built as described in Step 6, the *Project* window can be used to investigate the hierarchical structure of the design. Each node in the tree has a context sensitive pop-up menu that can be opened by right clicking on the node.

- **Expand** or **Hide** a branch by pressing + or - symbols.
- **View Source Code** by double clicking on a file name, port, signal, component, or port to open an editor window (see Chapter 4: Editor Functions).
- **View Simulation results** by opening the *Stimulus & Results* diagram.
- **Right click** on a node to view all of the available menu options.



Most of the project level features like saving, opening, creating, and editing the settings are accessed through the **Project** menu options.

- The bottom of the Project menu has a list of recently opened projects.
- All projects should have an file extension of **HPJ**.


Project
Editor
Simulate
Report
View
Options
New Project...
Open Project...
Close Project
Save Project
Save Project As...
Add User Source Files...
Print Project Hierarchy
Class Libraries and Variables...
Project Simulation Properties...
Project Generation Properties
Default Diagram Settings...
Set New Component Template File
Copy TestBencher Template File...
Export as a TestBencher Project
1 C:\SynaptiCAD\project\help_demo\help_demo.hpj

Step 4: Add Source Files to the Project

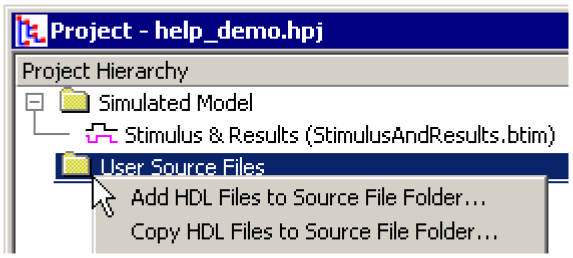
Once the project is created you can create new source files using the built in editors. Then add the source code files to the project so that BugHunter will know the location of the files to compile.

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.The **Editor** menu contains functions that act on the editor windows and Chapter 4: Editor Functions covers all of the editing features.

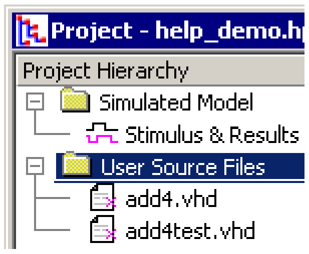
Add the source files to the project:

- Right click on the ***User Source Files*** folder and choose one of the **FIles to Source File Folder** menus to open a *file* dialog.
- The **Copy** menu copies the source file to the project folder and adds it to the project list.


Project - help_demo.hpj
Project Hierarchy
Simulated Model
Stimulus & Results (StimulusAndResults.btim)
User Source Files
Add HDL Files to Source File Folder...
Copy HDL Files to Source File Folder...

- The **Add** function adds the file and its path without moving it to the project folder. Files can also be added by choosing the **Project > Add User Source File(s)** menu from the main bar.

- When files are first added to the project, you can see the filename but you cannot see a hierarchical view of the components inside the files. This is shown by the pink X on the node. To view the internal components on the project tree you must first **build** or **run** a simulation as described in Step 6: Build the Project.



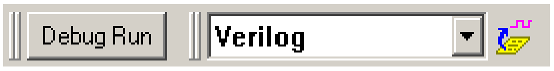
- Once the files are added to the project, double clicking on a source file name in the *Project* window automatically launches an editor window.

Step 5: Draw a Test Bench (optional)

If your top-level component 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. Each time a simulation is run (see Step 7: Simulate and Debug), BugHunter will create a test bench component from the drawn waveforms. A wrapper component that hooks up the test bench component to the design model is created at the same time.

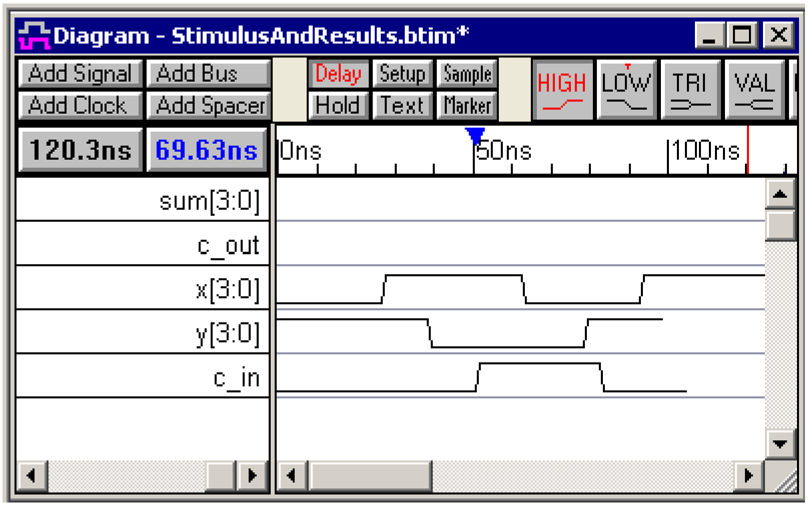
Draw a Stimulus Test Bench for unit level testing:

- Make sure the simulation mode is set to **Debug Run**, rather than ***Auto Run***, so that the simulator does not re-simulate while you are drawing.



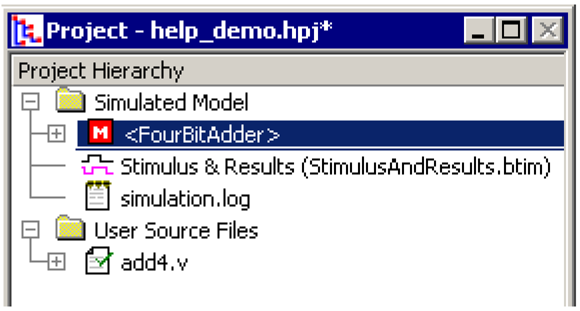
- Press the **Parse MUT** button to extract the port signal names and sizes and put them in the **Stimulus and Results** diagram. This will also populate the project window with the hierarchical.

- Draw waveforms on the input signals, which will be drawn in black.
- If you have the *Reactive Test Bench* option then you may also wish to draw waveforms on input signals to indicate the expected inputs to the testbench (or outputs from the model under test), and these waveforms will be drawn in blue.



Changing the Model Under Test:

- The Parse MUT function makes a guess as to which model is the model under test and displays that model with single brackets, <>, underneath the **Simulated Model** folder.



- To pick a different model under test, first right click on the MUT and choose **Unset Current Model Under Test**, and then right click on a different model under the *User Source Files* list and pick **Set as Model Under Test**.
- Then press **Parse MUT** button to re-populate the *Stimulus and Results* diagram.

Step 6: Build the Project and Set the top

Building the project compiles the source files, fills the Project window with the hierarchical structure of the design, and sets watches on all the signals and variables in the top-level component. A build will automatically be done each time the simulation is run, but having a separate build button enables you to create the project tree without having to wait for a simulation to run. After the build you are also able to set the top level component for the project and/or select additional signals to watch using the project tree context menus.

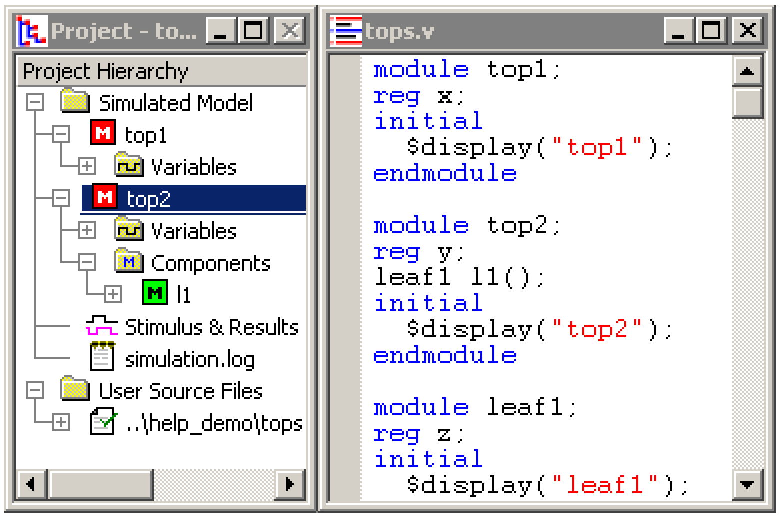
Three ways to build a project:

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

Set the <<<Top Level Component>>>:

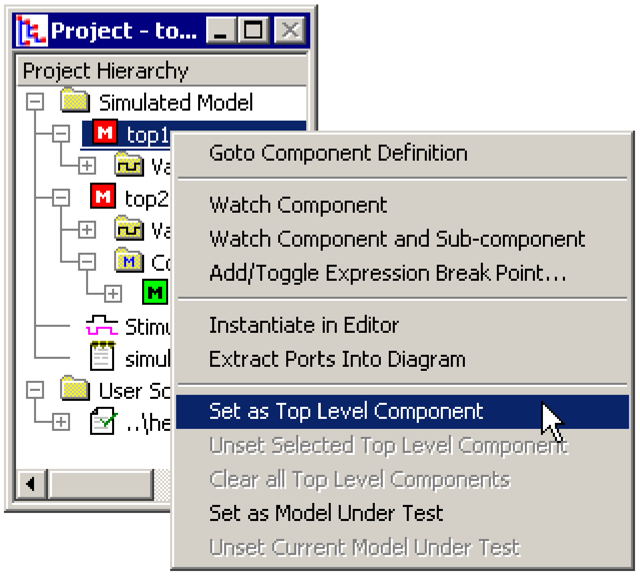
In languages that support multiple top-level components, BugHunter will find all of of the components that are not instantiated in any other component and list them under the ***Simulated Model*** tree without any brackets. If the language only supports one top-level, the program will grab the first that it finds in the files. All the top-level components will be simulated. Any component can be specified as top-level, by using the context menu.

- In this example, after the first build, both top1 and top2 will be listed under Simulated Model, because neither component is instantiated in another component. Both are default top-level modules and will be simulated simultaneously.

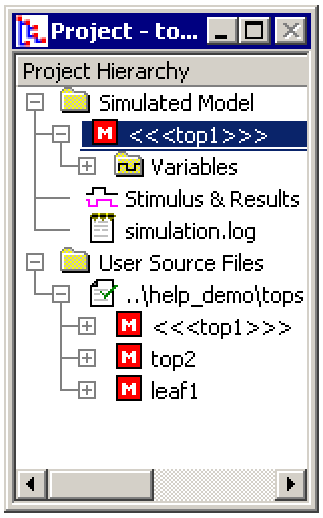


```
module top1;
reg x;
initial
  $display("top1");
endmodule

module top2;
reg y;
leaf1 l1();
initial
  $display("top2");
endmodule

module leaf1;
reg z;
initial
  $display("leaf1");
```

- To set one component as the top level, find the component under the ***Simulated Model*** or the ***User Source Files*** folders and right click and choose the **Set as Top Level Component** from the context menu.


Project - to...
Project Hierarchy
Simulated Model
top1
top2
Goto Component Definition
Watch Component
Watch Component and Sub-component
Add/Toggle Expression Break Point...
Instantiate in Editor
Extract Ports Into Diagram
Set as Top Level Component
Unset Selected Top Level Component
Clear all Top Level Components
Set as Model Under Test
Unset Current Model Under Test

- The top level component is displayed with triple brackets <<<>>> around the name.
- Now, only top1 will be simulated. The top2 component will not be simulated because it is not instantiated within top1.
- To undo this operation so that the default top-level components are automatically chosen by the tool, right click on the component and choose **Unset as Current Top-level** or **Clear all Top Level Components**.

