

**TestBencher Pro**

**User's Manual**

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

## **TestBench Pro Manual (rev 10.0) copyright 1994-2005 SynaptiCAD**

### Trademarks

- Timing Diagrammer Pro, WaveFormer Pro, TestBench Pro, VeriLogger Pro, DataSheet Pro, BugHunter Pro, Reactive TestBench Generation Option and SynaptiCAD are trademarks of SynaptiCAD Inc.
- VERA, OpenVera, VCS, and VCSi are trademarks of Synopsys, Inc.
- NC Verilog, NC VHDL, and Verilog-XL are trademarks of Cadence Design Systems, Inc.
- Pod-A-Lyzer is a trademark of Boulder Creek Engineering.
- PeakVHDL and PeakFPGA are trademarks of Accolade Design Automation Inc.
- V-System and ModelSim are trademarks of Model Technology Incorporated.
- Viewlogic, Workview, and Viewsim are registered trademarks of Viewlogic Inc.
- HP and Agilent are trademarks of Hewlett Packard.
- Tektronix copyright Tektronix, Inc.
- PI-2005 and PI-Pat are trademarks of Pulse Instruments.
- Timing Designer and Chronology are registered trademarks of Chronology Corp.
- DesignWorks is a trademark of Capilano Computing.
- Mentor and QuickSim II are registered trademarks of Mentor Graphics Inc.
- OrCAD is a registered trademark of OrCAD.
- PSpice is a registered trademark of MicroSim.
- Windows, Windows NT, and Windows 95/98/2000 are registered trademarks of Microsoft.

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

Information in this documentation is subject to change without notice and does not represent a commitment on the part of SynaptiCAD. Not all functions listed in manual apply to Timing Diagrammer Pro, WaveFormer Pro, DataSheet Pro, or VeriLogger Pro. The software and associated documentation is provided under a license agreement and is the property of SynaptiCAD. Copying the software in violation of Federal Copyright Law is a criminal offense. Violators will be prosecuted to the full extent of the law.

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

For latest product information and updates contact SynaptiCAD at:

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

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

phone: (540)953-3390

## Table of Contents

<b>Table of Contents .....</b>	<b>3</b>
<b>Introduction .....</b>	<b>7</b>
<b>Chapter 1: TestBencher Pro Design Flow.....</b>	<b>9</b>
Step 1: Create a New Project .....	9
Step 2: Add the MUT to the Project .....	10
Step 3: Extract Port Information .....	11
Step 4: Create a Timing Transaction .....	11
Step 5: Define Sequencer Process .....	12
Step 6: Generate the Test Bench .....	13
Step 7: Setting Up Simulators .....	14
Step 8: Simulate Test Bench .....	15
<b>Chapter 2: Projects and Component Generation .....</b>	<b>17</b>
2.1 Creating, Opening and Saving Projects .....	17
2.2 The Project Window .....	18
2.3 Sub-Projects .....	19
2.4 Component Instances of Sub-Projects .....	20
2.5 Component and Component Instance Generation Properties .....	21
2.6 Signals and Ports for Components .....	23
2.7 Golden Reference Models .....	24
2.8 Libraries and Use Clauses (VHDL only) .....	25
<b>Chapter 3: Transaction Overview .....</b>	<b>27</b>
3.1 Template Diagram and New Transactions .....	27
3.2 Extracting MUT Ports into a Timing Diagram .....	28
3.3 Transaction Level Variables .....	29
3.4 Diagram-Level Class Methods .....	30
3.5 Transaction Architecture .....	30
3.6 Diagram Properties .....	33
3.7 Diagram Settings Dialog - Overview .....	34
3.8 Diagram Settings Dialog - General Tab .....	36
3.9 Diagram Settings Dialog - Language Specific Tabs .....	38
<b>Chapter 4: Transaction Waveforms and Signals .....</b>	<b>41</b>
4.1 Drawing Transactions for TestBencher .....	41
4.2 Drawing Waveforms and Bi-Directional Signals .....	42
4.3 Driving Waveform States with Variables .....	42
4.4 Driving Conditional State Values .....	43
4.5 Adding Signals .....	44
4.6 Temporal Expressions for TestBencher .....	45
4.7 Controlling the Triggering Order of Parameters .....	45
4.8 Sensitive Edges .....	46

<b>Chapter 5: Transaction Delays, Setups, and Holds.....</b>	<b>49</b>
5.1 Adding and Editing Parameters .....	49
5.2 Delays .....	50
5.3 Resolving Multiple Delays .....	51
5.4 Setups and Holds .....	52
5.5 Creating Continuous Setups and Holds .....	52
<b>Chapter 6: Transaction Samples .....</b>	<b>55</b>
6.1 Adding a New Sample .....	55
6.2 Sample Condition and Actions .....	56
6.3 Interpreting Sample Conditions and Blocking Points .....	58
6.4 Samples Triggering a Delayed Transition or Another Sample .....	59
6.5 Using Sample Variables .....	60
6.6 Storing Sample Values in User Defined Variables .....	61
<b>Chapter 7: Transaction Markers.....</b>	<b>63</b>
7.1 Adding a Marker to a Diagram .....	63
7.2 End Diagram Markers .....	64
7.3 Pause Simulation Marker (Verilog Only) .....	64
7.4 Wait Until Marker .....	65
7.5 Loop Markers .....	65
7.6 HDL Code Markers .....	66
7.7 Semaphore Markers .....	67
7.8 Pipeline Boundary Markers .....	67
7.9 Documentation and Time Break Markers .....	68
<b>Chapter 8: Classes and Variables .....</b>	<b>69</b>
8.1 Class Libraries .....	69
8.2 Classes .....	71
8.3 Variables .....	72
8.4 Variable and Class Field Properties .....	73
8.5: Language Independent Types .....	74
8.6 Data Packing .....	76
8.7 Class Methods .....	77
8.8 Constrained Random Number Generation .....	79
8.9 File Input and Ouput Variables .....	81
8.10 Importing Fields from a Template File .....	82
8.11 Semaphores .....	82
<b>Chapter 9: Project Component and Transaction Sequencer .....</b>	<b>85</b>
9.1 Transaction Calls .....	85
9.2 Writing Code in the Template File .....	86
9.3 Transaction Manager and Test Reader .....	87
9.4 Transaction Generator .....	89
9.5 Transaction Monitor .....	91
9.6 Changing a Project's Template File .....	91

<b>Chapter 10: Generation and Simulation .....</b>	<b>93</b>
10.1 Generate the Bus Functional Model .....	93
10.2 Simulator and Compiler Settings Dialog .....	93
10.3 Project Simulation Properties Dialog .....	94
10.4 Simulating the Bus Functional Model .....	97
10.5 Generating Command Files for Third Party Simulators .....	99
10.6 TestBencher Simulation Modes .....	99
<b>Chapter 11: Test Bench Techniques.....</b>	<b>101</b>
11.1 Master and Slave Transactions .....	101
11.2 Waiting for Signal Transitions .....	101
11.3 Burst Mode Transactions .....	102
11.4 Conditionally Moving Signal Edges (Sweep Tests) .....	103
11.5 Reading and Writing Serial Data .....	104
11.6 Testing a Counter Model .....	105
11.7 External Model Support .....	106
<b>Chapter 12: Language Specific Details .....</b>	<b>107</b>
12.1 Verilog .....	107
12.2 VHDL .....	108
12.3 TestBuilder .....	112
<b>Appendix A: Editor Commands .....</b>	<b>117</b>
<b>Appendix B: Supported Simulators.....</b>	<b>121</b>
<b>Appendix C: Language Independent Operators .....</b>	<b>123</b>
<b>Appendix D: License Agreement .....</b>	<b>133</b>
<b>TestBencher Pro: Basic Tutorial .....</b>	<b>135</b>
1) Create a Project .....	135
2) Create the Write Cycle Transaction Diagram .....	137
3) Create the Read Cycle Transaction Diagram .....	138
4) Create the Initialize Transaction Diagram .....	140
5) Modify the Sequencer Process .....	141
6) Generate Test Bench and Simulate .....	144
<b>Index .....</b>	<b>147</b>



# Introduction

TestBencher Pro provides designers with a graphical environment for rapidly generating system level test benches composed of cycle-based or time-based bus functional models. TestBencher Pro's graphical interface speeds up test bench development for both expert and novice users. TestBencher generates all of the low-level transaction code, verification code, sequence detection, error reporting and file I/O code. The graphical representation also enhances the ability of engineers to share data across projects, even though new engineers might not be familiar with the details of the test bench design.

## Learning to use TestBencher

The quickest way to learn how to use TestBencher Pro is to work through the tutorials and to read through *Chapter 1: TestBencher Pro Design Flow*. Advanced features such as constrained random number generation, loops in test benches, generating code from samples, and controlling execution using markers can be found in the later chapters. Some resources to use when learning TestBencher are:

- **On-line Help:** Both the TestBencher and the Timing Diagram Editor manuals are available and cross-reference each other.
- **On-line Tutorials:** TestBencher ships with several tutorials that demonstrate how to create different types of bus-functional models. There are also several timing diagram editor tutorials.
- **Test Bench Examples:** Several test bench examples ranging from simple test benches to PCI and ARM bus examples are located in the *\Examples* subdirectory of the installation directory.
- **Context Help:** Many of the dialogs that you will use contain a context help feature that will allow you to learn more about the specific controls within the dialog. The dialogs with this feature have a small button with a question mark **?** in the upper right hand corner of the dialog (next to the close window button). To use context help, click the question mark, and then click the item in the dialog about which you would like to see help.
- **Test Bench Techniques:** *Chapter 11* of the manual catalogs techniques for generating models with different types of functionality.

Additional information regarding the architecture and design flow of TestBencher Pro is available at the SynaptiCAD website ([www.syncad.com](http://www.syncad.com)).





# Chapter 1: TestBench Pro Design Flow

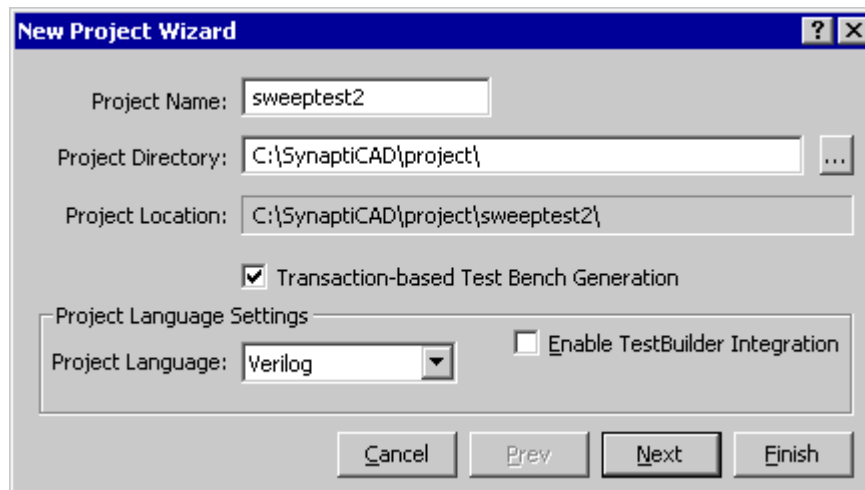
This chapter will cover the basic design flow for generating a bus-functional model using TestBench Pro. First you will create a new project file and add information about the model under test (MUT) files. Next you will create the timing diagrams that generate the reusable timing transactions. Then you will edit the top-level model and define the sequence for applying the transactions to the MUT. Finally you will simulate the test bench.

## Step 1: Create a New Project

Projects represent bus functional models (BFM) in TestBench. They hold all the information needed to generate the entire BFM including the transaction diagrams, top-level file, and the code generation settings. Projects can be included hierarchically in other projects. This allows TestBench to support multiple BFM component instantiation. Once a project has been completed, the entire bus functional model that it represents, or project component, can be instantiated in another project. *Chapter 2: Projects and Component Generation* has detailed information about creating projects, but essentially when you create a new project you set the name, language, and clocking options for the new BFM.

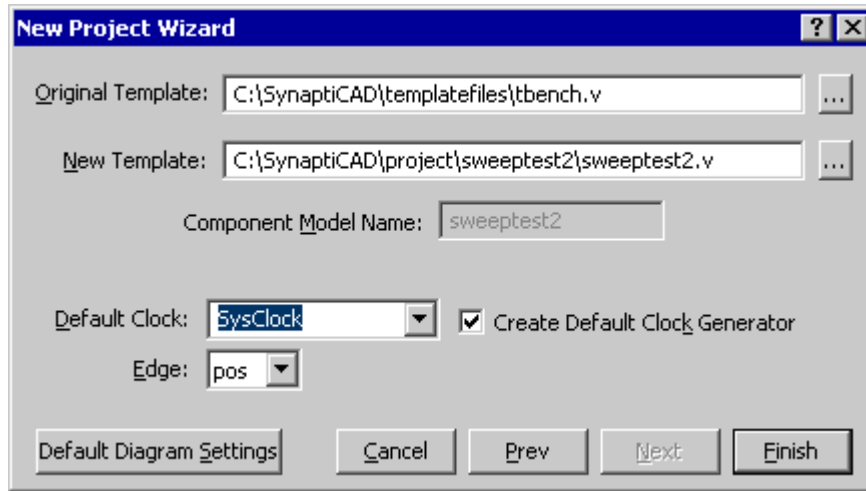
To create a project:

- Select the **Project > New Project** menu option. This will open the *New Project Wizard* dialog.