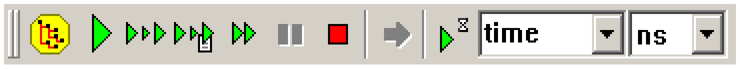


Step 7: Simulate and Debug

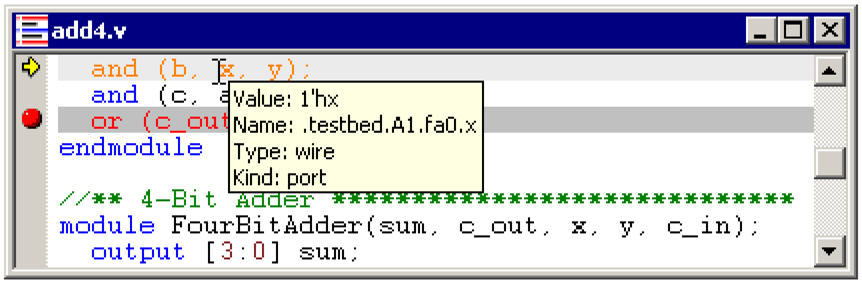
BugHunter can perform a variety of graphical debugging functions which are covered in detail in Chapter 2: Simulate and Debugging Functions. Basically, you will start the simulator and view the results either in the *Stimulus and Results* diagram or in one of the tabs of the *Report* window.

Start the Simulator:

- Start the simulator by pressing one of the ***green buttons*** on the *Build and Simulate* button bar. Section 2.1 Build and Simulate explains the differences between the types of single stepping and running.



- When single stepping, the ***yellow arrow*** in the editor window indicates the line of code that will be simulated next. The ***red dots*** are breakpoints. And variables can be inspected by moving the mouse cursor over the variable in the editor window.

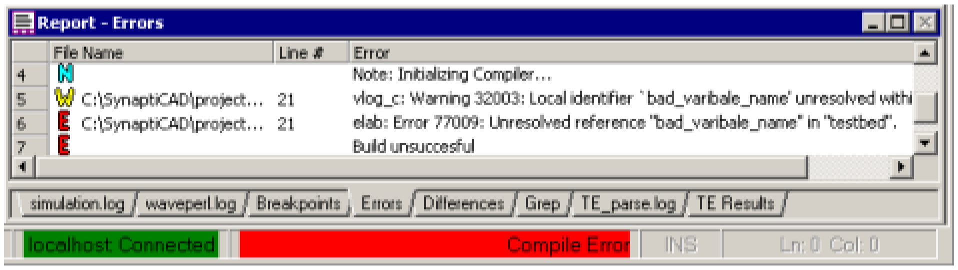


```
  and (b, x, y);
  and (c, a
  or (c_out
endmodule

//** 4-Bit Adder ******************************
module FourBitAdder(sum, c_out, x, y, c_in);
  output [3:0] sum;
```

Value: 1'hx
Name: .testbed.A1.fa0.x
Type: wire
Kind: port

Check for Errors:

- The status bar in the lower right hand corner displays a red message if an error is found during the build or simulation.
- In the ***Errors*** tab of the *Report* window, double click on an error to open an editor window that will display the code the caused the error. If you cannot see the *Report* window, select **Window > Report** menu to bring the window to the front.

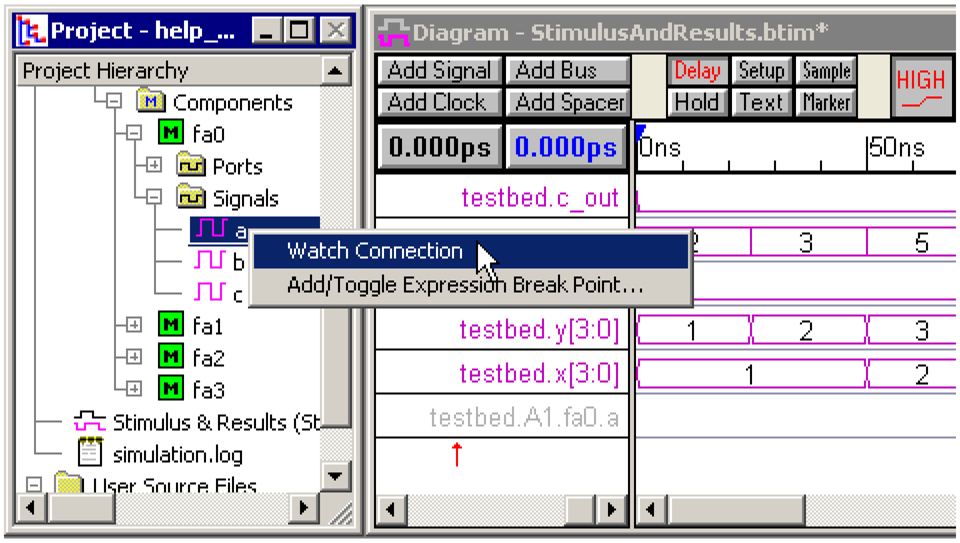


```
Report - Errors
   File Name                  Line #  Error
4  N                                  Note: Initializing Compiler...
5  W C:\SynaptiCAD\project...  21     vlog_c: Warning 32003: Local identifier `bad_varibale_name' unresolved withi
6  E C:\SynaptiCAD\project...  21     elab: Error 77009: Unresolved reference "bad_varibale_name" in "testbed".
7  E                                  Build unsuccesful
```

- The ***Simulation Log*** tab also displays error messages and other messages that are produced by the simulator, however these are not linked to the code.
- The ***Waveperl Log*** *tab* will display error messages that are associated with test bench code generation. Usually, only TestBencher Pro and Reactive Test Bench users need to check this tab.

View Waveform Simulation Results:

The signals in the top-level module will automatically be put into the *Stimulus and Results* diagram and waveforms will be displayed as the simulation progresses. Signals can be added to the diagram by right clicking on the desired signal in the Project window and setting a **watch** on it. Signals can be removed by deleting them from the *Stimulus and Results* diagram. See Chapter 3 for information on using multiple *Stimulus and Results* diagrams.



Step 8: Save the Project, Code and Waveform Files

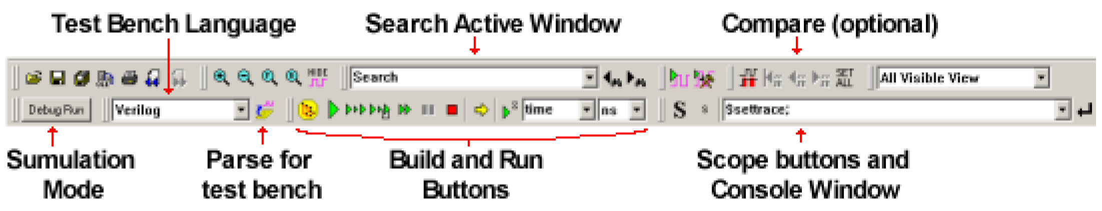
In BugHunter there are three types of files associated with a project.

- Project files have an extension of **hpj** and are saved by using the **Project > Save HDL Project** menu option. This saves the list of files that compose the current project and related simulation options. It does not save the watched signals list.
- 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.

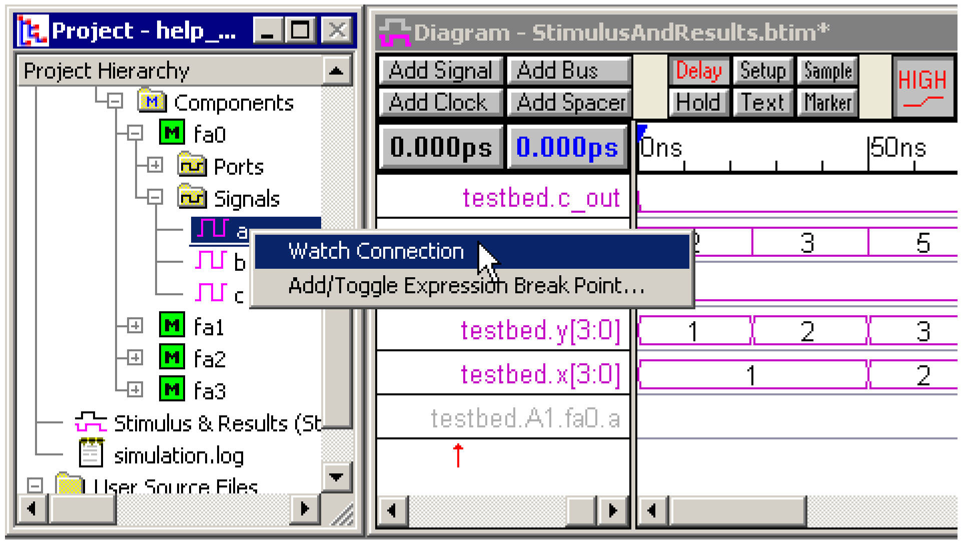
Saving watched signals in separate diagram files allows you to build several different test cases so you can compare and contrast future simulation results.

Chapter 2: Simulation and Debugging Functions

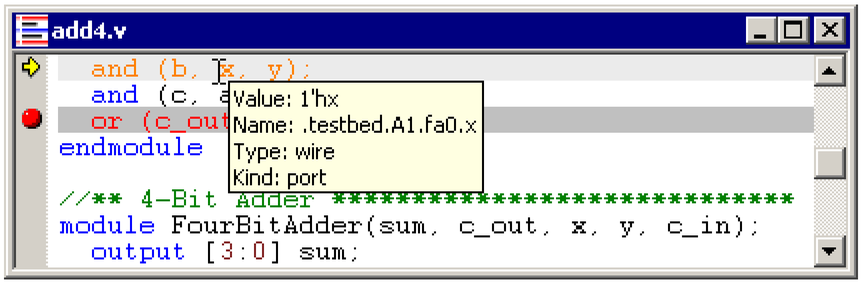
The Simulation Button Bar controls when and how simulations are performed. The interactive command console window can be used to enter simulator commands to observe and control variables and models during simulation. The Search Active Window box will search diagrams for signals, or the project window for anything.



The Stimulus and Results diagram shows the simulated waveforms. Additional signals can be added by right clicking on the signal or component in the *Project* window and setting a **watch** on the object and then re-simulating (or continue simulating).



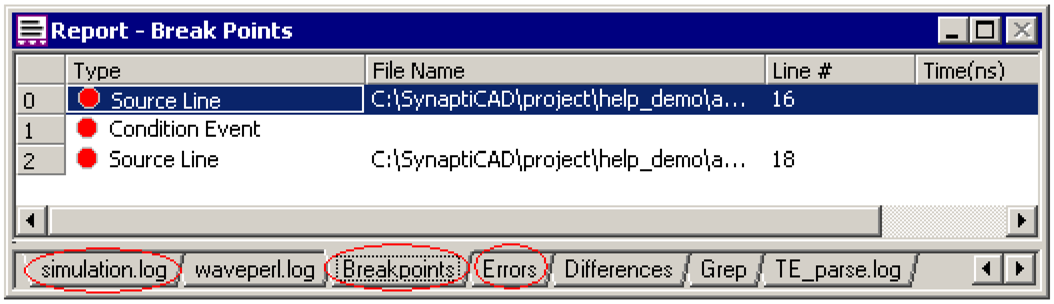
Quickly inspect a variables current value by placing the mouse over a variable in the *Edit* window. Or use the **Simulate > Inspect values** menu to investigate variables values at different times during the simulation. Breakpoints can be added to the source code by placing red dots in the grey bar to the left of the code. The current simulation line is indicated by the yellow arrow.



```
  and (b, x, y);
  and (c, a
  or (c_out
endmodule

//** 4-Bit Adder ******************************
module FourBitAdder(sum, c_out, x, y, c_in);
  output [3:0] sum;
```

```
Value: 1'hx
Name: .testbed.A1.fa0.x
Type: wire
Kind: port
```

The *Report* window manages several tab windows are important to simulation and debugging. The **simulation.log** file displays the default log file for the simulator. The ***Breakpoints*** tab displays all of the breakpoints that are set on the code in the editor windows and the components in the project file. And the ***Errors*** tab displays any compile or simulation bugs that are found in the design.


Report - Break Points

	Type	File Name	Line #	Time(ns)
0	● Source Line	C:\SynaptiCAD\project\help_demo\a...	16	
1	● Condition Event			
2	● Source Line	C:\SynaptiCAD\project\help_demo\a...	18	

2.1 Build and Simulate

BugHunter has two simulation modes, **Auto Run** and **Debug Run** that determine when a simulation is performed. 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. The Auto Run mode makes it easy to quickly test small components and do bottom-up testing. Click the mode button to toggle between the two simulation modes.