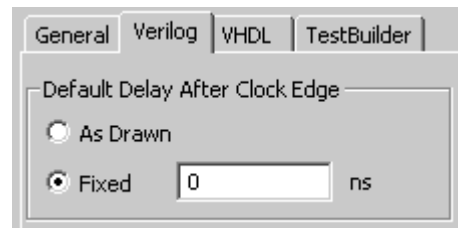


- The **Project Name** will be both the name of the project and the directory where the project is stored.
- The **Language** drop down list defines the generated language for the model. Certain features, such as valid signal types, are dependent on the generated language so this option affects the operation of TestBench. If you are using Verilog and want to enable constrained random number generation, check the **Enable TestBuilder Integration** checkbox.
- Check the **Transaction-based Test Bench Generation** to make TestBench generate bus-functional models instead of single diagram test benches.

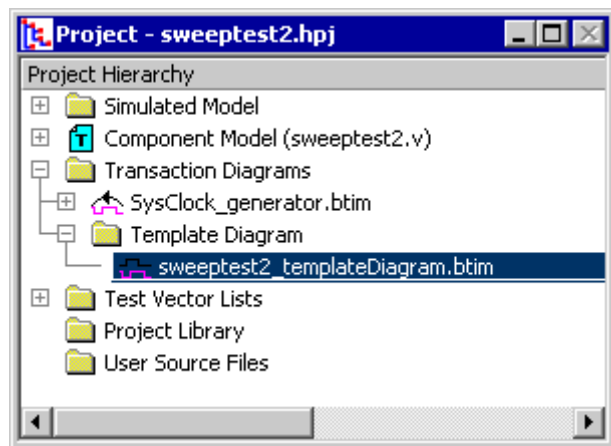
- Click the **Next** button to move to the next dialog.



- If your project is clocked, then type in the name of the clocking signal.
- Click the **Default Diagram Settings** button to edit the default diagram settings for new diagrams. By default, the **Include Delay Time** setting is not enabled for your design. See *Section 3.9: Diagram Settings Dialog - Language Specific Settings* for information about how this setting is used.
- Click the **Finish** button to close the *New Project Wizard* and create the new project.



Notice that TestBencher automatically opens the *Project* window (see *Section 2.2: The Project Window*) and populates it with the top-level template file (see *Chapter 9: Project Component Sequencer Files*) and template diagram (see *Section 3.1: Template Diagram and New Transactions*). The template diagram is the starting point for any new transactions that you add to the project. The *Project* window will be your main resource for navigating through the different parts of the bus functional model.



## Step 2: Add the MUT to the Project

The Model Under Test, MUT, files of the design should be added to the project. TestBencher will use these files to extract the signal and port information for use in the transaction diagrams and will use the file information to build make files for your simulator.

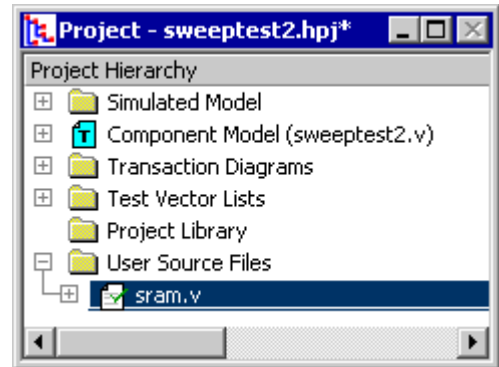
To add a source file to the project:

- Right-click in the *User Source Files* folder in the *Project* window to open the context menu, and select the **Add Files to User Source File Folder** menu option
- OR, choose the **Project > Add Files** menu option



- Both of these functions open a file dialog. Select the files that you would like to add to the project and click the **Open** button to close the dialog. All files necessary to compile and simulate the MUT should be added to the project because TestBencher parses the entire design.



Notice that the file names are listed in the *User Source Files* folder in the *Project* window. The source code can be viewed by double-clicking on the file name.

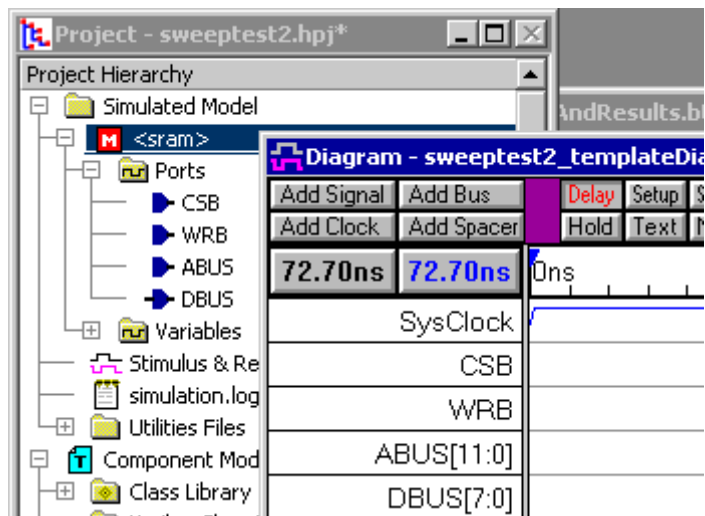


### Step 3: Extract Port Information

The next step is to build the MUT and extract the port information from it. TestBencher will parse the MUT and display the hierarchy of the design in the *Project* window. Also, the top-level ports or signals if there are no ports, will be inserted into current timing diagram.

To parse the source files and build the MUT:

- Double-click the diagram template file to open the timing diagram.
- Click the **Extract Ports From MUT** button  on the *Simulation Button Bar*. This will cause the source files to be parsed and the components to be built.
- Notice that the signal and port information has been added to the template diagram.  
Click the **Save** button  or choose the **File > Save Timing Diagram** menu item to save the template diagram.
- Notice that the *Simulated Model* folder in the *Project* window displays design hierarchy.



*Chapter 3: Transaction Overview* has more information about creating internal signals for the diagrams and manually editing the signal type, direction, and size.

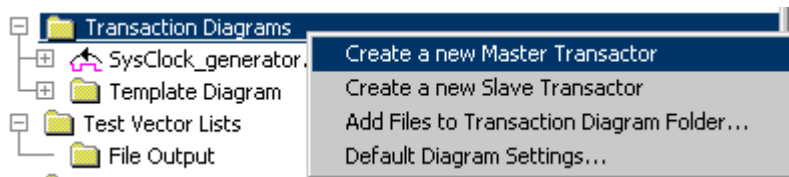
### Step 4: Create a Timing Transaction

A timing transaction is a timing diagram that represents a reusable interface specification of the bus-functional model that you are creating (e.g., read cycle, write cycle, interrupt cycle). Timing diagrams are created using the built-in timing diagram editor. The tutorials and *Chapters 3-7* describe how to draw the timing diagrams and control the generated code.

New timing diagrams that are created while this project is active will have the same properties and signals as the *Template Diagram*. This diagram is located in the *Template Diagram* folder in the *Project* window. Any signals that will be in all your timing diagrams (such as a global clock signal, or the ports for the MUT) should be in this diagram.

To draw a timing transaction:

- In the *Project* window, right-click on the *Transaction Diagrams* folder and choose either **Create a new Master Transactor** or **Create a new Slave Transactor** from the context menu. This opens a file dialog to create and save the diagram. After closing the dialog, a new diagram is created using the same properties and signals as the Template Diagram that you modified in *Step 3: Extract Port Information*. Slave diagrams run in a looping mode until they receive an abort call, and Master diagrams run once and stop.



- Create a timing diagram by sketching the waveforms using the *Timing Diagram Editor*. Optional components such as samples, markers, delays, variables, and class methods will be discussed in more detail later in this manual and are demonstrated in the tutorials. The optional components needed are determined by the needs of the test bench.
- Select the **File > Save Timing Diagram** menu option to save the timing diagram and generate the HDL code. Each time you save a timing diagram new code is generated for it.
- By default a timing diagram will generate Master Transaction code that will run once and then stop. To generate Slave Transaction code that will loop continuously you just need to change the diagram setting to slave: right click on the timing diagram name in *Project* window and choose **Diagram Settings** (see *Section 3.7: Diagram Settings Dialog Overview*) and then check the **Slave Transactor** radio button.

To view the generated HDL source code:

- Click the **Source Code** button to open an editor and view the code. Because the source code is generated for each timing diagram, it should not be edited. It is, however, useful to see how the low level code changes based on the constructs that are placed in the diagram.



At this point you can either design additional transactions, or you can continue with the next few steps and design the top-level test bench. By working with the top-level test bench early in the design you will be able to test individual transactions before constructing the entire bus-functional model.

## Step 5: Define Sequencer Process

The top-level template file represents the Component Model. The Component Model controls the execution sequence and monitors the status of each timing transaction in the project. It is also where the model under test is instantiated and connected to the test bench model.

Inside the top-level file is a Sequencer Process that controls the order and logic in which the timing transactions are applied to the model under test. Inside the sequencer process is the place that you will write the system level code to apply the transactions. In addition to the sequencer process, TestBencher can also generate an advanced Transaction Manager that can read transactions from files, randomly generate transactions, or accept transactions posted by the components in the project. *Section 9.6: Changing a Project Template File* describes the sequencer process, transaction manager, and the template file in detail.

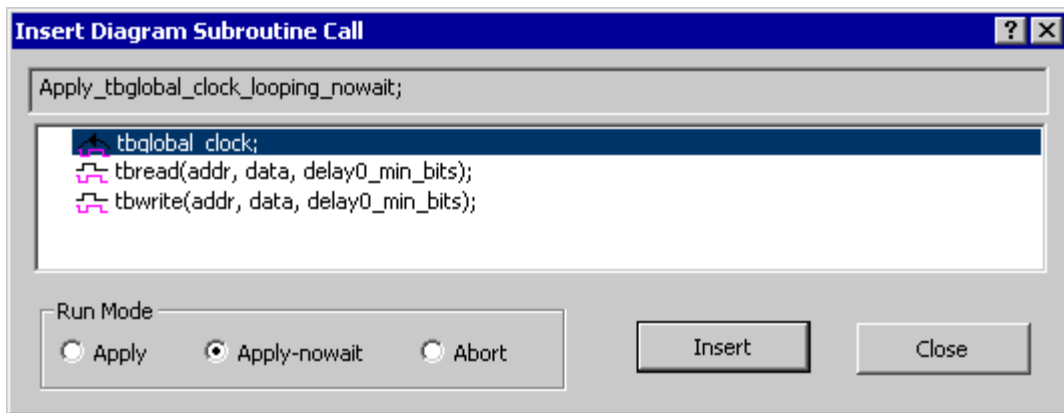
Use the *Insert Diagram Call* dialog to add timing diagram apply statements to the Component Model's Sequencer Process:

- Double-click on the *Component Model* folder in the *Project* window to open the template file.

- Scroll down in the template file until you find the Sequencer Process. A comment block in the code will help to locate this process. The comment will contain the following text:

```
//*****
// Transaction Sequencer - After this comment, define how to
//   apply transactions to the model under test using:
```

- Click in the *Editor* window just below this comment. Then right-click and select the **Insert Diagram Calls...** menu option from the context menu. This will open the *Insert Diagram Calls* dialog with a list of statements that represent each of the timing transactions that have been added to the project.



- Select a timing diagram name. The *Run Mode* radio buttons will default to **Apply** for Master transactions to run them in a blocking mode. For Slaves the default is **Apply-nowait** to run the transaction concurrently with subsequent apply calls.
- Choose a *Run Mode* radio button and press the **Insert** button.
- Notice that the Apply statement was inserted at the same line as your cursor. The *Insert Diagram Call* dialog is a modeless dialog it can remain open while you perform other actions. Inserting additional Apply statements causes those statements to be added on successive lines.
- If any of the applied transactions contain variables, then edit the Apply call to provide values for variable names. In the example Apply statement below, a value of hex 55 is assigned to addr.


```
// Apply_tbwrite(addr, data, delay0_min_bits)
Apply_tbwrite('h55, 'hee, $realtobits(delay0));
```

*Chapter 12: Language Specific Details* has more information about language specific features of editing parameter variables.

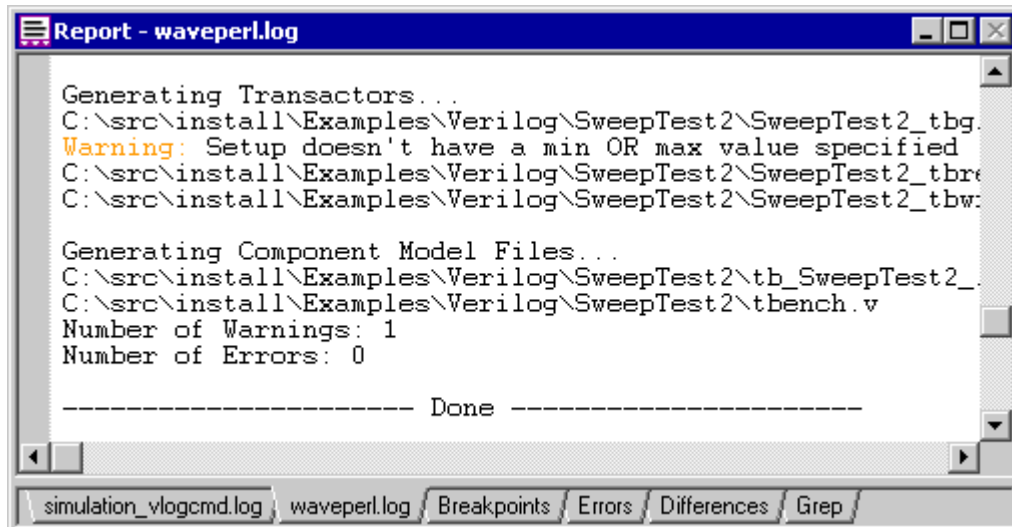
## Step 6: Generate the Test Bench

Once the Sequencer Process has been edited, the test bench is ready for generation. This step will expand a series of macros in the template file. Any code that is between the begin and end statements of a macro will be destroyed and re-generated. Any code outside of the macros, such the body of the Sequencer Process will be preserved. *Chapter 10: Generation and Simulation* describes the generation process.