Auto Run OR Debug Run

The active simulation mode is displayed on the left most button on the simulation button bar.

The build and simulate functions are accessed from the simulation button bar located at the top of the main window.

Build - compiles the project files, builds the hierarchical tree, populates the Stimulus and Results diagram, and if necessary generates a testbench. It does not run a simulation. The **<F7>** key and the **Simulate > Build** menu also perform the same function.

Run/Resume - compiles the files (if there have been changes since the last build) and then runs a simulation until it is stoped by a breakpoint, the pause button, the stop button, or the end of the simulation is reached. This button also continues a simulation when it is currently paused. The **<F5>** key and the **Simulate > Run** menu also perform the same function.

Step Into - steps to the next line of code and will also step into function calls.

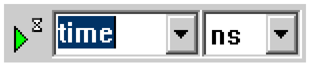
Step Into With Trace Calls - steps to the next line of code and also sends a trace statement to the **simulation.log** file. This button will also step into function calls.

Step Over Calls - steps to the next line of code. It does not 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.

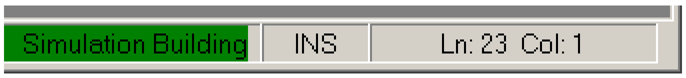


Run To Time button bar runs the simulation for the specified time. Type in a time into the time box, pick the units of time, and then click the green triangle with the hourglass button.

First Build to debug syntax errors:

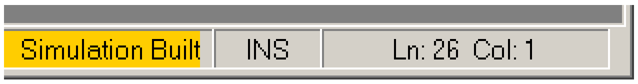
- Normally you will first press the **Build** button to compile the code and debug any syntax errors. The status of the build is reported in the lower right hand corner of the screen.

- **Simulation Building** means that the compile is still compiling.



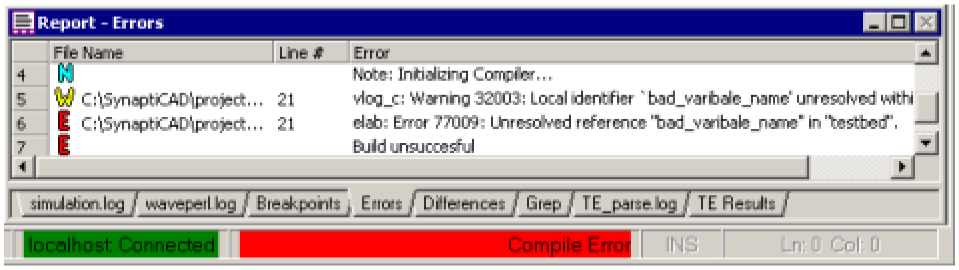
Simulation Building	INS	Ln: 23 Col: 1

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



Simulation Built	INS	Ln: 26 Col: 1

- **Compile Error** means that the compile failed and the syntax errors will be listed in the *Error* tab of the *Report* window (see 2.5 Report Window Error and Log file tabs). Double click on an error to be taken to the code that caused the error.



Report - Errors

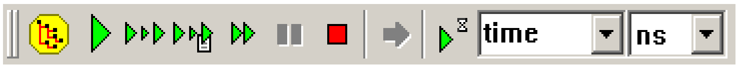
	File Name	Line #	Error
4	N		Note: Initializing Compiler...
5	W C:\SynaptiCAD\project...	21	vlog_c: Warning 32003: Local identifier `bad_varibale_name' unresolved withi
6	E C:\SynaptiCAD\project...	21	elab: Error 77009: Unresolved reference "bad_varibale_name" in "testbed".
7	E		Build unsuccesful

simulation.log / waveperl.log / Breakpoints / Errors / Differences / Grep / TE_parse.log / TE Results

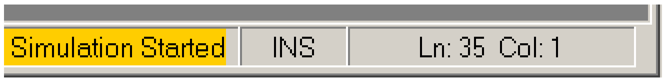
localhost Connected	Compile Error	INS	Ln: 0 Col: 0

Then Run the simulation:

- Press one of the green Run buttons to start the simulator. The status of the simulator is reported in the lower right hand corner of the screen

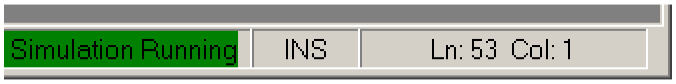


- **Simulation Started** shows that the simulation has been paused by either a breakpoint or the pause button.



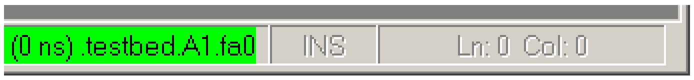
Simulation Started	INS	Ln: 35 Col: 1

- **Simulation Running** shows that the simulation is currently running and may be stoped using the pause or stop button.



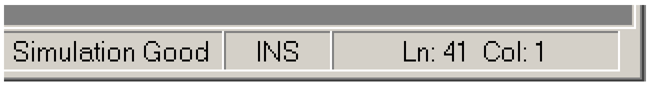
Simulation Running	INS	Ln: 53 Col: 1

- When single stepping through a simulation, the simulation time and scoping level are listed in the status bar.



(0 ns) .testbed.A1.fa0	INS	Ln: 0 Col: 0

- **Simulation Good** is displayed when the simulator is completed without errors.



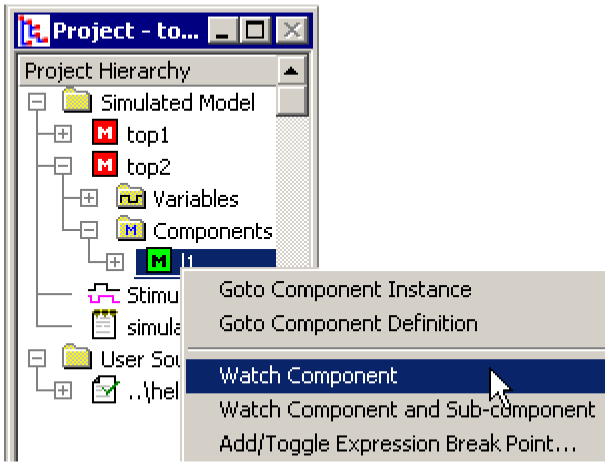
Simulation Good	INS	Ln: 41 Col: 1

2.2 Watching Signal and Component Waveforms

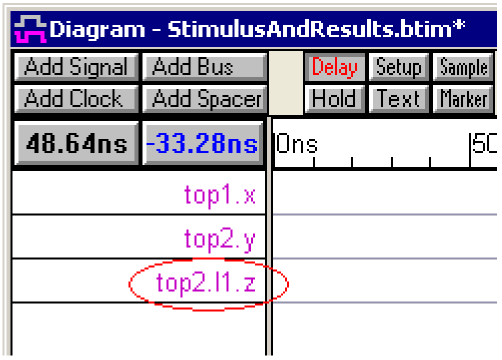
After compiling the project, use the *Project* window to pick signals to be watched and placed in the Stimulus and Results diagram. To maximize simulation speed, simulators do not automatically store signal transition times unless a signal is specifically tagged as one to watch. Chapter 3: Waveforms and Test Bench Generation covers all the the intricacies of managing multiple Stimulus and Results diagrams.

Watch anything under Simulated Model: signals, ports, variables, or components

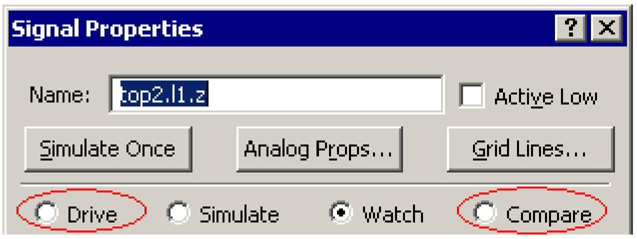
- Expand the ***Simulated Model*** folder until you locate something that you would like to watch.
- Right-click on the node and choose one of the **Watch** menus, which will vary according to what type of object is selected.



- After setting the watch, the signal name will appear in the *Stimulus and Results* diagram. The waveform data will be displayed during the next simulation run.
- To **remove a watched signal**, just delete it from the *Stimulus and Results* diagram.



- To **temporarily stop watching** a signal, double click on the signal name to open the *Signal Properties* dialog and change the signal type from ***watch*** to ***drive*** or ***compare***.

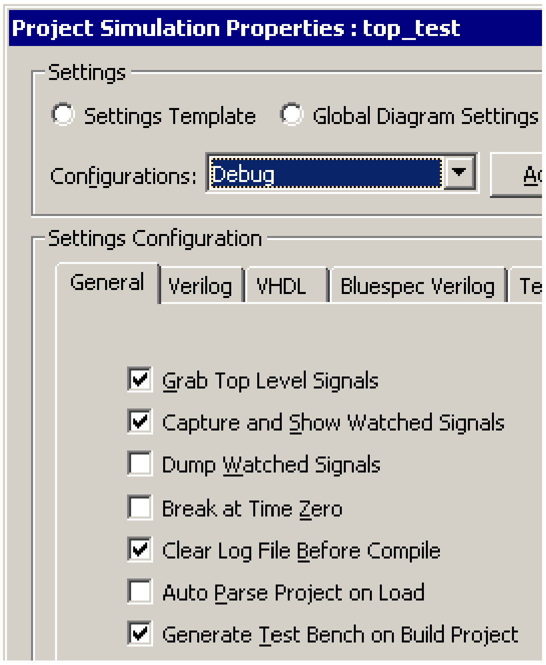


Top-level Models are automatically watched:

After you build the project, the signals or the ports in the top-level component are automatically added to the *Diagram* window. If the top-level component does not have port signals, the internal signals of the component are viewed. If the top-level component 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 component. The waveform drawing functions are covered in Section 3.2 Drawing Waveforms for Stimulus Generation.

Global Settings for watch signals:

- Select the **Project > Project Simulation Properties** menu to open the *Project Simulations Properties* dialog.
- **Grab Top Level Signals** allows BugHunter to grab the signals in the top-level components and set them as the default watch signals.
- **Capture and Show Watched Signals** causes watched signals to display their waveform data in the *Stimulus and Results Diagram*. Normally this is unchecked if the ***Dumped Watch Signals*** is checked.
- **Dump Watched Signals** will cause the watched signal data to be written to a Verilog dump file. This is normally unchecked because the *Stimulus and Results Diagram* is a much faster and more compressed format than VCD.


Project Simulation Properties : top_test

Settings
( ) Settings Template ( ) Global Diagram Settings
Configurations: Debug

Settings Configuration
General | Verilog | VHDL | Bluespec Verilog | Te

[✓] Grab Top Level Signals
[✓] Capture and Show Watched Signals
[ ] Dump Watched Signals
[ ] Break at Time Zero
[✓] Clear Log File Before Compile
[ ] Auto Parse Project on Load
[✓] Generate Test Bench on Build Project

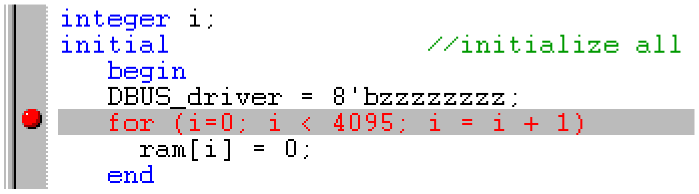
2.3 Breakpoints

Breakpoints pause the simulation at a particular source code line, simulation time, or activity on a particular variable. The *Report* window displays the list of breakpoints in the **Breakpoints** tab. Each break point can be also be temporally made inactive without having to remove the breakpoint from the project.

Add Source Code Breakpoints through the Editor Windows:

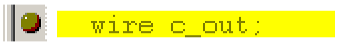
Source code breakpoints stop the simulator each time a particular line of code is executed.

- In an *Editor* window, click on the gray line on the left side of the window, to add a breakpoint, indicated by the red circle on the line.



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

- During simulation, a source code breakpoint can turn grey to indicate that it is on an invalid line of code.

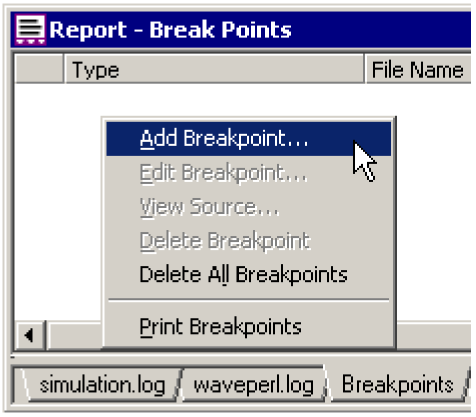


```
  wire c_out;
```

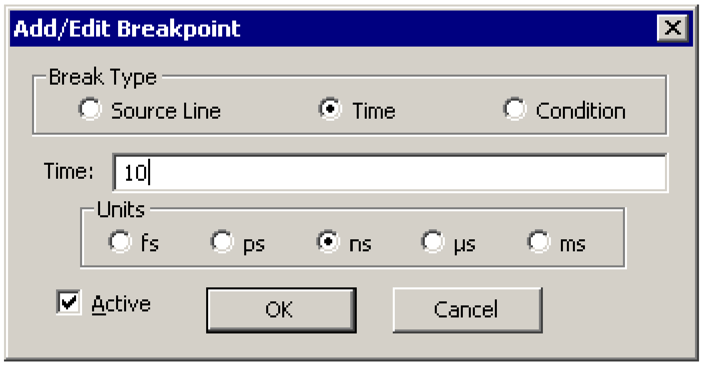
Add Time Breakpoints through the Report Window Breakpoint Tab:

Time based breakpoints stop the simulator at a particular simulation time.

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



- Select the **Time** radio button to change the dialog to the time configuration.
- Enter a time and a time unit, then press Ok to close the dialog.



Break Type: Source Line [✓ Time] Condition

Time: 10

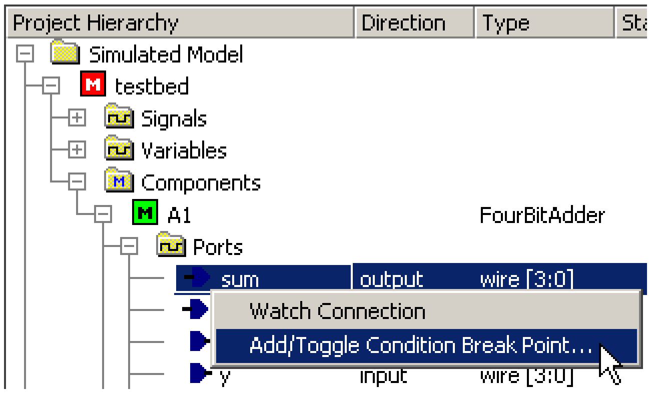
Units: fs ps [✓ ns] µs ms

[✓ Active]

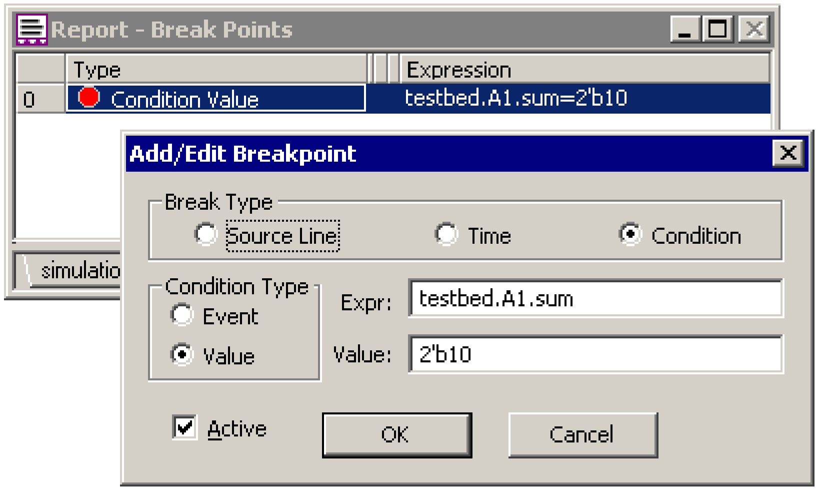
Add Condition Breakpoints through the Project window:

Condition breakpoints will break every time a particular variable/signal changes or every time it reaches a specific value.

- The easiest way to add a Condition breakpoint is to find the variable in the Project tree and right click and choose choose **Add/Toggle Condition Breakpoint** menu. This will open the *Add/Edit Breakpoint* dialog with the Expr box filled.



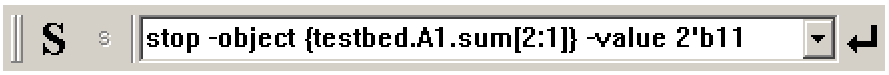
- If the **Event** condition type is chosen, then the simulation will break on any change in the expression listed in the **Expr** box.
- If the **Value** condition type is chosen, then the simulation will break only when the expression matches the value in the **Value** edit box.



	Type	Expression
0	● Condition Value	testbed.A1.sum=2'b10

Break Type: Source Line Time [✓ Condition]
Condition Type: Event [✓ Value]
Expr: testbed.A1.sum
Value: 2'b10
[✓ Active]

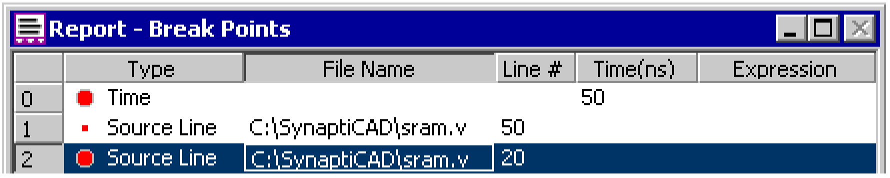
- Most simulators only accept a hierarchical signal name for the **Expr** in a condition breakpoint, but some simulators accept more complicated expressions. Graphical breakpoints generate simulator ***stop*** console commands, so these condition breakpoints should have the same functionality as that command. Below is an example of a console command for a value based on a bit slice of the variable. In the breakpoint GUI, you would enter *testbed.A1.sum[2:1]* into the Expr box, and *2'b11* into the value box.



```
stop -object {testbed.A1.sum[2:1]} -value 2'b11
```

Turn Breakpoints ON and OFF using the Breakpoint Tab window:

- Each breakpoint, regardless of how or where it is added, will be listed in the ***Break points*** *tab* in the *Report* window.



	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		

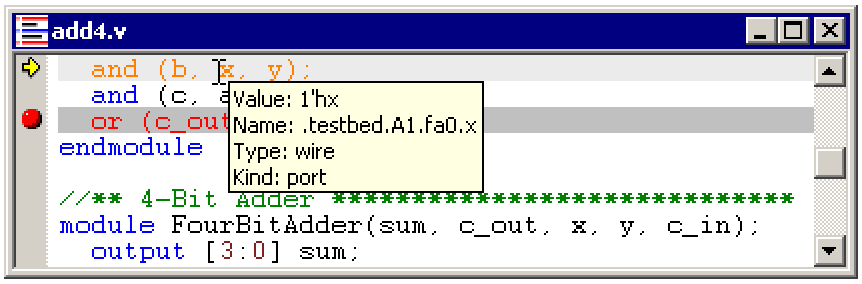
- Clicking on a red breakpoint button will toggle it between active and inactive states. An inactive breakpoint is displayed as a small red circle and is ignored in a simulation.
- Double-clicking on a source code breakpoint will open an editor starting at that line in the source code.
- Right-clicking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

2.4 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.

Use the Editor window to inspect current values:

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



```
  and (b, x, y);
  and (c, a
  or (c_out
endmodule

//** 4-Bit Adder ********************************
module FourBitAdder(sum, c_out, x, y, c_in);
  output [3:0] sum;
```

Value: 1'hx
Name: .testbed.A1.fa0.x
Type: wire
Kind: port

Use Inspect Values dialog to inspect at previous simulation times:

- Choose **Simulate > Inspect Values** menu option to open the dialog.

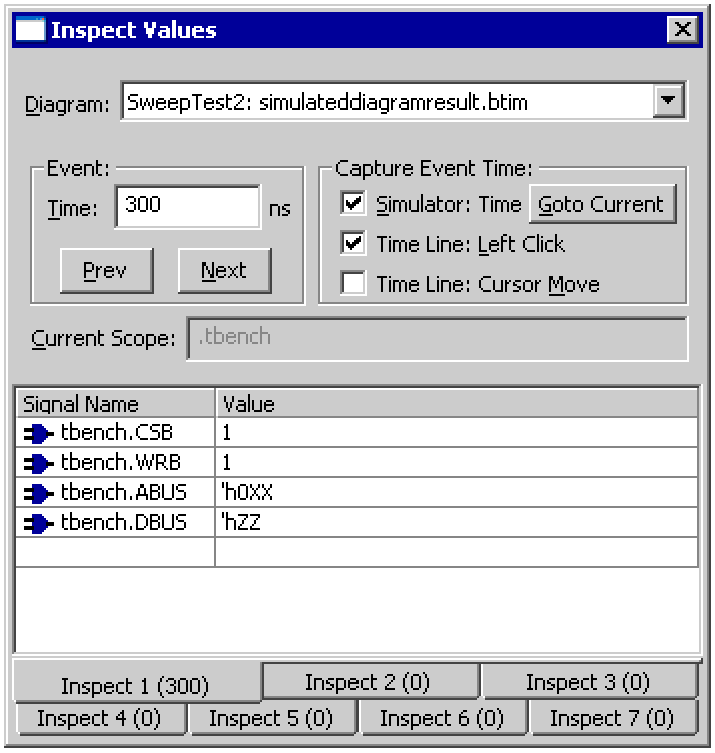


Diagram: SweepTest2: simulateddiagramresult.btim

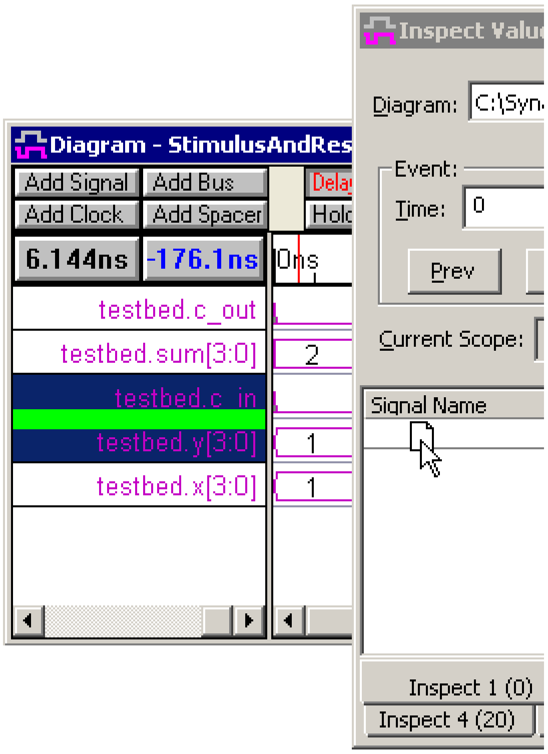
Event: Time: 300 ns

Capture Event Time:
[✓] Simulator: Time
[✓] Time Line: Left Click
[ ] Time Line: Cursor Move

Current Scope: .tbench

Signal Name	Value
tbench.CSB	1
tbench.WRB	1
tbench.ABUS	'h0XX
tbench.DBUS	'hZZ

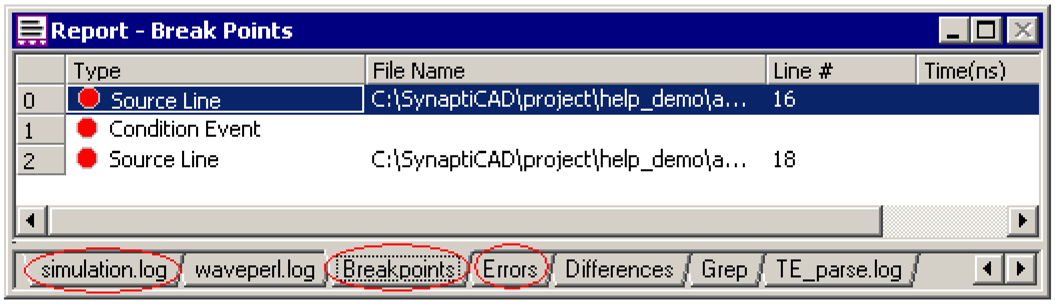
- Drag and Drop signals from the *Stimulus and Results* diagram into the **Signal Name** box. Use the green bar at the top of the signals when doing the drag.
- You may also type variable names into the **Signal Name** boxes. However, only variables that are also displayed in the *Stimulus and Results* diagram will be able to view previous values. In order to speed simulation times, the tool only remembers the current values for all of the variables and the diagram stores the previous values for just the signals that are specified as important.