To generate the test bench:

- Click the **Generate Test Bench** button  on the simulation button bar. This expands the macros in the template file.

- Check for generation errors by looking at the **waveperl.log** file in the *Report* window. If there are no errors then you are ready to simulate the BFM.



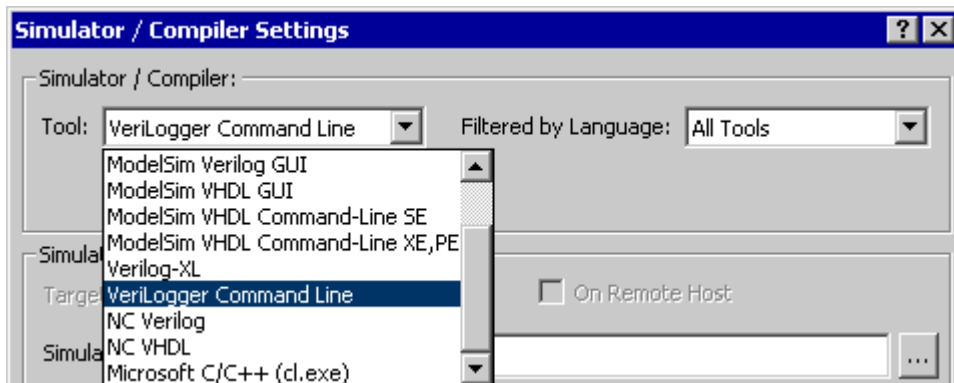
During the Generate Test Bench process several files are generated and they depend on the generation language (see *Chapter 12: Language Specific Details* for more information). All of the generated files are displayed in the *Project* window.


## Step 7: Setting Up Simulators

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

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

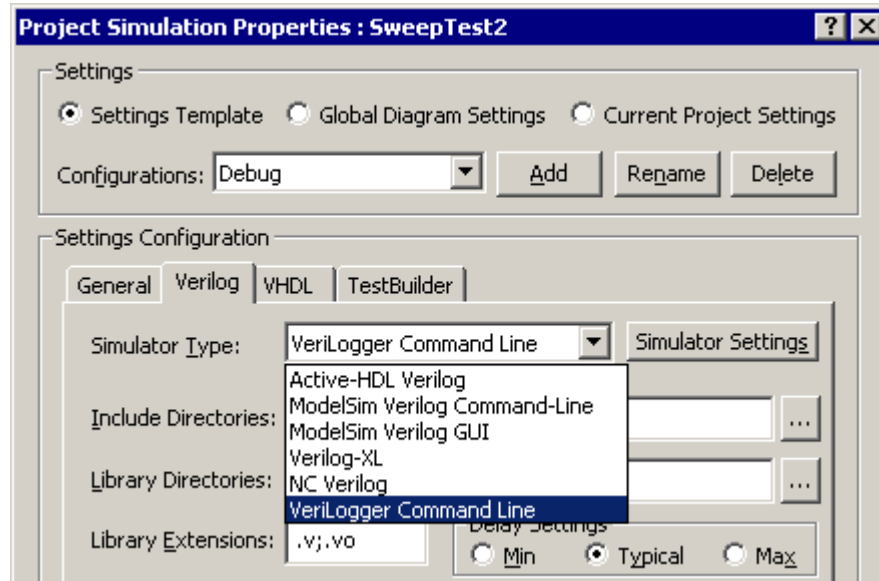
- Choose the **Options > Simulator / Compiler Settings** menu option to open a dialog of that name.



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

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

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





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

## Step 8: Simulate Test Bench

TestBench Pro ships with a basic version of BugHunter Pro a graphical debugger that can control external simulations. For comprehensive instructions on BugHunter read the BugHunter manual.

Start a simulation:

- Click the **Compile the Active Project**  button on the simulation button bar.
- In the *Report* window, check compile error in the simulation results file tabs. If there are no errors then continue.
- Either click the green **Run** button  on the simulation button bar or press the <F5> key.



View Simulation Results:

- When the simulation is complete the waveform results will be placed in the *Diagram* window. The simulation output will be displayed in the *Report* window.



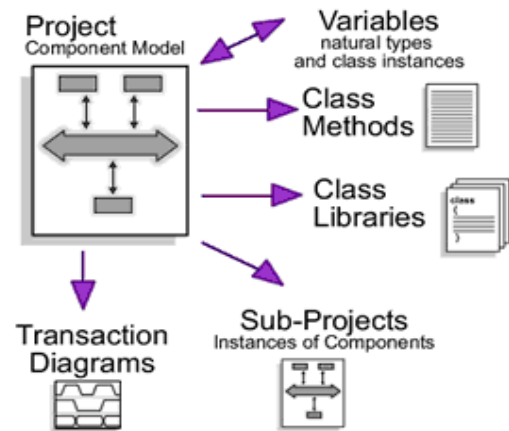


## Chapter 2: Projects and Component Generation

TestBench Pro uses a project file to represent and to control the generation of a bus-functional model (BFM) component. The information in the project file is displayed in the *Project* window. Context sensitive menus provide a list of actions that can be performed for the elements in the project tree.

Multiple test bench components can be made by including a project inside of another project, and then instantiating the sub-project. This allows complex test benches to be developed and verified in an incremental manner. This method also supports multiple port testing.

TestBench can also generate a C++, VHDL or Verilog Golden Reference model that runs in parallel with the VHDL or Verilog Model Under Test. During simulation transactions are sent to both the MUT and the reference model. The results from the two models are then compared.



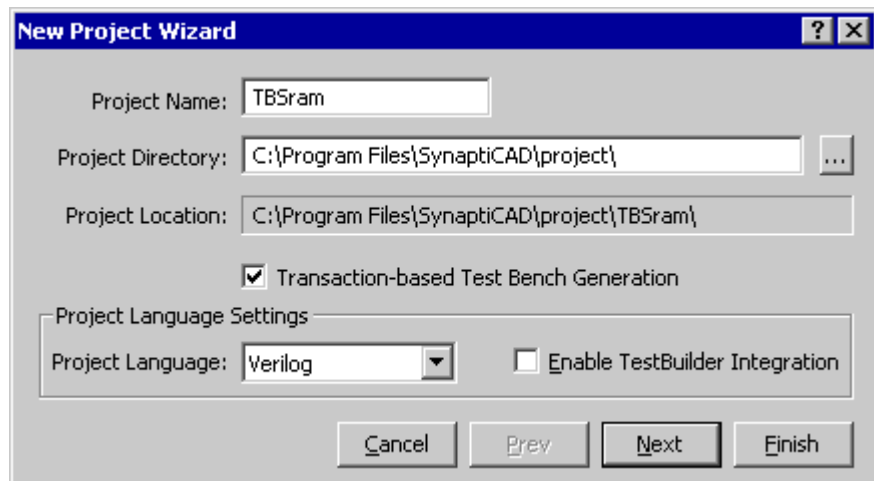
### 2.1 Creating, Opening and Saving Projects

Projects are created, opened, saved, and closed using the Project menu options.

- Select the **Project** menu option and choose one of the project submenus: **New Project**, **Open Project**, **Save Project**, or **Close Project**.
- The **New Project** menu option opens the *New Project Wizard* dialog that steps through the process of creating a new project.

The *New Project Wizard* pane one:

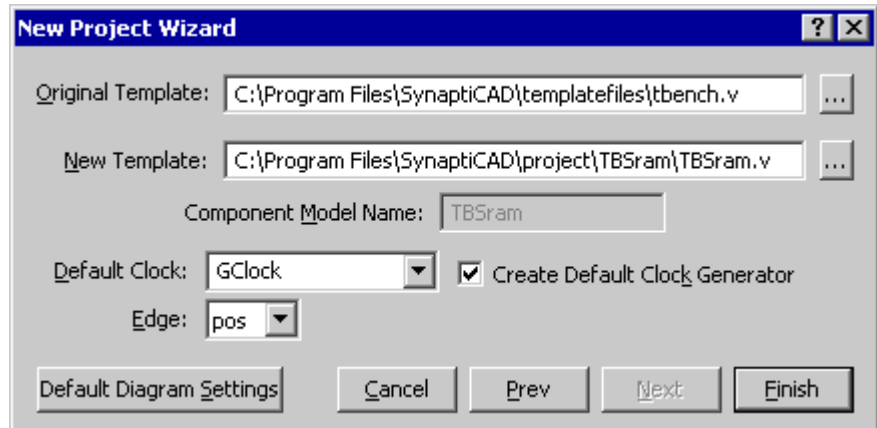
- **Project Name** will be the name of project and the subdirectory that the project will be stored in.
- **Project Directory** contains the path for the base directory of the project.
- **Project Location** displays the complete path to the project file.
- **Project Language** controls the test bench generation language. The available language selection is based on your TestBench License file.



- **Enable TestBuilder Integration** enables C++ code generation for Verilog projects. This adds many advanced language features, such as constrained random number generation.
- **Transaction-based Test Bench Generation** check box enables the generation of the multi-transaction based bus-functional model generation. If you uncheck this TestBench will generate single diagram test benches.

Click the **Next** button to view pane two of the *New Project Wizard*:



- The **Template** edit boxes specify the original and new (copied) template file names. The new template file will serve as the top-level source file for the test bench. This file is where the Component Model is generated. A default original template file name is provided, but you can create your own template files. (*Chapter 9: Project Component and Transaction Sequencer* has information on editing and changing the template file).
- The **Default Clock** drop-down edit box specifies the default clock to use when creating new constructs.
- **Edge** specifies the default clocking edge to use when creating new constructs.
- **Create Default Clock Generator** will create a transaction diagram with an output clock for clocked projects. This is useful for top-level projects. Sub-projects usually receive their clocks from the parent projects.
- The **Default Diagram Settings** button opens the *TestBencher Diagram Settings* dialog which is used to set up the default diagram settings, such as verbose code generation. This dialog is discussed in *Section 3.7 Diagram Settings Dialog - Overview*.

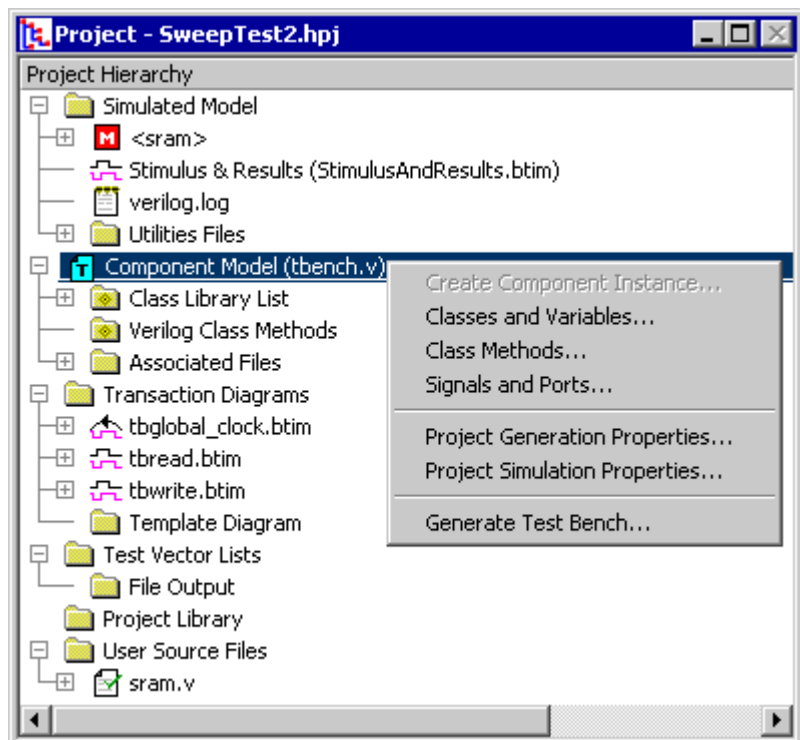


## 2.2 The Project Window

The *Project* window displays all of the different elements of the bus-functional model and the user source code (MUT). This includes all of the available transaction diagrams and sub-projects that can be used in the top-level Component Model. It also displays the project level variables and classes.

The basic controls of the *Project* window are as follows:

- Right click on any node in the Project tree to open a context sensitive pop-up menu that contains all of the operations that can be done to that particular node type.
- Double left click on any node to perform the default action for that node (usually open that file or object in an appropriate editor).
- To expand or hide branches of a tree, click  or .



- Drag and drop the column headings to resize columns.

Several folders are created in each project and are used to organize files and objects at the different levels of the test bench. Each of these folders will be discussed at different points in the manual. As an overview:

- **Simulated Model** folder contains the compiled Model Under Test and the *Stimulus & Results* diagram. See *Chapter 10: Generation and Simulation* for more information.
- **Component Model** is the top-level template file for the test bench. The folder contains all of the project-level classes, variables, class methods, and instances of sub-projects. See *Chapter 9: Project Component and Transaction Sequencer* for more information.
- **Transaction Diagrams** folder contains the template timing diagram and the timing diagrams that have been added to the project, and their associated source code files (in the level beneath the timing diagram). See *Chapter 3: Transaction Overview* for more information.
- **Test Vector Files** folder contains input and output test vector files. See *Chapter 8: Classes and Variables* for more information.
- **Project Library** folder holds any sub-projects that may be instantiated within the current project. *Section 2.3: Sub-Projects* discusses this folder.
- **User Source Files** folder contains source files for use in the test bench. Files with a green checkmark icon have been compiled into the test bench; files with a red X icon have not yet been compiled. *Section 3.2: Extracting MUT Ports into a Timing Diagram* has more information.