- 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 the information is coming from the simulator or from 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 affect the scope of the variables in the dialog. S **and** s
- There are 7 different inspect tabs that let you group a set of related variables together for easier debugging.
- When **Simulator: Time** 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** affect 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.

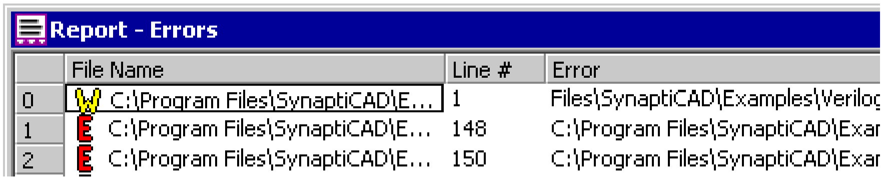
2.5 Report Window Error and Log file tabs

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


Report - Break Points

	Type	File Name	Line #	Time(ns)
0	● Source Line	C:\SynaptiCAD\project\help_demo\a...	16	
1	● Condition Event			
2	● Source Line	C:\SynaptiCAD\project\help_demo\a...	18	

- The ***simulation.log*** 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.
- The ***Errors* tab** displays errors an warnings that are hyperlinked to the actual code that threw the message. 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	W C:\Program Files\SynaptiCAD\E...	1	Files\SynaptiCAD\Examples\Veriloc
1	E C:\Program Files\SynaptiCAD\E...	148	C:\Program Files\SynaptiCAD\Exar
2	E C:\Program Files\SynaptiCAD\E...	150	C:\Program Files\SynaptiCAD\Exar

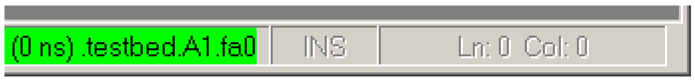
- The ***Breakpoints* tab** is shows a tabular form for all the breakpoints in the current project. This is covered in Section 2.3: Breakpoints.

2.6 Console Window for Interactive Debugging

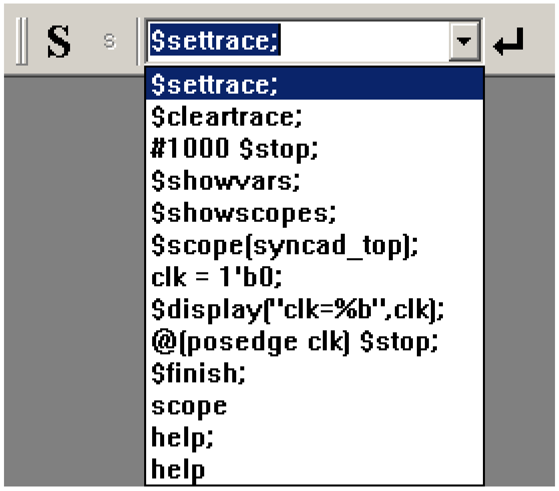
BugHunter has an interactive command console for entering simulator commands to observe, control, and debug a simulation. For example, during an interpreted Verilog simulation, you can enter a Verilog command such as **$finish;** (to end the simulation) or **$display;** (to display the value of a variable). 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 console.

To use the command console window:

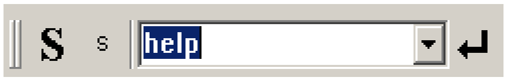
- Stop the simulator during a simulation run by either (1) single stepping into the design, (2) hitting a breakpoint, (3) pressing the pause button, or (4) inserting a **$stop** system task into the code. When a simulation is stopped, the simulation display on the status bar turns bright green and displays the current simulation time and scoping level.

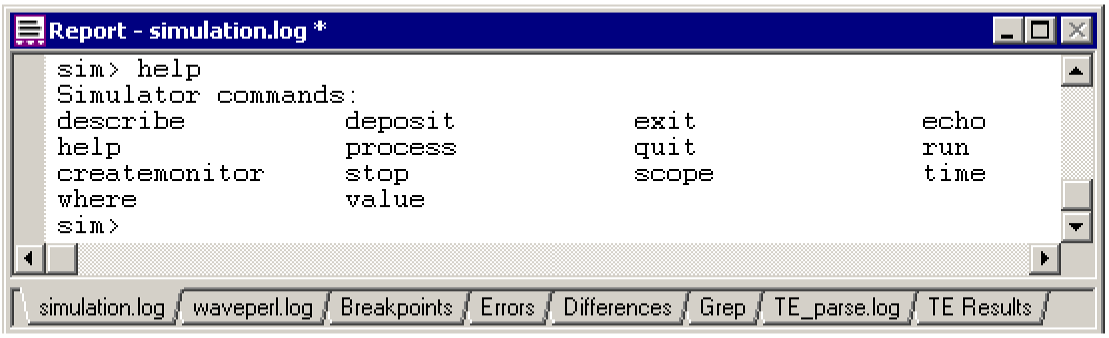


- Type a command into the console window or pick one from the drop-down list and press the <Enter> key.
- The scope buttons change the scoping level for the commands in the console window. The "S" changes the scope to the top-level component. The "s" changes scope to the current simulation level.



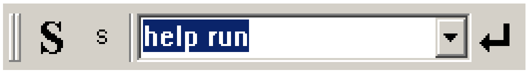
- Type the **help** command to retrieve a list of available commands for your simulator. BugHunter just passes the command to the simulator and there is not a standard list of commands that all simulators support. The list is displayed in the simulation.log tab of the *Report* window.

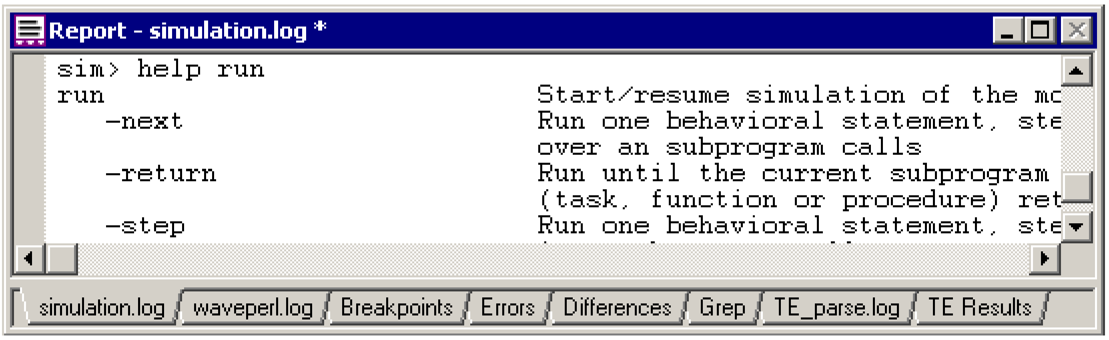




```
sim> help
Simulator commands:
describe           deposit            exit               echo
help               process            quit               run
createmonitor      stop               scope              time
where              value
sim>
```

- To get more information about a specific command, type **help *name_of_command***.





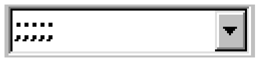
```
sim> help run
run                           Start/resume simulation of the mc
   -next                      Run one behavioral statement, ste
                              over an subprogram calls
   -return                    Run until the current subprogram
                              (task, function or procedure) ret
   -step                      Run one behavioral statement, ste
```

Some VeriLogger Pro and Cadence Verilog XL commands:

Interpreted Verilog simulators such as VeriLogger Pro and VerilogXL can execute lines of Verilog behavioral code entered into the console window. These commands will not work with VeriLogger Extreme and other compiled simulators.

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:

- For example: 

  ;;;;;

  would cause the simulator to execute five lines of code.
- 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 **verilog.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 **$finish;** command or press the red **STOP** button.
- **Displaying Variables:** Use the **$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 component, 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 file name, 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 execution, 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 3: 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 component has input ports, BugHunter can take the drawn waveforms in the stimulus and results file and create a stimulus component and a wrapper component 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 BTIM or VCD waveform file and turn off the graphical display of waveform data during simulation. After the simulation you can load in the waveform file and view it graphically. Also, you can run Verilog simulations from the command line as described in Chapter 5: VeriLogger Command Line Simulators.

3.1 Stimulus and Results Diagram

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 3.2: 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.

3.2 Drawing Waveforms for Stimulus Generation

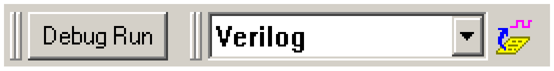
BugHunter can generate stimulus code for signals drawn in the *Stimulus and Results* diagram and use that code as inputs to the project. At the beginning of a compile, BugHunter will take the drawn waveforms and create a stimulus component. It will also create a top-level component that will hook up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

Input Signals: After you build the project, the input and output ports of the top-level component 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:

The simulation mode button, generation language and Parse MUT buttons are used for the simulation and generation of testbenches

- Make sure the simulation mode is set to **Debug Run**, rather than ***Auto Run***, so that the simulator does not re-simulate while you are drawing.



- The testbench generation language is listed in the drop-down.
- Press the **Parse MUT** button to populate the *Stimulus and Results* diagram with the component’s input and output ports.
- Draw waveforms on the input signals. Timing Diagram Editors manual Chapter 1: Signals 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 components.
- 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 2.2: 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.

3.3 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 Diagram Analysis, Reactive Test Bench Generation, and GigaWave require optional feature licenses. The timing diagram editor is described in the Timing Diagram Editors manual on-line help.

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

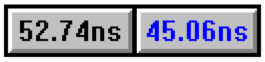
- Click in the time line in the *Diagram* window. This displays a marker line that shows the value of each signal at that particular time.

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, with the black writing, displays the current position of the mouse cursor in the drawing window. The Delta Button, 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.



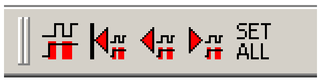
- Clicking on either button opens an edit box that accepts time values.
- Entering a value in the **Time** button causes the drawing window to scroll to that exact time.
- Entering a value in the **Delta** button causes the drawing window to scroll that amount from its current position.

3.4 Waveform Comparisons (Optional Features)

If you have purchased the Comparison option then VeriLogger can 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 in the Timing Diagram Editors manual (these settings will appear when the signal is made into a compare signal).

The **compare** toolbar contains five buttons. The **compare** button, with the yel low 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, labeled **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 signal will appear in red under the original signal.

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 action will make 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 comparison 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 ns.)
- Specify the tolerance range after the edge by typing a value into the **+Tol:** textbox. (This value will also be in ns.)
- 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.

3.5 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 Properteis* 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.
- Click 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 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, choose 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 file names have an extension of **v**, VHDL file names 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 components 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.

Move to a specific line in your code:

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

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 particular file extension.