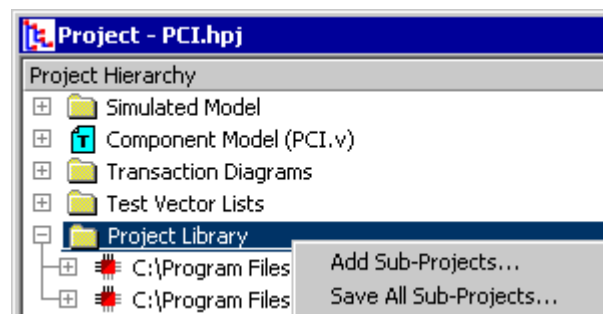
## 2.3 Sub-Projects

TestBench Pro supports hierarchical BFM design by allowing projects to be instantiated inside other projects. This lets you develop and verify complex test benches in an incremental manor. For example, if you are designing a test bench for an ATM switch, you can develop a project that can transmit an ATM cell to an interface port on the ATM switch. After you have tested your transmitter project, you can make it a sub-project and instantiate a copy of it for each different port of the ATM switch.

To use a sub-project you first add the sub-project to the **Project Library** folder in the *Project* window and then edit the default signal mappings that will be used when the sub-component is instantiated. *Section 2.4* covers the sub-project instantiation and port mapping.

Add a sub-project to the **Project Library** folder:

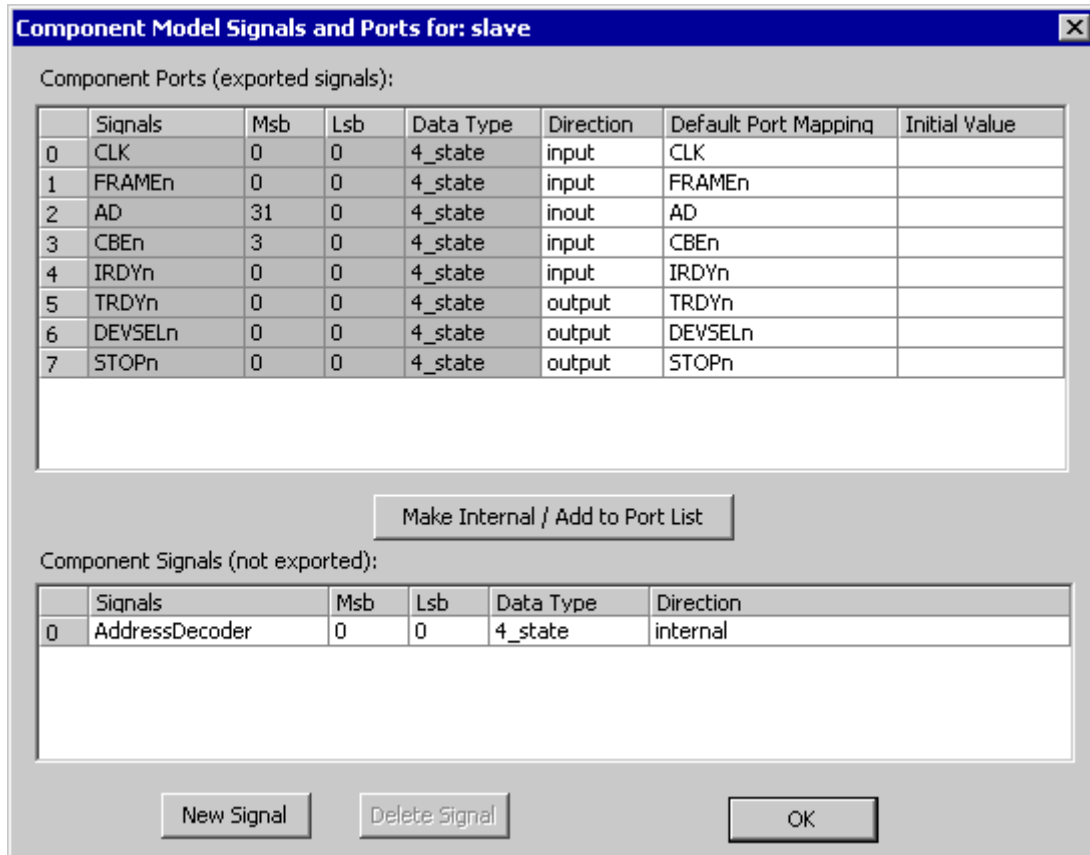
- Right Click on the **Project Library** folder and choose the **Add Sub-Projects** menu option. This opens a file dialog that lets you browse for projects. After you close the dialog, a sub-project with a red chip icon is added to the folder.
- Note the sub-project can be modified while the owning project is open, by expanding the sub-project tree. The sub-project will remain an independent project that can be opened and edited alone, as well.



Once you add the sub-project, you will need to setup the signals to be exported and the default mapping names for the component. To edit the default signal mappings

- Right-click on the sub-component (red chip icon) and choose **Signals and Ports** from the context menu. This opens the *Component Model Signals and Ports* dialog.
- Create ports for the sub-project by selecting signals in the *Component Signals (Internal)* section and clicking the **Make Internal/Add to Port List** button. This will move the signals up to the *Component Ports (exported signals)* section. Signals can also be dragged from one list to the other.

- Double-click on cells in the **Default Port Mapping** column and either type in a name or select a mapping signal name. Since the sub-project is not the real component you can type in partial names. For example if you plan to connect each instantiation of the sub-project up to a different port of the MUT with port names like *signame0* and *signame1*, a good partial port name would be *signame*. That way, after you instantiate the components you will only have to add the port numbers to the signal names.



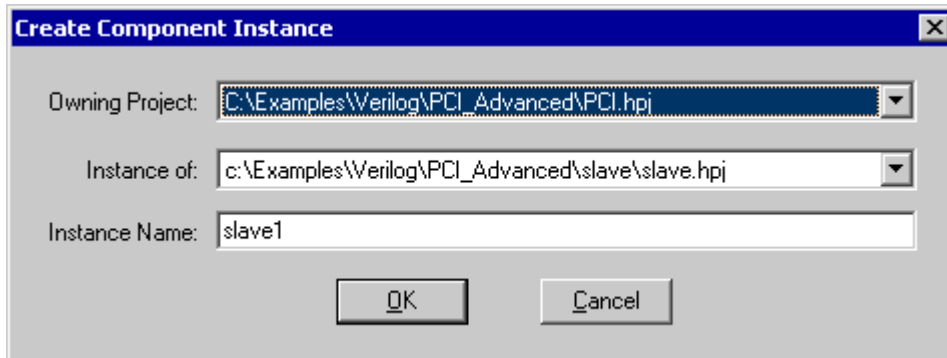
## 2.4 Component Instances of Sub-Projects

Projects listed in the **Project Library** folder can be instantiated and used by the containing project. When a sub-project is instantiated, TestBench will generate a component instantiation in the top-level template within the *STBSubComponentInstantiation* macro and make all of the transaction diagrams of the sub-project available to the owning project. The owning project can call the transactions of each instance of a sub-project. This is discussed in *Section 9.1: Transaction Calls*.

Create a Component Instance of the sub-project:

- Right-click on the sub-project node and select **Create Component Instance** from the context menu option. This will open the *Create Component Instance* dialog.
- Notice that by default the name of the selected **Owning Project** is the immediate parent project of the sub-project to be instantiated. The owning project can be changed using the drop-down list box. The **Instance of** drop down allows you to change the sub-project that will be instantiated.
- Enter the **Instance Name** to use in the instantiation of the Project Component.

- Click **OK** to close the dialog and add the component instance (green chip icon) to **Component Model** folder.



After the component instance is created, the port mappings should be edited in order to hook up the model to the signals in the owning project, unless the default port mapping are correct. To edit the port mapping for a component instance:

- In the **Component Model** folder, double-click on the name of the component instance to open the *Component Instance Signals and Ports* dialog.
- The top part of the dialog displays the ports of the sub-component. The **Port Mapping** column are the signal names that will be mapped to the sub-components outputs.
- To change the port mapping, double-click on a cell in the **Port Mapping** column and either type in a name or choose a signal name from the drop-down list. The signals in the list are the signals in the current project or the ports of the MUT.
- Click **OK** to apply the port mapping and close the dialog.

For more information about the *Component Signals and Ports* dialog, see *Section 2.6: Signals and Ports for Components*.

The Component Instance will be added to the *Project* window under the Component Model of the owning project. A green chip icon is used in the project tree to represent a Component Instance. If the Project Component that is instantiated by the Component Instance has ports, then the port mapping will appear beside the Component Instance.

## 2.5 Component and Component Instance Generation Properties

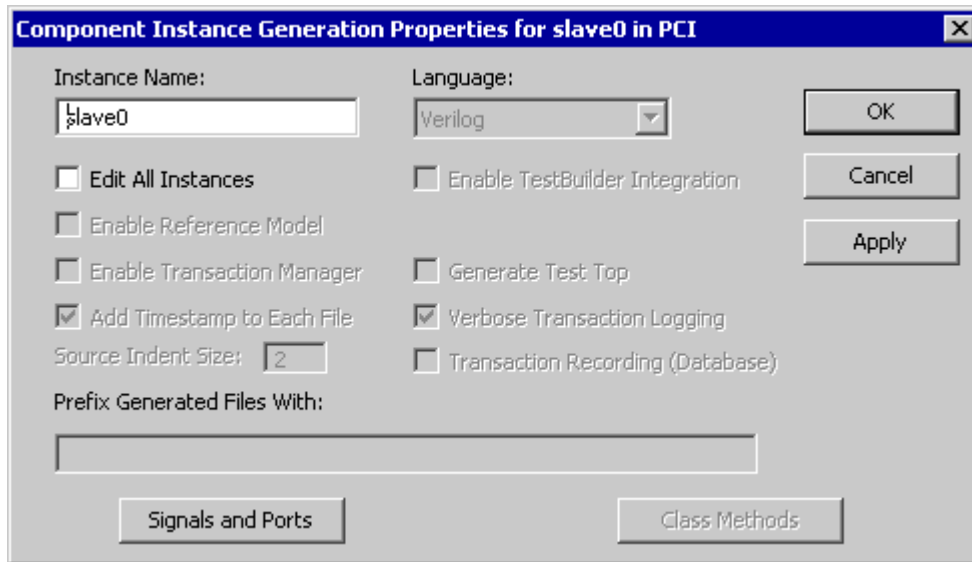
The top-level component of the project and each instance of a sub-project has its own set of properties that determine the way the code is generated for the component. These properties are edited through the *Project Generation Properties* dialog. This dialog also gives you access to the variables, parameters, and signal mappings of the Component. To open the *Project Generation Properties* dialog:

- Right-click on either the **Component Model** folder or on a component instance and choose **Project Generation Properties** or **Component Instance Generation Properties** from the drop-down list box to open the *Project Generation Properties* dialog.

The controls in this dialog change depending on if you are editing a Component Instance or the actual Component Model. For a Component Instance the following properties can be set:

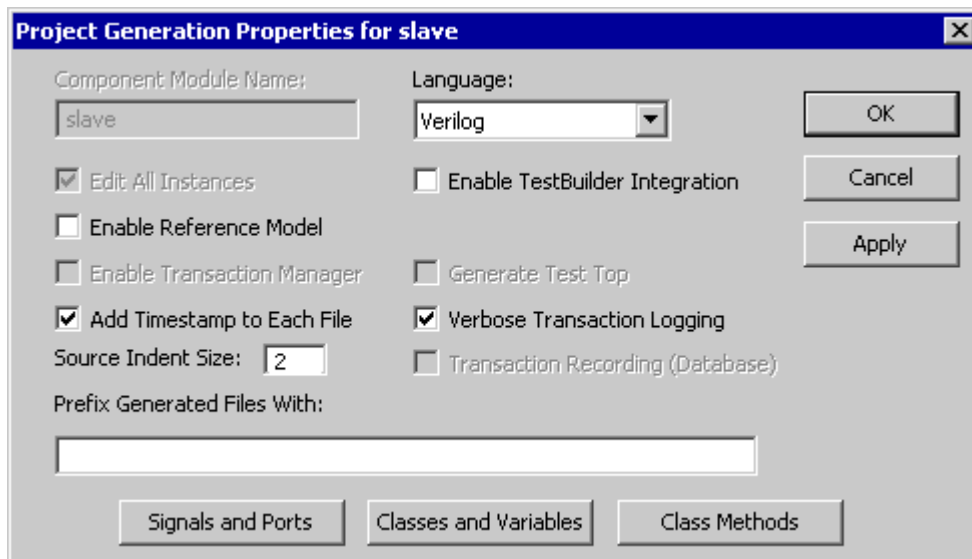
- The **Instance Name** box indicates which component instance is being edited and allows you can change the name of the component instance.
- Checking the **Edit All Instances** check box indicates that changes in the properties will affect all instances of a project. If you are editing from the **Component Model** folder of either the containing project or in a sub-projects **Component Model** folder you will be editing all of the instances.
- The **Signals and Ports** button opens a dialog that lets you edit how the component instance is hooked up to the containing project.

- The **Class Methods** button opens a dialog that lets you edit the project level class methods (transaction level class methods must be edited from the transaction). See *Section 8.7: Class Methods* for more information.



The top-level project and each sub-project definition contain defaults for the component generation properties used during code generation. The properties for this component can be edited by finding the **Component Model** folder and then opening the *Project Generation Properties* dialog. For sub-projects, the **Component Model** folder is located under the **Project Library** folder tree. When you are editing the properties at the component level you are editing all instances of that component. The following properties can be edited at the Component Model level:

- **Enable Reference Model** enables the generation of a golden reference model in C++, VHDL or Verilog. This feature is covered in *Section 2.7: Golden Reference Models*.
- **Enable Transaction Manager** enables the generation of the transaction manager code. This feature is covered in *Section 9.3: Transaction Manager and Test Reader*.



- **Language** and **Enable TestBuilder** control the generation language for this component. If the language for the project is changed, then the project template file must also be changed. *Section 9.6: Changing a Project Template File* discusses changing the project template file.

- The **Generate Test Top** checkbox is currently not in use.
- **Add Timestamp to Each File** is useful when the generated code is versioned. Disabling this will prevent the generated file from being different just because the time is different.
- **Verbose Transaction Logging** turns on extra reporting features for the Component Model and its transactions. This is useful for debugging and testing your components.
- The **Source Indent Size** specifies the number of spaces that are used for indenting blocks of code.
- **Transaction Recording** enables SDI transaction recording calls. These record every transaction that is run during simulation along with applied parameters. This is written to a database that can then be imported into Cadence's SignalScan. The simulation can then be viewed as a set of transactions.
- The **Prefix Generated Files With** edit box allows a line of text to be output at the beginning of each generated file. This is useful for noting author information or for adding keywords for version systems.
- The **Signals and Ports** button opens a dialog used to specify which signals will be available to external projects. It also allows you to define internal signals that are not contained in any transaction diagram.
- The **Classes and Variables** button opens a dialog that is used to edit the classes and variables that are included in that project. See *Chapter 8: Classes and Variables* for more information.
- The **Class Methods** button opens a dialog that lets you edit the project level class methods (transaction level class methods must be edited from the transaction). See *Section 8.7: Class Methods* for more information.

## 2.6 Signals and Ports for Components

The *Signals and Ports* dialog is used to view and edit the ports and internal signals of Component Models and definitions as well as the port mappings of Component Instances. TestBench automatically creates certain signal and port information based on the Model Under Test and diagram ports. Internal signals are also created automatically based on the specified port mappings for component instances.

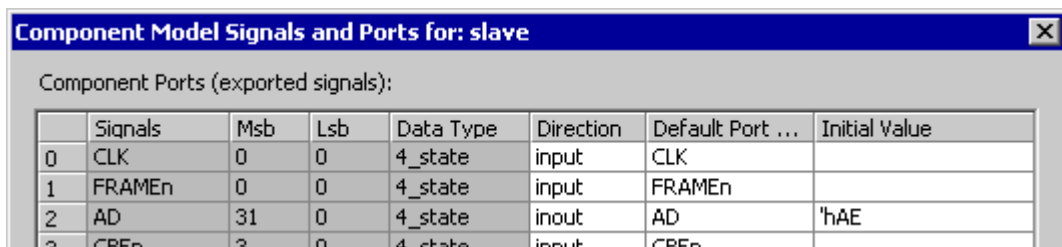
To open the *Signals and Ports* dialog:

- Right click the item to be edited (a Component Model, Component Definition or a Component Instance) and select **Signals and Ports** from the context menu. This will open the *Signals and Ports* dialog.

Note: This dialog is modal, which means that selecting any Component or Component Instance in the *Project* window will update the information displayed in the dialog for the selected item.

Editing signals and ports with the dialog:

- The signals and ports that have gray cells are automatically generated for the MUT and diagram ports, as well as from port mappings that have been defined from child projects.
- For Component Definitions, internal signals can have their direction edited, and ports can have either the direction, default port mapping, or the initial value edited.



- Component Instance internal signals can not be edited, and ports can have only the specific Port Mapping for the instance modified.

	Signals	Msb	Lsb	Data Type	Direction	Port Mapping	Initial Value
0	CLK	0	0	4_state	input	CLK	
1	FRAMEn	0	0	4_state	input	FRAMEn	
2	AD	31	0	4_state	inout	AD[63:32]	'hAE
3	CBFn	3	0	4_state	input	CBFn	

- New component level signals can be added when editing signals and ports for the Component Model (the component definition) by clicking the **New Signal** button. These signals can be made ports by changing the **Direction** of the signal from internal to input or output, or by dragging and dropping the signal from the bottom grid tree to the top. Note that when editing a Component Instance the signal and port definitions can not be changed.
- Port Mappings can be specified by double clicking a cell in the **Default Port Mapping** or **Port Mapping** column. When working with a Component Definition, the **Default Port Mapping** specified provides a default that will be used for the instances of the component being edited. If a Component Instance is being edited then the actual **Port Mapping** for that instance is being specified.
  - If the signal that the port should connect to is not yet defined in the owning component, just type the name of the new signal into the **Port Mapping** column. TestBench will automatically create a new signal in the owning component for this connection.
  - A **bit slice** can be specified for each **Port Mapping**. TestBench allows a bit slice to be specified as part of the port mapping, so that the connecting signal may be larger than the port. If the specified **bit slice** for the port mapping is outside of the connecting signal's bit range in the owning component, TestBench will automatically combine the two bit slices in the connecting signal. Each time the test bench is generated the bit ranges of component signals will be verified. This means that if a bit range is extended, but then no longer needed it will be reduced to its original size. The only case in which this will not happen is if the signal was explicitly added in the Signals and Ports dialog by the user. In this case, TestBench may need to extend the bit range, but will never reduce it. The Component Instance Signals and Ports image above shows an example of a bit slice specification for port **AD**.
- The **Initial Value** column provided for ports when editing a Component Definition allows an initialization for the port to be provided. The string entered in this field is placed directly into the generated source code without formatting. This value can not be edited at the Component Instance level. The images above show an example of specifying an initial value for port **AD**.

The Component Model has both exported signals and internal signals. By default, all signals that are generated by TestBench are internal, but the Component Signals and Ports dialog allows these signals to be exported by simply changing the direction of the signal.

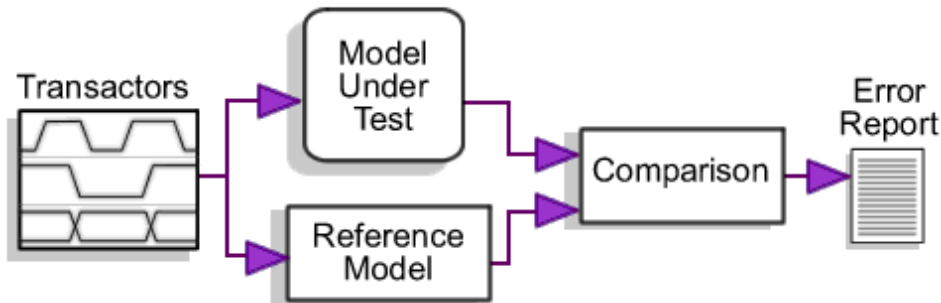
## 2.7 Golden Reference Models

TestBench can generate C++, VHDL and Verilog golden reference models that run in parallel with a VHDL or Verilog RTL model. Golden reference models are high-level descriptions of a design and are used to compare to the results of an RTL-level model during simulation. Reference models usually model interaction between components at the transaction level (e.g. read transaction/write transaction) instead of at the signal level. When the reference model is



created the apply calls will call both the diagram transactions and the equivalent reference model transaction. At the end of each transaction the outputs for the MUT and the reference model are compared and logged to the simulation log file.

## Golden Reference Model



TestBench generates all of the stub-functions for the golden reference model, keeping the transaction interface to the reference model the same as the HDL level model. TestBench uses the TestBuilder library to generate the C++ models. The user writes the behavioral C++, VHDL or Verilog code inside the stub-functions that enables the golden reference model to emulate the RTL-level model. To enable Reference Model Generation:

- In the *Project* window, right click on the **Component Model** folder and choose **Project Generation Properties** from the context menu. This opens the *Project Generation Properties* dialog.
- Check the **Enable Reference Model** checkbox and click **OK** to close the dialog.
- During the next code generation TestBench will generate either a C++ or Verilog reference module depending on the generation language. *Section 10.1: Generate the Bus Functional Model* describes how to generate the project code.
- The reference model file is written to the Project directory and is named *projectName\_skeleton* with the appropriate extension for the language (**.cpp**, **.vhd**, or **.v**). For C++ there is also a header file named *projectName\_emulator.h* that contains the class declaration for the reference model.
- Copy the *projectName\_skeleton.cpp*, *.vhd* or *.v* to a file named *projectName\_emulator.cpp*, *.vhd* or *.v*.
- Inside *projectName\_emulator* file uncomment any functions that you want to model, and insert behavioral code into the functions.
- *Section 10.2: Simulator and Compiler Settings Dialog* describes how to setup the C++ compiler so the TestBench can compile the model and hand it off to the HDL simulator.
- During the simulation the reference model will automatically compare the results of the MUT to the results of the reference model and send the results to the simulation log file.

### 2.8 Libraries and Use Clauses (VHDL only)

The *VHDL Libraries and Use Clauses to Include* dialog allows you to control the libraries and use clauses used by the VHDL diagrams in your project and by the top-level template file of the project.

To open the VHDL Libraries and Use Clauses dialog:

- Select **Options > VHDL Libraries and Use Clauses...** from the main menu.

OR

- Right-click on the project and choose **VHDL Libraries and Use Clauses...** from the context menu.

Changes made in this dialog will be applied to the current project, and stored in the **.hpj** file. If no project is open then the current settings will be applied to any new projects that are created, and saved in the TestBencher configuration.

The **View** dropdown allows you to change between selecting the VHDL libraries and the use clauses to include in your diagram.

#### To edit or add a new use clause or library include:

- Select **Use Clauses** or **VHDL Libraries** from the **View** dropdown.
- Double-click on an entry or on the first empty line in the list window and type in the use clause or library name. If the *use* or *library* tag is omitted from the statement, TestBencher will automatically add the tag to the statement before including it in the source code. Semicolons are also added as needed.
- All three of the following use clauses will work:

```
use myLib1.all;
myLib2.all;
myLib3.all
```

- All three of the following VHDL Library statements will work:

```
library myLib1;
myLib2;
myLib3
```

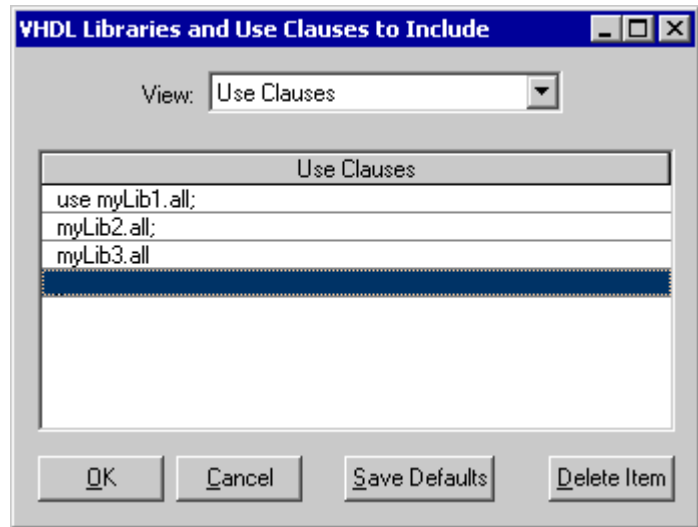
#### To delete a use clause or library include:

- Click on a clause or library in the list window to select the entry.
- Click the **Delete Item** button.

The **OK** button saves the current settings and closes the dialog. The **Cancel** button closes the dialog without saving the new settings. The **Save Defaults** button allows the current settings to be applied to any new projects that are started. If no project is open then this button will not be present, as settings will automatically be saved as defaults.

#### Exporting a diagram to VHDL

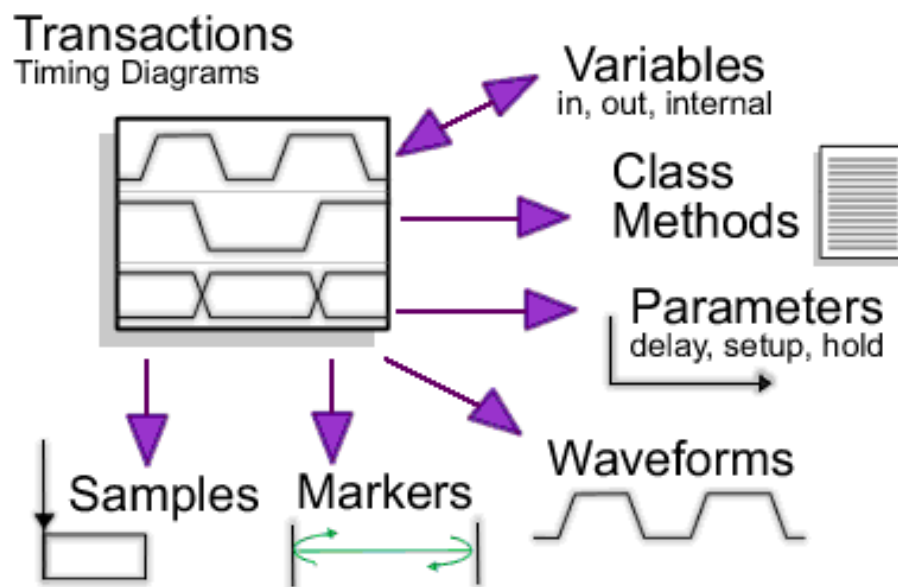
Whenever VHDL code is generated for a diagram, TestBencher checks to see if the diagram is included in the current project. If it is, then the settings for that project are used. If the diagram is not included in the current project, then the diagram will be exported using the default settings.



## Chapter 3: Transaction Overview

TestBench uses graphical timing diagrams to generate reusable timing transactions (e.g., read cycle, write cycle, interrupt cycle). The built-in timing diagram editor allows timing transactions to be described graphically using waveforms, samples, markers, delays, setups, and holds. A combination of variables and class methods are used to define algorithmic functions and attach them to the diagram.

The **Transaction Diagrams** folder in the *Project* window holds all of the transactions and the template diagram for the project. This branch in the project tree is used to create new diagrams, open diagrams for editing, and edit the transaction settings including Master and Slave setting. TestBench can extract signal and port information from the user MUT files and place it in a timing diagram.