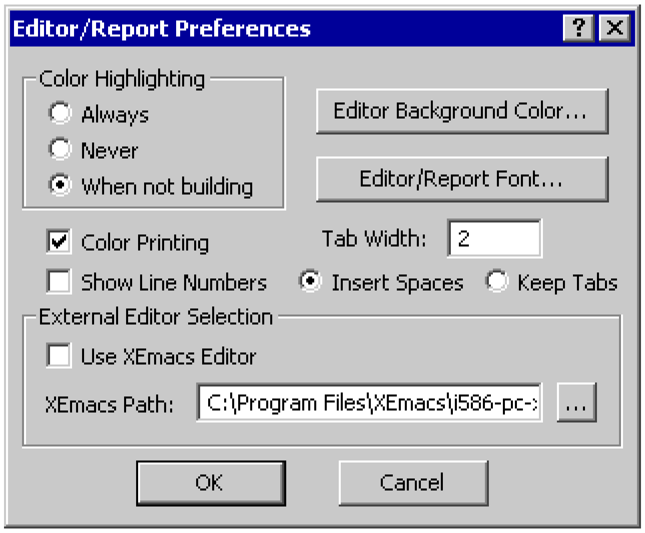
- Select the **Editor > Find in Files…** menu option. This will open the *Find In Files* dialog.
- Type the test phrase that you are searching for in the **Find what:** combo box. **Note:** the drop down list can be used to select a search that was previously performed. If you want to redo a search or 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 options for the *Editor* and *Report* windows. This information is stored inside the project **HPJ** file.

- Select the **Editor > Editor/Report Preferences** menu option to open the *Editor/Report Preferences* dialog:.



Editor/Report Preferences

Color Highlighting: Always [✓ When not building] Never

Color Printing: ✓

Show Line Numbers: ☐

Tab Width: 2

[✓ Insert Spaces] Keep Tabs

External Editor Selection

Use XEmacs Editor: ☐

XEmacs Path: C:\Program Files\XEmacs\i586-pc-:

- 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. From this 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 editor to edit HDL files instead of its internal editor. For more information on this feature see Section 4.5 XEmacs Integratior.
- 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.

Key	Purpose
Left/right arrow keys	Moves the cursor one space left or right
Up/down arrow keys	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 OR deletes the selected text
Delete	Deletes the character to the right of the cursor OR deletes the selected text
Shift+Left	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 OR, cancels the selection if the next window is already selected
Shift+Page Up	Selects text up one window 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 editor
F4	Print from window
Shift+F4	Print options
Ctrl+F	Search and/or Replace Dialog
Tab	Tab
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

- 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 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.)

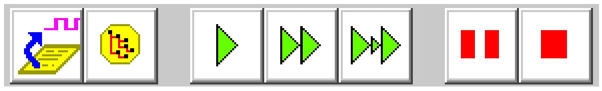
To single-step through a BugHunter simulation from within XEmacs:

- 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 **verilog.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.



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

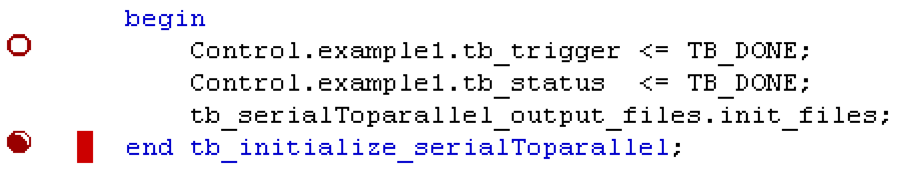
- Place the cursor in the line that the breakpoint should be added to or removed from.
- Press the **<F9>** key.

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.



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

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

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: VeriLogger Command Line Simulators

This chapter describes how to launch the VeriLogger command line simulators (vlogcmd and simx) and the command line options to control the simulation. Vlogcmd is the interpreted simulator that comes with VeriLogger Pro. Simx is the compiled-code simulator that comes with VeriLogger Extreme.

You can enter most of the command options directly into BugHunter’s *Command Console* window on the simulation button bar (see Section 2.6: Console Window for Interactive Debugging). 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.

5.1 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 **verilog.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.

5.2 Using the Command Line Simulator

The examples show how to run the interpreted vlogcmd simulator. Wherever you see vlogcmd, the equivalent can be done with the compiled code simulator by replacing **vlogcmd** with **simx**.

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
```

Using_Command_filesTo 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 5.3) 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 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
```

Automatically generate a command file that contains all project settings project files:

- Select the **Project > Project Settings** menu option to open the *Project Settings* 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.

5.3 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 (vlogcmd can be replaced with simx in the commands below to run the compiled-code simulator). 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 commend 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 **verilog.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
```

5.4 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 (vlogcmd only) displays warnings for constructs that are either not supported or ignored by the Synopsys HDL Compiler.

+noshow_var_change (vlogcmd only) 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 **$showvars** directive. This feature may cause 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_tcell** 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_tfcell* register_syncad_tasks()
 {
 static s_tfcell 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;
 }
```

5.5 VeriLogger Extreme tools: Simx and Simxsim

The command line simulation compiler for VeriLogger Extreme is called **simx**. Simx compiles the user's source files into an executable file called **simxsim**. By default, simx then launches simxsim to run the actual simulation. Simxsim can subsequently be run again standalone with different runtime simulation options without re-running simx, when there is no need to change the HDL source code or compile time simulation options. Running **simxsim -h** will display a list of the command line options that are available for use with simxsim.

The next sections describe command line options that are only available with simx and/or simxsim; they are not supported by the interpreted simulator, vlogcmd.

5.6 Simx Simulation Build Command Line Options

Below are the command line options to control how simx builds the simulation executable (by default called simxsim):

-o filename

The output option allows the user to set a filename of the generated simulation executable. By default, the simulation executable is named simxsim.exe.

--scd_cleanup_objs

This performs a clean up of the generated files from a simulation build. This is primarily useful for reclaiming space when finished working with a simulation.

--scd_nosim

Generate the simulation executable (simxsim), but does not start the simulation. The simulation can later be started by running simxsim from the command line.

--scd_jobs=n

Compile simulation executable with n jobs. Default value is 0 which sets the number of jobs to the number of processor cores. For example, on a dual core both processors will be used by default to compile the simulation executable as quickly as possible. To reduce the load on a dual core machine, n could be set to 1 to keep one processor free.

--scd_usemake

Use make as the build tool. By default, simx uses Scons as the build tool as it will typically reduce the amount of files that need to be recompiled after edits of the HDL source code.

5.7 Simx Debug and Logging Options

These options control debugging and logging capabilities available during simulation. By default, many debugging capabilities are disabled when running from the command line simulator to ensure maximum runtime performance. When simx is run from the BugHunter graphical interface, many of these options are enabled to allow features such as single-step debugging and watching net values.

+access [+] [-] access_specification

Sets the visibility access for all objects such as nets and variables in the simulation for PLI/VPI (also required when using the debugger). The **access_specification** options are **r** (read access), **w** (write access), and **c** (connectivity access). Use the **plus** sign to turn on the specified access. Use the **minus** sign to turn off the specified access. If no plus or minus sign is used, + is the default. By default, objects do not have read, write, or connectivity access, so, the default is +access -rwc. Objects that are given write access are also given read access. Objects that are given connectivity access are also given write access, and, therefore, read access. Examples:

- Read access only: **+access +r**
- Write access: **+access +w**
- Read/Write access: **access +r+w**
- Read/Write/Connectivity access: **+access +r+w+c**
- You can also use multiple +access options: **+access +r +access -w**

+afile+accessfilename

Use the specified file, *accessfilename*, to set the visibility access for particular instances or portions of a design. You can also use the +access option to specify global visibility access for all objects in the design.

+append_log

Append log information from multiple runs of simx into one log file. Use this option if you are going to run simx multiple times and you want all the log information in one log file. If you do not use this option, the log file is overwritten each time you run simx. If you use both +append_log and +nolog on the command line, +nolog overrides +append_log.

+linedebug

Enable line debugging capabilities (for example, single stepping with a debugger).

+nostdout

Turn off output to the screen (terminal).

+scd_dbgsymbols

Generates symbolic information for all HDL objects from simulated design.

+scd_msg_enable+msg_level=0|1

Enables or disables printing messages at the specified message level. The allowed msg_levels are: +failure, +error, +warning, +note, +diagnostic.

5.8 Simx Specify block and SDF Timing Options

Below is a short summary of the timing-related command line options. For a full description of these options, please consult the Specify blocks chaptor of the Verilog 2001 LRM.

+epulse_ondetect

Enables On-Detect filtering of error pulses. This option extends the e state back to the edge of the event that caused the pulse to occur. See section 6.10 for the differences between on-detect and on-event pulse filtering.

+epulse_onevent

Enables On-Event filtering of error pulses. See section 6.10 for the differences between on-detect and on-event pulse filtering.

+epulse_neg

Filter canceled events (negative pulses) to the e state. This option makes canceled events visible. Using this option overrides any showcancelled and noshowcancelled settings in specify blocks.

+epulse_noneg

Do not filter canceled events (negative pulses) to the e state. Using this option overrides any showcancelled and noshowcancelled settings in specify blocks.

+notimingcheck

Do not execute timing checks

+notchkmsg

Do not display timing check warning messages.

+no_notifier

Ignore notifiers in timing checks.

+nospecify

Ignores timing checks, path delays, and $sdf_annotate calls.

+pulse_e error_percent

Set the percentage of delay for the pulse error limit for both module paths and interconnect. If the -pulse_int_e option is also used, this option applies only to module paths.

+pulse_int_e error_percent

Sets the percentage of delay for the pulse error limit for interconnects only.

+pulse_int_r reject_percent

Sets the percentage of delay for the pulse reject limit for interconnect only.

+pulse_r reject_percent

Set the percentage of delay for the pulse reject limit for both module paths and interconnect. If the +pulse_int_r option is also used, this option applies only to module paths.

+pathpulse

Enable PATHPULSE$ declarations. These declarations set the module path pulse control on a specific module or on specific paths within modules.

5.9 Simx Miscellaneous Options

This section contains miscellaneous comand line optins that do not fit into any of the previous categories.

+define+macroname

Define a macro name with a blank value for use in conditional compilation. For example: +define+mymacro

+define+macroname[=macrovalue]

Define a macro name as a string. For example: +define+mymacro=1

+loadpli1=pli_shared_lib_name:boot_function_name[,boot_function_name ...]

Dynamically load the specified PLI application and optionally specify a boot_function that will execute when the simulation is started. The argument to this option is the name or full path of the shared library that contains the PLI application followed by the name of the function or functions that registers the new system tasks. This function, called the boot function, is part of the PLI application, and is defined in the shared library. Any number of applications can be loaded in the same statement by separating the names of the bootstrap functions witha comma. No spaces are allowed in the argument. The file extension of the shared library is optional.

As an example, to load the shared library syncadverilogx.dll, execute the boot function register_default_tasks, then execute the boot function register_syncad_tasks, use the following option to simx or simxsim:

```
+loadpli1=syncadverilogx.dll:register_default_tasks,register_syncad_tasks
```

+loadvpi=vpi_shared_lib_name

Dynamically load the specified VPI application.

+tcl+filename

Read TCL simulator control commands from a file.

+version

Displays simulator version number.

5.10 Simx On-Event and On-Detect Pulse Filtering

Simx supports two methods of pulse filtering: On-Event and On-Detect. On-event filters pulses so that transitions to and from X occur after the delay for the originally scheduled transition and the new output state respectively. On-Detect is more conservative; it filters pulses so that the transition to the X state occurs immediately on detection of the pulse error. This X state then remains until the originally calculated delay for the new output state.

The On-Detect method allows more pessimism when filtering pulses to the X state, producing a longer X region. On-Detect filtering allows for a better understanding of the outputs caused by two or more changing inputs that result in output scheduling conflicts, but has more impact on simulation speed and yields more pessimistic results.

5.11 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 **verilog.log** file type the comma (**,**) character.
- To ***display the current code line*** execution, type the colon (**:**) character.
- To ***terminate*** the simulation, type the **$finish;** command, or press **<Ctrl>+C**.

Chapter 6: VeriLogger SDF Support

In the initial stages of your design you will be performing "functional" simulations to ensure 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 describe how to include the SDF for these tests in VeriLogger. 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.

6.1: 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 model of the FPGA. To include the timing from this file into your design, add an `$sdf_annotate` command in the FPGA module whose timing is to be modified. Include the bolded lines 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: 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 whatever filename your tool generated. The file extension `.sdf` should be used.)