The next four chapters cover the graphical elements that make up a timing diagram: signals and waveforms, timing parameters, samples, and markers. Timing diagrams can contain variables and class methods in addition to graphical elements. Transaction Level Variables can be used to pass information to and from the transaction and can also be used internally to store the results of calculations and samples. Class methods provide a graphical interface for developing functions and tasks that the timing diagram can call during simulation. To perform a calculation that is not easy to describe graphically, a HDL Code Marker can call a class method to do the calculation.

Timing Diagrams are used to generate the HDL code transactions for the bus functional model. The sections on Transaction Architecture, Diagram Properties, and Diagram Settings will provide a better understanding on how the transactions are generated and what type code is generated for the graphical elements.

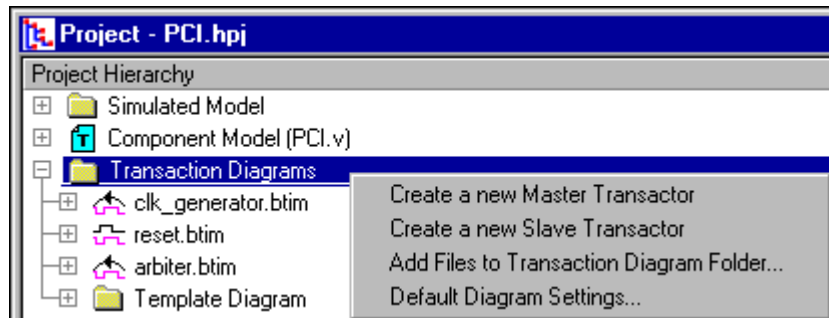
### 3.1 Template Diagram and New Transactions

The **Transaction Diagrams** folder in the *Project* window holds all of the transactions and the template diagram for the project. The template diagram will serve as a beginning diagram for all new transactions that are added to the project. You can add common elements to the template diagram like signals, waveforms, and variables. Each time you create a new transaction diagram for the project it will copy all of the elements of the template diagram into the new transaction. Then you can edit the new transaction as needed.

By expanding **Transaction Diagrams** tree and right-clicking on the tree nodes you can access a context menu with commands that will act on that node in the tree. Each Transaction consists of several files including the timing diagram file (\*.btim, binary timing diagram format) and depending on the generation language one or more code generation files. Each time the timing diagram is saved the source code updated for the transaction.

**To create a new transaction diagram:**

- Right-click on the *Transaction Diagrams* folder and choose either **Create a new Master Transactor** or **Create a new Slave Transactor** from the context menu. This will open a file dialog.



- Name the new transaction and save the file. This will add a **name.btim** file to the **Transaction Diagram** folder and load the new timing diagram into the *Diagram* window. The new timing diagram will contain all of the elements and settings of the template diagram.

**To open an existing timing diagram:**

- Double-clicking on a timing diagram node will open the timing diagram file in the *Diagram* window.


**To change the Master or Slave Setting of a Timing Diagram:**

- Right click on the timing diagram name and choose **Diagram Settings** from the context menu to open the *Diagram Settings* dialog.
- Choose either **Master Transactor** or **Slave Transactor** from the *Diagram Execution* section. Master transactors will run once and then stop. Slaves will continuously loop. For more information, see *Section 9.1: Transaction Calls* in the TestBencher Pro manual.

## 3.2 Extracting MUT Ports into a Timing Diagram

TestBencher can extract port and signal information from the user source files and place the data in the active timing diagram. Usually you will want to place the signals in the template diagram for the project, so that the same set of signals will be available for all new transactions that will be added to the project.

Add the MUT files and extract the ports:

- Right-click on the **User Source Files** directory and choose **Add Files to User Source Files Folder** from the context menu. This will open a file dialog that will let you browse and add the model under test files.
- Click the **Extract MUT ports into diagram** button  on the *Simulation Button Bar*. This will cause the source files to be parsed, the MUT to be built, and the top-level port information to be added to the active timing diagram.

Note: After *Extract MUT Ports* has been performed, the MUT can be changed to any module in the **User Source Files** folder by right-clicking and choosing **Set As MUT** from the context menu. You can then rerun the *Extract MUT Ports* to grab the ports from new module.

When the *Extract MUT ports* step is performed, one of the components is selected as the top-level MUT file. This component is designated by one set of brackets (ex., <bsram>) and placed in the **Simulated Model** folder. For Verilog users, the selected component will be the top-level component found in the hierarchy specified by the source code. For VHDL users, this will be the first parsed component.

### 3.3 Transaction Level Variables

In addition to the graphical elements of a transaction, timing diagrams can use variables to store sampled data, used within an expression (e.g., a sample's condition), or as an input/output to the transaction. There are three types of variables that can be used inside a transaction: diagram level, state variables, and parameter based variables. This section gives a quick overview of the variable types and how they can be used. Later sections cover the specific variable types in greater detail.

When using a variable as an input or output to the transaction there are certain rules that are based on how the variable type is defined. The following table describes the differences and uses for the different variable types when used as a port in the transaction.

Structure	Direction	Advantage	Disadvantage
Diagram Variable	Input, Output, or Internal to the diagram	Size and Type are controlled by the definition. Type can be simple or complex. Most versatile variable type.	Only visible in the View Variables dialog until referenced in the diagram. Not convenient for driving Parameter constructs such as Delays.
State Variable	Input only	Very quick to add.	Size and direction is controlled by the signal and waveform direction to which the variable is attached.
Parameter	Input, Output for Samples	Contains both min and max values for passing time values to Parameter constructs.	Both min and max values are not needed for many situations. Assumed to hold either a real or clock time.

**Table 1: Transaction Level Variable Types**

#### Diagram Variables

These are the most versatile structures for holding information and for passing information into and out of the Transaction. *Section 8.3: Variables* has more information on defining and using these structures.

To add and use a diagram variable:

- To create a diagram level variable, click the **View Variables** button in the *Diagram* window to open *Classes and Variables* dialog. Define a Variable as described in *Section 8.3: Variables*.
- To use a diagram variable as a waveform state, put an @ symbol prefix in front of the name like **@name** to indicate that it is a variable and not just a text string. You can also select the variable from a list by clicking the **Variables** button in the Bus State dialog.
- To use a diagram variable in a block of code just use the variable name. Some examples of code that might use a variable are class methods, condition statements for samples and markers, and marker loop parameters such as begin, end, and increment values.
- To use a diagram variable to store a sample value, use the **Variable** features of sample's *Code Generation Options* dialog as described in *Section 6.6: Storing Sample Values in User Defined Variables*.

#### State Variables

State variables are a special form of diagram variables that are used exclusively for setting waveform state values. The size of the state variable is set by the containing signals. For example if \$\$addr is used in two signals Address\_low[3:0] and Address\_high[12:8] then \$\$addr will be defined to have a size of [12:0] or 13 bits. *Section 4.3: Driving Waveform States with Variables* has more information on state variables.

To add a State variable:

- Double-click on a waveform segment to open the *Edit Bus State* dialog.
- Type in a name with the prefix \$\$ into the *Virtual Edit* box. For example, \$\$addr is a valid name for a state variable.

### Parameter Variables

Graphical and free parameters can be used to hold information and to pass information into a transaction. Samples can also pass information out of a transaction. These are specially designed variables that are used to pass time (either real time or clock cycle time) to transactions. These variables can have either a one or both a min and a max value defined for one variable name.

These are the recommended variable type to use to pass information to the min and max boxes for delay, setup, hold, or sample parameters. They are also the recommended variable type to use in a Clock period formula. Other variables can be used but then you need two of them, unless you want the min and max to be the same value.

To add a parameter variable:

- Either add a graphical parameter to a timing diagram (delay, setup, hold, or sample) or click the **Add Free Parameter** button in the *Parameter Window*.
- Double-click on the parameter to open the *Parameter Properties* dialog. Check the **Is Apply Subroutine Input** and the **Enable HDL Code Generation** check boxes. This causes a subroutine parameter to be added to the transaction apply calls.
- Samples can also generate an output parameter that passes the sample value out of the transaction when the transaction ends. To do this, check the **Store Sample Values as Subroutine Output** check box, in the *Code Generation Options* dialog (see *Section 6.2: Sample Conditions and Actions*).
- To reference the parameter variable use *parameterName\_min* or *parameterName\_max* in any code block, condition statement, or in the min or max boxes of a parameter. For parameters you can also use the name without the min/max suffix, and TestBench will make the best guess as to which value you want.

## 3.4 Diagram-Level Class Methods

In addition to the graphical elements of a transaction, timing diagrams can use class methods. Class methods are user defined functions or tasks that can be called from within the transaction to perform an algorithmic process. Class methods can be defined at the diagram and project level. Diagram level class methods can access any diagram-level variable, state variable, or parameter variable (*Section 3.3: Transaction Level Variables*).

Class methods can be called by HDL Code Markers and the Sample Actions to perform zero time calculations for the timing diagram. Diagram-level class methods are written in the transaction generation language. See *Section 8.7: Class Methods* for more details. To define a diagram-level class method:

- Click the **Class Methods** button in the *Timing Diagram* window. This will open the *Class Methods* dialog that is used to define and edit diagram-level class methods. *Section 8.7: Class Methods* describes how to use this dialog.
- To use a diagram-level class method in an HDL Code marker (see *Section 7.6: HDL Code Markers* for more information), type in the name of the class method along with any parameters into the code box of the marker.
- To use a diagram-level class method in a Sample Action, choose **User Defined Action** for the action type. Then type in the name of the class method along with any parameters into the code box of the marker.

## 3.5 Transaction Architecture

This section describes how TestBench models a transaction diagram. A firm understanding of this material will help you avoid errors in your transaction diagrams and speed the process of debugging your system.

TestBench generates a transaction for each timing diagram in the project. These transactions are *modules* for Verilog, *entity/architecture* pairs for VHDL, and *classes* for TestBuilder. Regardless of the language, the transactions use the same general architecture. And in all languages, the transactions have a similar functional API that can be used to trigger them (diagram apply calls).

### Clock domains

Inside each transaction there may be one unlocked sequence process and several clocked sequence processes. A sequence process is created for each clocking domain in the diagram to drive signals and trigger parameters (Delays, Samples, Holds, Setups, Markers) that are synchronous with the given clock. Each domain will run in parallel (concurrently) once the diagram is started. Typically, there will only be one clock domain in the diagram. But, if you have multiple domains in the diagram, then it's important to know what is placed in each domain if you have looping or blocking parameters. For example, a Marker loop that is attached to the falling edge of CLK will only loop around items that are also in the CLK\_neg clock domain. Items that are in the unlocked domain wouldn't get placed into the loop. Also, items that can potentially block a process (Samples, Markers, Sensitive Edges) will only block the clock domain that they are placed in. The following sections will go into more detail on how blocking and looping constructs work.

The table below shows how the clock domain is determined for each type of construct.

Construct Type	Clocking Domain
Signals	Clock and Edge of signal ( <i>Signal Properties</i> dialog)
Sensitive Edge	Clock and Edge of signal that contains sensitive edge
Samples	A sample's clock domain is the process that triggers it. See the table below to determine a sample's triggering process. If <i>not unlocked</i> , then Clock and Edge in <i>Delay</i> dialog. Otherwise, the starting edge of the delay sets the clock domain.
Delays attached to edge	Signal and edge that is pointed to by the Setup
Setups	Signal and edge that is pointed to by the Hold
Holds	The relative edge sets the clock domain
Markers attached to edge	Unlocked
Markers attached to time	

### Signals

Signal states are driven based on three factors: how it is drawn, its clocking domain, and the cycle based setting **Include Time Delays** in the *Diagram Settings* dialog. Unlocked signals are driven at the times that the edge transitions are drawn. Clocked signals are driven based on the clocking edges detected during simulation. The **Include Time Delays** option controls whether or not inter-clock cycle delays are generated for clocked signals. If this option is off, then clocked signals are only driven at clock edges (See *Section 3.9 Diagram Settings – Language Tabs*). The event timing for signals is covered in detail in *Section 4.1 Drawing Waveforms*.

### Blocking Constructs (Sensitive Edges, Samples, and Markers)

There are three different types of constructs that can be used to block the execution of a clock domain. A *sensitive edge* (*Section 4.8*) will cause its clock domain to wait on the edge, which will block all other items in that same clock domain until the edge is detected. A Sample that has the **blocking** setting checked (*Section 6.3*) will block its clock domain until the sample completely finishes, including execution of its *then* or *else* action. And a *Wait Until Marker* (*Section 7.4*) will block its clock domain until the condition specified becomes true. If the marker is attached to an edge it will only check for the condition at each clock edge of the clock domain.

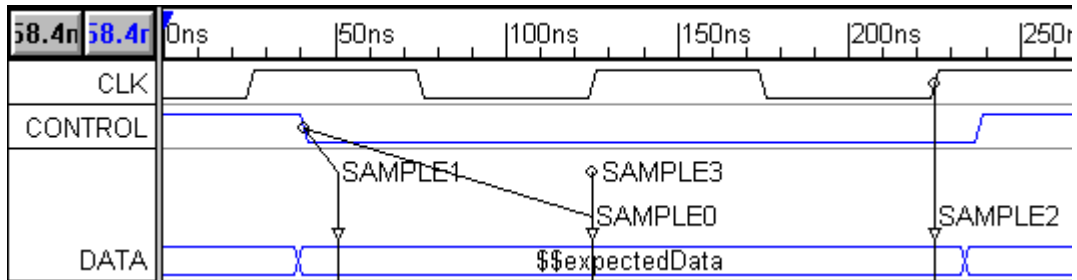
### Samples

The code for samples will sometimes be generated in a separate process and sometimes within the clock sequence process that triggers it (in-line). Whenever possible, the sample code will be generated in-line to make it easier to debug the generated code. However, if the sample is *non-blocking* and needs to wait for simulation time to pass, then that sample will be placed in its own process or task and triggered by the sequence at the appropriate time. Some examples of samples that need to wait for simulation time to pass are *windowed samples* or samples that are delayed from their triggering point.