Chapter 7: VeriLogger Pro Notes (vlogcmd)

VeriLogger Pro is an interpreted simulator. When vlogcmd 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.

7.1 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.

7.2 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.

7.3 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 (=>), and the other is the full case (*>). 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.

7.4: 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 connected 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.

7.5 Implementation Differences from Verilog-XL

This section describes the differences between the way VeriLogger works and the way Verilog-XL works. 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 both VeriLogger 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 continuous 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.

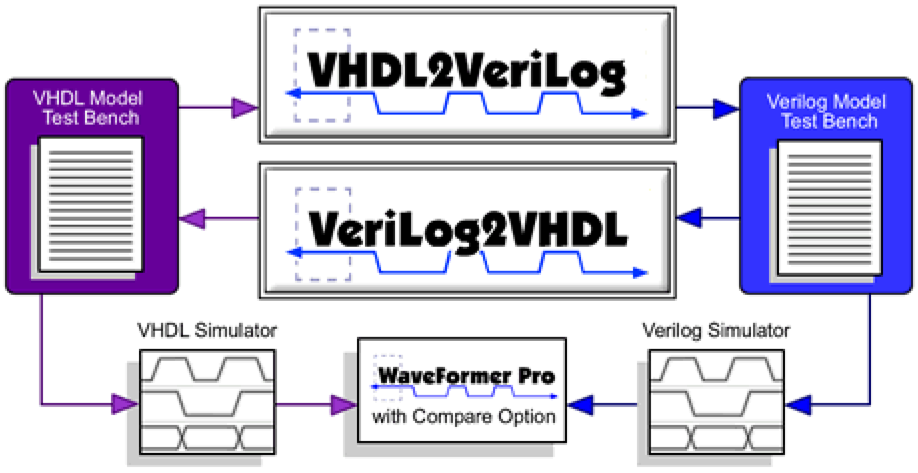
The $monitor System Task

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 changes, 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);
```

Chapter 8: VHDL <=> Verilog Translation

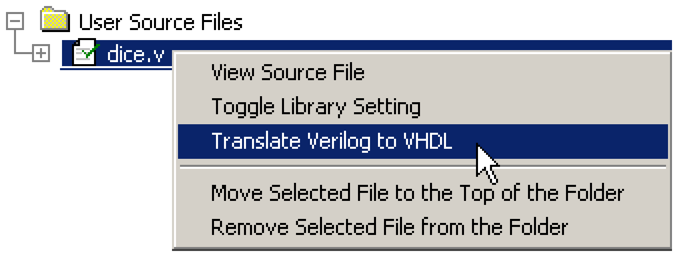
BugHunter Pro can also act as a graphical interface for SynaptiCAD's V2V code translators. To use the features in this chapter you will need a license for Verilog2VHDL or VHDL2Verilog in addition to your BugHunter Pro license. The translators can also be run from the command line, but the graphical interface makes it a lot easier to setup the options and see the results.



8.1: Setup Project for Translation

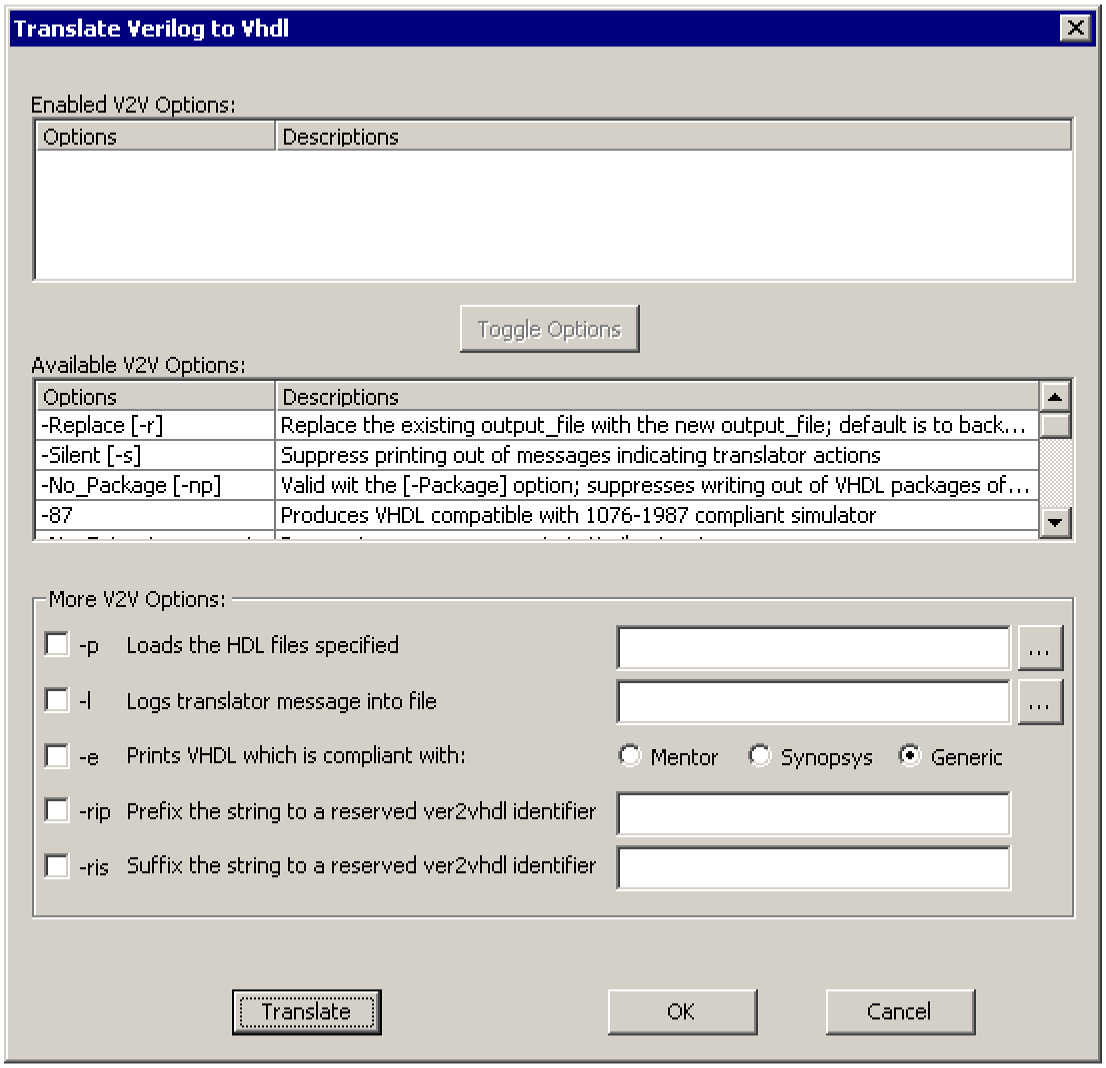
BugHunter Pro stores the translation options and list of files using the Project *.hpj file. Before starting a translation create a project and add in the files to be translated.

- Open a project or create a new one using the **Project > New Project** menu option as discussed in Step 3: Create a Project.
- Add source code files to the project by right clicking on the ***User Source Files*** folder in the Project Window and choosing one of the **FIles to Source File Folder** menus to open a *file* dialog as discussed in Step 4: Add Source Files to the Project.
- Set the translation options by choosing either **Project > Translate Verilog to VHDL** or **Project > Translate VHDL to Verilog** depending on which direction you plan to translate. This opens the translate options dialog as described in the next two sections. If you press the **Translate** button in this dialog, all the HDL source files in the project will be translated.
- To translate a single source file in your project, right click on one of the source files and choose a **Translate** function from the context menu. This will translate the file with the options that were set in the previous step above.


User Source Files
dice.v
View Source File
Toggle Library Setting
Translate Verilog to VHDL
Move Selected File to the Top of the Folder
Remove Selected File from the Folder

8.2: Verilog to VHDL

Verilog2VHDL provides several options that allow you to tailor the operation of the tool. To enable an option, select it in the **Available V2V Options** list and then click the button in the middle to move it to the **Enabled V2V Options** list at the top of the screen.



Enabled V2V Options:

Options	Descriptions

Available V2V Options:

Options	Descriptions
-Replace [-r]	Replace the existing output_file with the new output_file; default is to back...
-Silent [-s]	Suppress printing out of messages indicating translator actions
-No_Package [-np]	Valid wit the [-Package] option; suppresses writing out of VHDL packages of...
-87	Produces VHDL compatible with 1076-1987 compliant simulator

More V2V Options:

- [ ] -p Loads the HDL files specified
- [ ] -l Logs translator message into file
- [ ] -e Prints VHDL which is compliant with: Mentor Synopsys [✓ Generic]
- [ ] -rip Prefix the string to a reserved ver2vhdl identifier
- [ ] -ris Suffix the string to a reserved ver2vhdl identifier

If you are using the command line version of the translator you may list the commands after listing the files:

```
     v2v input_file [output_file] [options] [-Help] [-Usage] [-Version]
```

where options can be a combination of:

[-Replace] [-Silent] [-No_Package] [-No_Extract_comments]

[-Environment {Mentor|Synopsys|Generic}] [-Package {HDL files}]

[-Log {logfile_name}] [-87|-93] [-No_Component_check]

[-Component_check] [-SYNTH] [-Map_Regs_to_Variables]

[-No_Synth] [-No_Zero_wait] [-Make_Defines_Constants] [-Make_Parameters_Constants]

[-Reserved_Identifier_Prefix {prefix string}] [-Reserved_Identifier_Suffix {suffix string}]

[-Preserve_Order] [-Verilog_PreProcessing] [-Architecture_Name] [-No_header]

Command Summary

-Replace (-r):

Replace the existing output_file; default is to backup the output_file to <output_file>.old

-Silent (-s):

Supress printing out of messages indicating translator actions

-No_Package (-np):

Valid with the [-Package] option; suppresses writing out of VHDL packages of Verilog files supplied with [-Package] option; default is to create VHDL package with filename <pkgfilename_without_extension>_pack.hdl

-Environment {Mentor|Synopsys|Generic} (-e {m|s|g}):

Prints VHDL which is compliant with specified option, default is Generic

-Package {HDL files} (-p {HDL files}):

Loads the HDL files specified with this option into database

-Log {logfile_name} (-l {logfile name}):

Logs translator messages into file <logfile_name>; default log file is 'v2v.log'

-87:

Produces VHDL compatible with 1076-1987 compliant simulator

-No_Extract_comments (-ne):

Does not preserve comments in Verilog input

-No_Component_check (-ncc):

Does not look for module declarations for modules instantiated in instantiations. This is useful for translating designs which use large libraries

-SYNTH (-synth):

Produces synthesizable VHDL

-Map_Regs_to_Variables (-mrv):

Produces synthesizable VHDL with procedural assignments being mapped to variable assignments

-No_Zero_wait (-nz):

Does not print 'WAIT FOR 0 ns;' overridden by -synth option

-Make_Defines_Constants (-mdc):

Makes all `defines encountered in the input file constants in the Architecture; default is to create Generics in Entity

-Reserved_Identifier_Prefix <prefix string> (-rip <prefix string>):

Prefixes specified string to a reserved verilog2vhdl identifier. (See User's Manual, Chapter 3) for VHDL compliance

-Reserved_Identifier_Suffix <suffix string> (-ris <suffix string>):

Suffixes specified string to a reserved verilog2vhdl identifier. (See User's Manual, Chapter 3) for VHDL compliance

-Preserve_Order (-po):

When printing VHDL, retain the order of concurrent constructs in the Verilog input; default is to format the output VHDL

-Verilog_PreProcessing (-vpp):

Preprocess the Verilog files before translation them so that the Verilog compile directives can be elaborated and therefore supported

-Architecture_Name (-an):

Set the name of the generated VHDL architecture. The default name is "VeriArch"

-No_header [-nh]

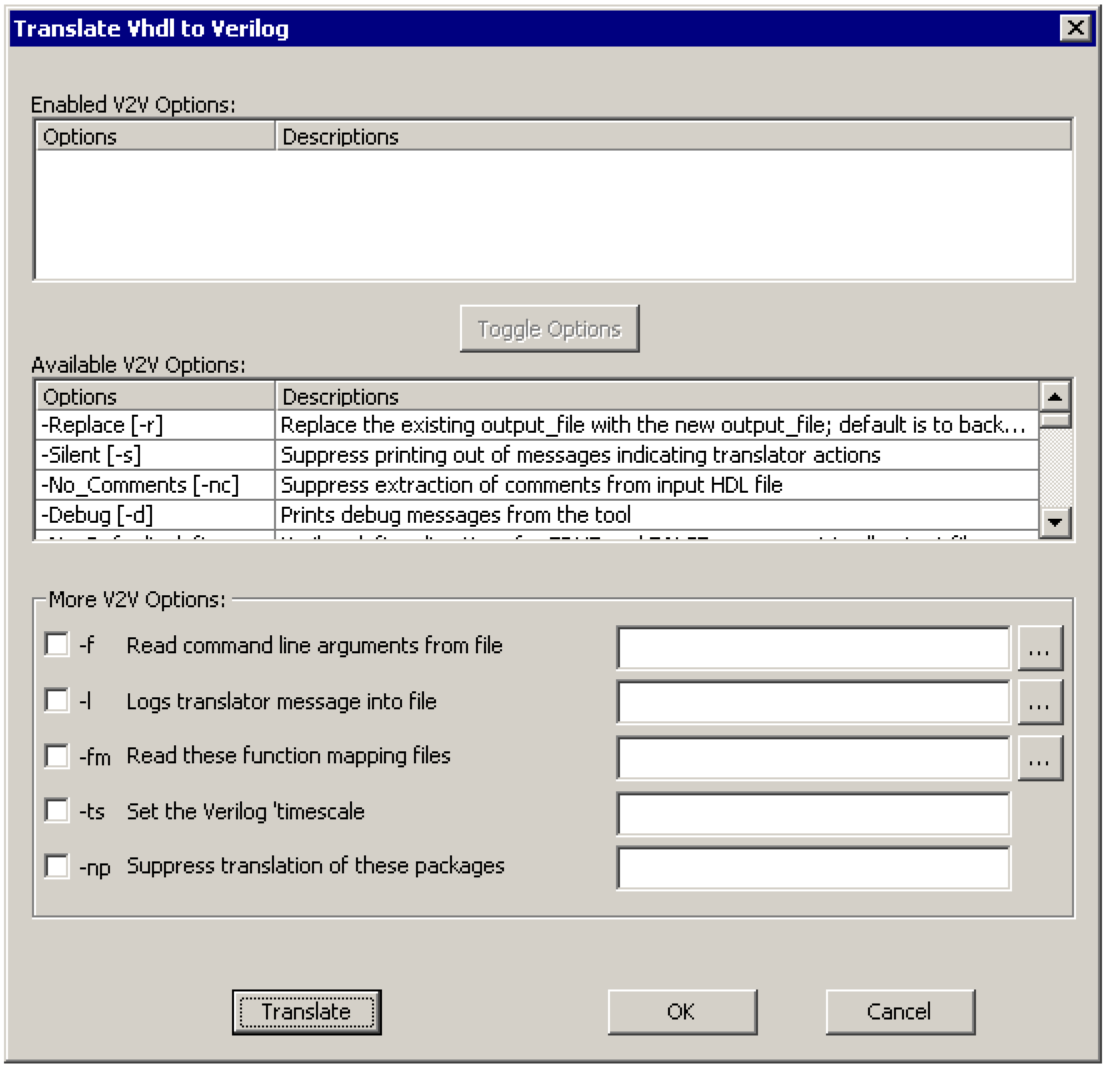
Do not print ASC header in the output file

-No_Verilog_PreProcessing [-novpp]

Disable the Verilog preprocessor (disables compiler directives) By default, the preprocessor is enabled.

8.3: VHDL to Verilog

VHDL2verilog provides several options that allow you to tailor the operation of the tool. To enable an option, select it in the **Available V2V Options** list and then click the button in the middle to move it to the **Enabled V2V Options** list at the top of the screen.



Options	Descriptions

Options	Descriptions
-Replace [-r]	Replace the existing output_file with the new output_file; default is to back...
-Silent [-s]	Suppress printing out of messages indicating translator actions
-No_Comments [-nc]	Suppress extraction of comments from input HDL file
-Debug [-d]	Prints debug messages from the tool

☐ -f Read command line arguments from file

☐ -l Logs translator message into file

☐ -fm Read these function mapping files

☐ -ts Set the Verilog 'timescale

☐ -np Suppress translation of these packages

If you are using the command line version of the translator you may list the commands after listing the files:

```
    vhdl2v input_file [output_file] [options] [-Help] [-Usage] [-Version]
```

where option can be one or all of

[-File {file_name}] [-Replace] [-Silent] [-No_Comments] [-Debug]

[-No_Default_defines] [-No_Package_translation {package_file_names}]

[No_Component_Check] [-Include_Package_files] [-Function_Map {files}]

[-Ignore_Subprogram_Calls] [-Translate_Subprogram_Bodies]

[Preserve_Generate] [Support_Multi-dimensional array] [Ignore_Integer_Range]

[Support_Directives] [-Force_Lower_Case] [-Time_Scale {time}]

[-Log {logfile_name}] [-SYNTHesis] [-87] [-No_header] [-No_timescale]

Command Summary

-File {file_name} [-f {file_name}]:

Read command line arguments from the specified file

-Replace [-r]:

Replace the existing output_file with the new output_file ; default is to backup the output_file to <output_file_name>.old

-Silent [-s]:

Suppress printing out of messages indicating translator actions

-No_Comments [-nc]:

Supress extraction of comments from input HDL file

-Debug [-d]:

Prints debug messages from the tool

-No_Default_defines [-nd]:

Verilog define directives for TRUE and FALSE are present in all output files; use of this switch suppresses printing of default defines

-No_Package_translation {package_names} [-np {package_names}]:

Packages with <package_name> or having <package_name> as prefix are not translated; this can be used to suppress translation of specific packages

-No_Component_Check [-ncc]:

Translation will proceed even if some component definiations or entity declarations are missing; this can be used to force translation of individual components without all its components; the translation result might not compile because of the absence of its conpoment;

-Include_Package_files [-ip]:

Use of this switch will direct the translator to write the translation of packages into files '<package_name>_package.verilog' and '<package_name>_modules.verilog'; default is to include them in the output file

-Function_Map {files} [-fm {files}]:

Read the function mapping files specified for translation

-Ignore_Subprogram_Calls [-ssc]:

Do not translate the headers of function and procedure calls

-Translate_Subprogram_Bodies [-tsb]:

Translate the bodies of function and procedures into a separate file (override -isc switch). Translation is done on unconditional basis: Output is not guaranteed to work!

-Preserve_Generate [-pg]:

Preserve generate statement. Generate Statements will not be elaborated if the concurrent body contains only concurrent assignments

-Support_Multi-dimensional_array [-sm]:

Bit-wise access of multi-dimensional array will be translated. Additional signals or subprograms may be created for translating multi-dimensional arrays

-Ignore_Integer_Range [-iir]:

Ignore the integer range and translate VHDL integers to 32 bit Verilog integers

-Support_Directives [-sdi]:

Support translation directives, such as translate_off/translate_on

-Force_Lower_Case [-flc]:

Changes all identifiers to lower case

-Time_Scale {timescale} [-ts {timescale}]:

Set the Verilog 'timescale

-Log {logfile_name} [-l {logfile_name}]:

Logs translator messages into file <logfile_name>; default log file is 'vhdl2v.log'

-SYNTHesis [-synth]:

Produces synthesizable code; presently restricted to suppressing initial statements and using '==' Verilog operators instead of '===' operators

-87 [-87]:

Disable support for VHDL-93; enable VHDL-87 support instead

-No_header [-nh]

Do not print ASC header in the output file.

-No_timescale [-nt]

Do not print timescale directive in the output file. Default is OFF(timescale is printed in the output file).

-Save_Parenthesis [-sp]

Inserts parenthesis in expressions to prevent confusions about precedence of operators

-Blocking [-bl]

Use blocking assignments in all combinational procedural blocks

-Force_If_Generate [-fig]

Force translation of 'if_generate' statement, regardless the 'if' conditions. The result might not be equivalent to the put when this option is set. Manual editing of the translation result may be necessary."

-VHDL [-VHDL]

Prints out the VHDL read into filename ,inputfile>, vhdl only

-XML[-XML]

Prints out the XML documents of the input files.

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("myfile.btim"); . If you are running from Bug Hunter's graphical environment, you can execute these system tasks from the console window 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 theses system tasks (the graphical environment does this automatically when it launches the simulator).

	VeriLogger vlogcmd	VerilogXL	ModelSim	NC	ActiveHdl (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 ***1**	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_removetimebreak	-	Yes	Yes	Yes	-
db_enabletimebreak	-	Yes	Yes	Yes	-
db_disabletimebreak	-	Yes	Yes	Yes	-
db_getbasictype	Yes	Yes	-	Yes ***2**	-
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 supported for NC Verilog but not NC VHDL.

Command Line Simulator Support

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( )
```

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
```

where filename is the name of the btim file to receive the signal data

Related Functions:

```
btim_AddDumpSignal, btim_closedumpfile
```

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
```

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 )
```

Arguments:

```
char* signalname
```

relative or full signal name are accepted. The function outputs an error if the signal doesn't exist.

Related Functions:

```
btim_dumpfile, btim_AddDumpSignal
```

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
```

db_printinteractivescope

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
```

db_finish

Finishes the current simulation.

Syntax:

```
$db_finish()
```

db_addtimebreak

Adds a break point at the absolute time specified. The time should be specified in the internal simulator time precision that can be retrieved by calling db_printinternaltimeprecision.

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_printinternaltimeprecision, db_removetimebreak,
db_enabletimebreak, db_disabletimebreak
```

db_removetimebreak

Removes the time break point that was added previously with db_addtimebreak using given id of the break point.

Syntax:

```
$btim_removetimebreak( id )
```

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

Arguments:

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

Related Functions:

```
db_addtimebreak, db_enabletimebreak, db_disabletimebreak
```

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_removetimebreak, db_disabletimebreak
```

db_disabletimebreak

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

Syntax:

```
$db_disabletimebreak( id )
```

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

Arguments:

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

Related Functions:

```
db_addtimebreak, db_removetimebreak, db_enabletimebreak
```

db_getbasictype

Prints the basic type of the given object.

Syntax:

```
$db_getbasictype( 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 : basictype"
```

Arguments:

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

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 db_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
```

db_printinternaltimeprecision

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
```

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

You can use SynaptiCAD's btim commands by launching your cmd line simulator with the appropriate options (the BugHunter/VeriLogger GUI is not required). This is particularly useful for directly dumping BTIM waveform files instead of dumping VCD files, as the simulation will run much faster and the resulting files are much smaller.

Below are the appropriate syncad PLI libraries for each simulator and the command line options to load the library for that simulator. The PLI libraries are located in the Synapticad\bin directory.

VHDL

- **ActiveVHDL:** vsim -callbacks -pli syncadactivevhdl
- **ModelSim SE** (PE and XE not supported): vsim.exe -c -foreign "initForeign syncadmodelsimvhdl"
- **Cadence NC VHDL:** ncsim.exe -LOADCFC syncadncvhdl:register_syncad_tasks

Verilog

- **ActiveVerilog:** vlog.exe -pli syncadactiveverilog
- **ModelSim Verilog:** vsim.exe -c -pli syncadmodelsimverilog
- **Cadence NC Verilog**: ncverilog.exe +access +rwc +loadvpi=syncadncverilog:register_syncad_tasks
- **VeriLogger Extreme (simx):** simxsim.exe +access +rwc +loadvpi=syncadncverilog:register_syncad_tasks

    - Note: simxsim is the simulation-specific exe built by simx.

- **VeriLogger (vlogcmd):** vlogcmd.exe +loadpli1=syncadvlogcmd.dll:register_default_tasks,register_syncad_tasks
- **VCS:** Btim PLI not supported

Index