The sequence process that triggers the sample is determined from the combination of the triggering edge and the *Samples Properties* dialog *clock* and *edge type* settings.

Triggering Edge	Sample Properties clock and edge type	Triggering Sequence
No trigger (time only)	Ignored when no trigger edge	Unclocked sequence
Attached to an edge	Unclocked	Trigger edge sequence
Attached to clock edge	Matches triggering edge	Clock sequence from dialog
Attached to an edge	Different than triggering edge	Clock sequence from dialog with a level sensitive check on the triggering signal

The example diagram below contains three domains: CLK\_pos, CONTROL\_neg, and Unclocked.



**CLK\_pos:** This is a clocked diagram so most of the graphical elements were automatically created with the clock/edge already set to **CLK** and **pos edge** in the *Properties* dialog of the element.

- **SAMPLE2:** triggered from the third clock edge.
- **SAMPLE0:** at second clock edge, a level sensitive check is performed on the CONTROL signal and if it is 0 then the sample will trigger. If instead of a level sensitive check on CONTROL, you want to perform an edge sensitive wait on CONTROL, then set the **falling edge sensitive** check box in *Signal Properties* dialog for the CONTROL signal.

**CONTROL\_neg:** When SAMPLE1 was created we used the *Sample Properties* dialog to change the clock setting to **unclocked**. This setting change will allow SAMPLE1 to be triggered when the CONTROL signal goes negative (compare this to the behavior of SAMPLE0 above).

**Unclocked sequence:** SAMPLE3 is an absolute sample (not attached to an edge) so it will be placed in the unclocked sequence. SAMPLE3 will trigger at 125 ns.

### Delay Parameters

Delays are placed in clock domains based on the same rules that apply to Samples. The only difference is that it is not possible to create a delay that is not attached to an edge. So, Delays will never be triggered by the unclocked sequence.

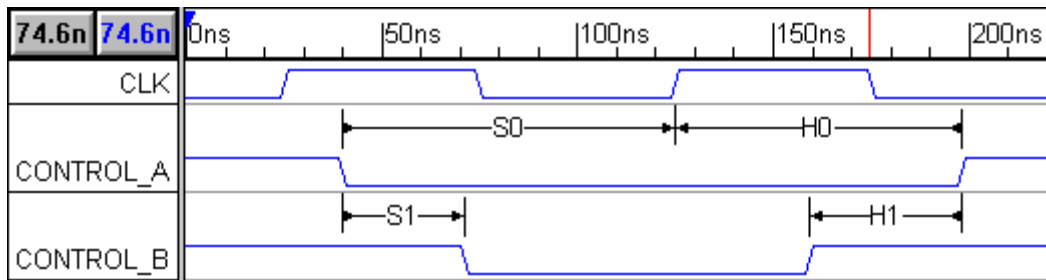
### Setups and Holds

Setups and Holds are placed in clock domains based on the edge that they point to. Since they cannot be attached to time (such as Samples), they will never be triggered by the unclocked sequence.

In the following example there are three different clock domains because the setups and holds point to three different edges:

- **CLK\_pos** triggers both S0 and H0 at the second positive edge of CLK.
- **CONTROL\_B\_neg** triggers S1 at the first negative edge of CONTROL\_B.
- **CONTROL\_B\_pos** triggers H1 at the first positive edge of CONTROL\_B.

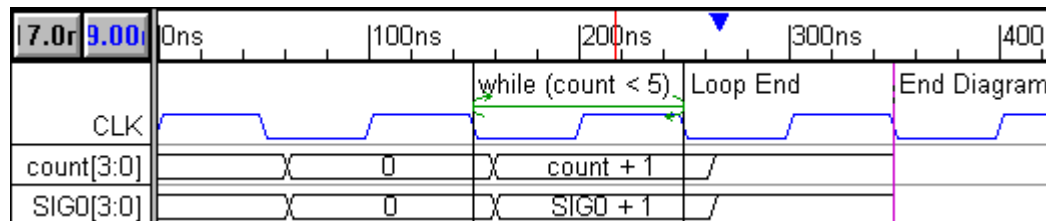




### Markers

When looping behavior is needed over a particular set of clock cycles or time, then *Looping Markers* have to be used (see [Section 7.5](#) for more details on markers). They will only loop over the clocking domain that they are placed in.

The following example demonstrates how a marker loop might not cover everything in the diagram. The *count* signal has its Clock set to "CLK" and Edge set to "neg". The *SIG0* signal is unlocked. The marker loop will loop over the CLK\_neg clocking domain since the Begin and End loop markers are attached to falling edges of CLK. Since signal *count* is in the same clock domain, during simulation *the signal* will be incremented at each negative clock edge until it reaches 5. Since *SIG0* is Unlocked it is not included in the loop and therefore will only get incremented once.



### Output Clocks (Clock generators)

When creating a clocked test bench with TestBench, there is usually either one timing diagram that has an output clock or the clock is generated in the MUT code. All of the other timing diagrams use an input clock. This makes it easier to synchronize the transactions during simulation.

Each output clock has its own process that generates the clock during a simulation. This clocking process is in addition to any unlocked or clocked processes that are used to synchronize signals and parameters. The clock generation process will take into account as many of the *Clock Properties* as are supported by the generation language.

## 3.6 Diagram Properties

The cycle based settings and the include file list of a timing diagram are edited using the *TestBench Diagram Properties* dialog. Diagram properties are significant to the operation of the diagram and can break or dramatically change the way the diagram works during simulation. These properties are saved in the timing diagram file. Other diagram settings that affect the generation of the code but not the operation of the diagram are edited through the *TestBench Settings* dialog as discussed next in [Section 3.7: Diagram Settings Dialog - Overview](#).

To edit the Diagram Properties:

- Open the diagram for which you will be changing the properties.
- In the *Diagram* window, right-click in the signal label area and choose **TestBench Diagram Properties** from the context menu. This will open the *TestBench Diagram Properties* dialog.

### Including HDL Code Library Files

If you have external code modules that you want to make available to the transaction then you can use the interface in the *Diagram Properties* dialog to make that code available. Files can either be included before the transaction, using the equivalent of the Verilog *include* statement, or files can be included inside the module. The method for including code within the transaction varies by language. If possible the code is included using something like the *include* statement and if that concept is not supported then the code is echoed within the transaction. If you have HDL functions or tasks that you would like to write and use within a transaction then use the **Class Methods** dialog as discussed *Section 8.7: Class Methods*. Class Methods is a newer interface that is more flexible and it makes it easier to modify the code and parameters of the functions.

#### To Add an HDL Code Library File to the Diagram:

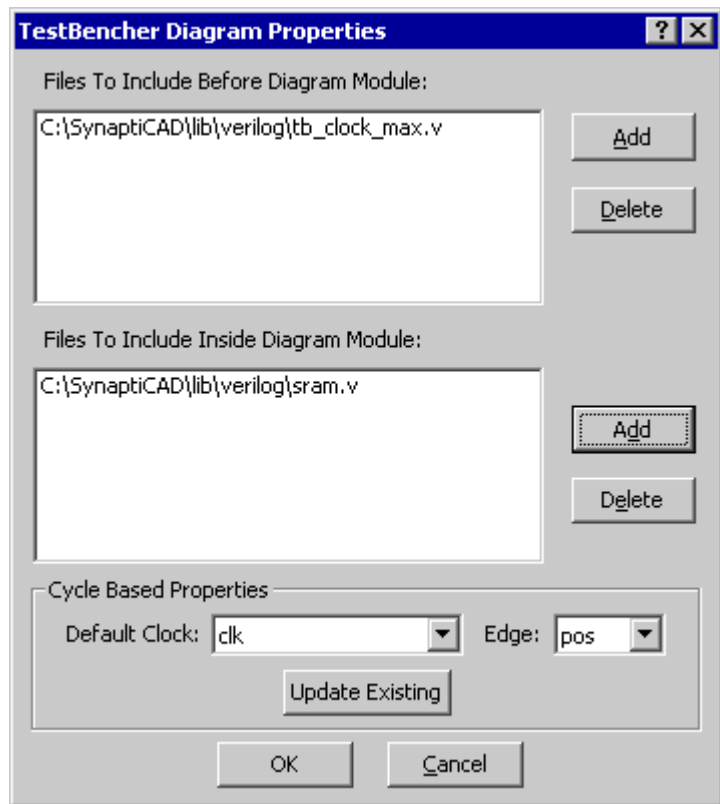
- Click the **Add** button to the right of the appropriate list box to open a file dialog that lets you browse for the include file. Click **Open** to close the file dialog.

Although the code generation for Verilog and VHDL will treat the file lists from this dialog differently, the file selection process for the languages is the same in this dialog.

### Cycle Based Properties

The *Cycle Based Properties* control how clocked signals and events are generated. These settings provide default clocking signals and edges to be specified for a diagram. This area also allows existing signals and parameters to be updated to a new clocking signal and edge.

- The **Default Clock** and **Edge** settings provide default values for the clocking signal and sensitive clock edge in a diagram.
- The **Update Existing** button is used to update all signals, samples, delays and anything with a clocking signal defined to the currently selected **Clock** and **Edge/Level**.



## 3.7 Diagram Settings Dialog - Overview

The *TestBencher Diagram Settings* dialog allows you to change the settings for a specific diagram in the project, or to change the default settings for new timing diagrams created while the project is open. These settings control how the source code is generated for a timing diagram and provide defaults for new items added to a timing diagram. In particular, this dialog lets you enable verbose code generation options for each timing diagram, which is especially useful for debugging a transaction. Properties that affect the basic operation of the timing diagram like cycle based settings and the include file list are edited using the *Diagram Properties* dialog as discussed in *Section 3.6: Diagram Properties*.

The diagram settings are saved as part of the project. The settings for the default diagram and for each timing diagram in the project are saved separately. Since this dialog controls code generation options that are specific to TestBencher it is not available for VeriLogger-style projects.

To open the *Diagram Settings* dialog:

- Right-click on the diagram name in the *Project* window, and choose **Diagram Settings...** from the context menu.

OR

- Select **Project > Default Diagram Settings...** from the main menu.

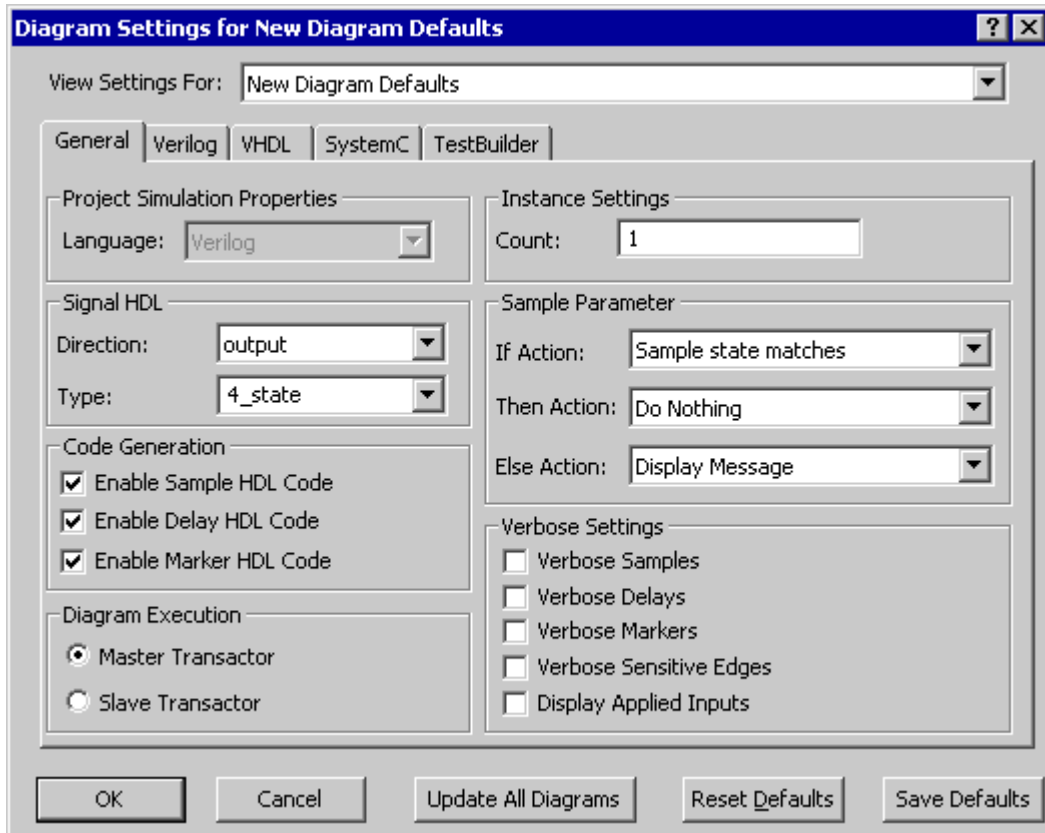
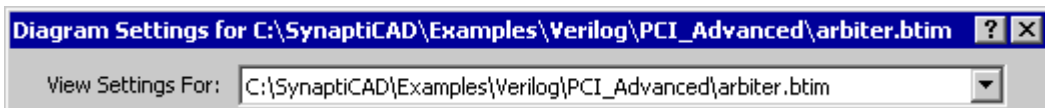


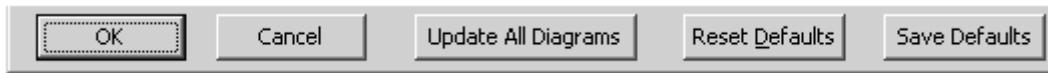
Diagram Operation:

- The **View Settings** drop down allows you to quickly view the settings for different timing diagrams in the project.



- There are several options available for saving and closing the dialog.
  - The **OK** button to saves the new settings
  - The **Cancel** button undoes all of the changes.
  - The **Update All Diagrams** button changes the settings of all timing diagrams associated with the current project to these settings.
  - The **Reset Defaults** returns the settings for the current diagram to the application default settings (*not* the New Diagram Defaults).

- The **Save Defaults** button saves the current settings as default diagram settings for new projects.



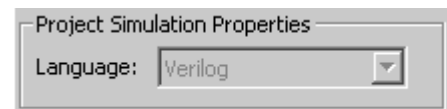
The *Diagram Settings* dialog contains a *General* tab and one or more language-specific tabs. The language-specific tabs that are present will depend on what languages you have licensed for use with TestBench. TestBench can currently support Verilog, VHDL and TestBuilder. These settings are covered in sections 3.8 *Diagram Settings - General Tab* and 3.9 *Diagram Settings - Language Specific Tabs*.

### 3.8 Diagram Settings Dialog - General Tab

The **General Tab** in the *Diagram Settings* dialog controls code generation features for non-language specific items, such as Verbose settings and the default actions for samples. *Section 3.7: Diagram Settings Dialog - Overview* covers the basic operation of the dialog, and *Section 3.9: Diagram Settings Dialog - Language Specific Tabs* covers the language specific options.

#### Project Simulation Properties

The **Language** dropdown controls what language the timing diagram will be generated with. The language choices depend on what languages have been licensed for use with TestBench. TestBench can generate Verilog and VHDL testbenches.



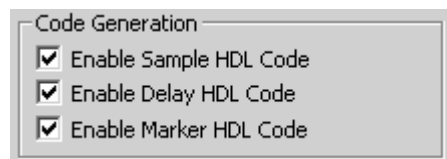
#### Signal HDL

The **Direction** dropdown allows you to select the default signal direction for new signals. For more information on signal directions, see *Section 4.2 Drawing Waveforms and Bi-directional Signals*. The **Type** determines the default Syncad signal type for new signals. The Syncad signal types are language-independent - these are converted to the appropriate language-specific signal types during generation.



#### Code Generation

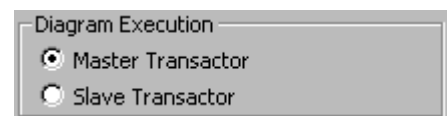
The three check boxes in the *General* tab allow you to toggle whether HDL code generation is enabled for samples, markers and delays .



#### Diagram Execution

The *Diagram Execution* settings control how the diagram will be executed in the test bench. There are two types of transactors available, Master and Slave. Execution control is specified in the Sequencer Process of the Template File (*Section 9.1: Transaction Calls* has more information):

- The **Master Transactor** type runs a single time, and can either block the execution of other transactors or it can be executed concurrently. An abort method is also provided for Master Transactors.
- The **Slave Transactor** type runs in a looping mode until it is stopped with an Abort method. Slaves can either execute concurrently or block other transactions while they are looping.

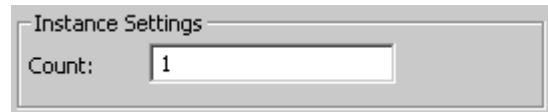


For more information, see *Section 9.1: Transaction Calls*.

### Instance Settings

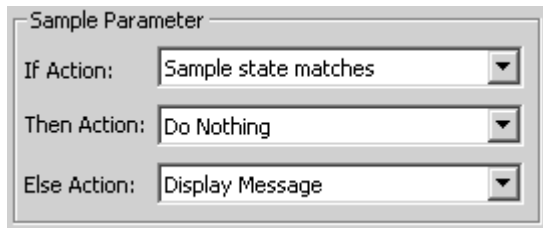
The Instance Settings are used to specify information for Piplining Transactions.

- The **Count** determines the number of instances of a transaction. These instances will be automatically instantiated in the test bench.



### Sample Parameter (General)

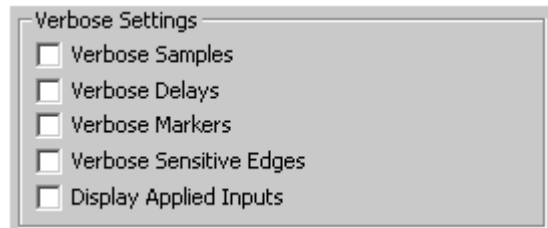
Sample parameters generate self-testing code in the test bench. By default, if the sampled state matches the expected state, then the code will continue as planned; otherwise, the code will display an error message in the message log. The default conditions for new samples created in the diagram can be set from these three dropdowns. For more information, see *Chapter 6: Transaction Samples*.



### Verbose Settings (General)

The *Verbose Output Settings* are used to control the amount of debugging output that occurs during simulation. The verbosity settings in this dialog will control the level of information that is output during the transaction execution for the active timing diagram.

Note: All of the verbose messages include the diagram name and simulation time.



#### Verbose Sensitive Edges display for VHDL and Verilog:

- When Sequence Verification process starts.
- When an edge is detected on a sensitive signal.
- When an unexpected edge occurs on a sensitive signal.

#### Verbose Delays display:

- When the delay starts.
- When a conditional test fails, causing the Delay to be disabled
- When the Delay makes a signal assignment.

#### Verbose Samples display:

- When the Sample has been triggered to run.
- When the Sample window has been entered.
- When a Sample causes the diagram to restart.
- When a Sample triggers another sample.
- When a Sample triggers a delay.

#### Verbose Markers display their name, type, and the simulation time. Some markers display more information:

- Wait Until Marker:
  - When the marker starts waiting for its particular condition.
  - When the marker finishes waiting.

- While Loop Marker, For Loop Marker, and Repeat Loop Marker display:
  - When the loop starts.
  - Each subsequent time and condition for the loop
- Exit Loop When Marker displays when and the condition for breaking the loop.

### Display Applied Inputs

- When enabled, all inputs to calls for the selected transactor will be displayed in the simulation log.

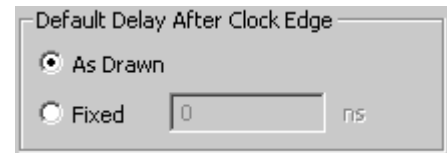
## 3.9 Diagram Settings Dialog - Language Specific Tabs

The **Language Specific Tabs** in the *Diagram Settings* dialog controls code generation features for the language specific items like default signal type that are different for each generation language. *Section 3.7: Diagram Settings Dialog - Overview* covers the basic operation of the dialog, and *Section 3.8: Diagram Settings Dialog - General Tab* covers the general options that are the same across languages.

### Cycle Based Settings

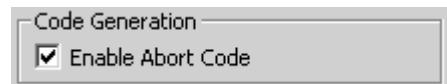
This setting affects code that is generated for clocked signals. If **Fixed** is selected, then the amount of time specified is the delay after the clock edge before events occur. If **As Drawn** is specified then the length of the delay is dependent upon the time at which events on the signal are drawn. This option is a direct replacement for the **Include Time Delays** option in previous versions of TestBencher. Previous projects will be converted as follows:

- Include Time Delays ON => As Drawn
- Include Time Delays OFF => Fixed to 0



### Code Generation (VHDL Only)

The **Enable Abort Code** checkbox allows you to toggle whether or not abort code will be generated. The biggest advantage for turning off this code generation is that the amount of code for each diagram will be reduced. The disadvantage is that certain features will be disabled because they rely on the abort code to function. These features include abort transaction apply call, end diagram marker, end diagram sample conditions, diagram timeouts, and delay timeouts.



It is expected that this feature will be used mostly for limited testing or for diagrams that do not need an abort. The global clock diagram, for instance, should never have the abort code turned off or you would have to manually end the simulation.

### Time-out Settings (Language-Specific)

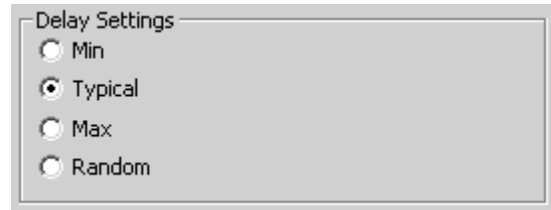
A transaction timeout will aid in the prevention of an endless wait condition occurring in a timing transaction. The time selected as a time-out duration is measured in diagram lengths for



an individual diagram. For instance, a diagram whose entire execution should be complete in 150ns could have a time-out duration of 150ns, 300ns, 450ns, etc.

### Delay Settings (Verilog & VHDL)

The delay settings determine how delays will be computed during diagram code generation. This setting determines which delay value is used in min:typ:max expressions. The random setting allows a random value to be computed from within the range of the min and max specified for the delay.



A dialog box titled "Delay Settings" with a light gray background and a thin border. It contains four radio button options: "Min", "Typical", "Max", and "Random". The "Typical" option is selected, indicated by a black dot in the center of its radio button.

Delay Settings

- Min
- Typical
- Max
- Random





## Chapter 4: Transaction Waveforms and Signals

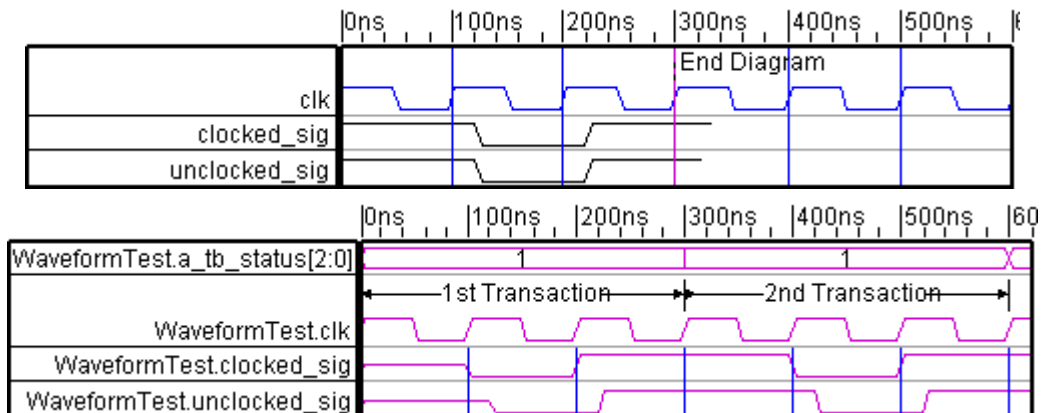
Signals and waveforms are the heart of the timing diagram. The waveforms can be quickly sketched using the built-in timing diagram editor. The state values of waveforms can be hard coded. In TestBencher, state values can also be passed into waveforms through a variable, or conditionally driven by a variable.

Most of the signals will be automatically added to the timing diagrams by extracting the signal and port information from the model under test files. However signals can be added manually. Several types of signals including internal and clock signals can be added to the timing diagram to achieve different behaviors.

The edges on waveforms are responsible for triggering the markers and parameters that are attached to them. If more than one parameter or marker is attached to the same edge then triggering order can be set using the *Edge Properties* dialog. Also, edges of a can be made sensitive so that the transaction will wait for that particular edge to occur.

### 4.1 Drawing Transactions for TestBencher

When drawing the waveforms for a timing diagram it is important to remember that transactions do not automatically include an event at time zero and that only the drawn events are driven. This is a feature that allows transactions to be reused any time during simulation without implying any initialization information. Below is a timing diagram that we drew, and a simulation results diagram that shows what happens when the timing diagram is applied twice. In the simulation results diagram, note that both signals are tri-stated until they are driven the first time. Also notice the transitions of *clocked\_sig* automatically sync up to the clock edges, because *clocked\_sig* is a clocked signal with *Include Time Delays* (see *Section 3.9: Diagram Settings Dialog - Language Specific Tabs*) is unchecked.



When signals require a specific initial state, there are two ways to do it: an initialization event and an initialization timing diagram. A small initialization event can be drawn at the beginning of the timing diagram as shown in the initialization signal. A potential disadvantage is that this event will be driven each time the transaction is called. To overcome this disadvantage, an initialization diagram can be developed and called once at the start of the test bench execution. An initialization diagram is just a simple timing diagram with waveforms that drive the required signals to an initial state. The TestBencher Pro Basic tutorial provides an example of an initialization diagram.

How TestBencher generates the code for the waveform depends on three things: how it is drawn, the clocking domain, and the cycle based setting **Include Delay Time** in the *Diagram Settings* dialog. When TestBencher generates code for clocked signals they are driven based on the clocked cycle that they are drawn in. If the **Include Delay Time** cycle based setting is checked then the signal will be driven as drawn after the clock edge. If the **Include Delay Time** cycle based setting is not checked the signals will be driven at the sensitive clock edge so it is not important to sketch the waveform at exact times for these kind of clocked signals. *Section 3.9 Diagram Settings Dialog - Language Specific Tabs* has more information on the **Include Delay Time** setting. For unlocked signals waveform events will be generated at the times they are drawn.

## 4.2 Drawing Waveforms and Bi-Directional Signals

The timing diagram editor is always in drawing mode. Waveforms are sketched by clicking the mouse button in the diagram window. The state buttons control which type of waveforms will be drawn next. The state buttons are the buttons with the waveforms drawn on their face: HIGH, LOW, TRIstate, VALid, INValid, WHI weak high, and WLO weak low. When a state button is activated, it is pushed in and colored red. The active state will be the type of waveform that is drawn next. Waveforms can also be edited by dragging and dropping edges, and by selecting segments and choosing another waveform state. The Timing Diagram Editor on-line menu provides in-depth information for the drawing environment.



### To Draw a Waveform:

- Click the type of state that you want to add in the group of 7 states on the right side of the *Signal Button Bar*.
- Click in the waveform section of the *Diagram* window to the right of the signal or bus name at the approximate time that you want the state transition to occur. This will place the transition in the waveform. Waveforms are built from left to right.
- Repeat the first two steps until you have completed the signal's waveform.

Signals with a direction of **output** or **internal** have black waveforms, and signals with a direction of **input** have blue waveforms. Bi-directional signals with a direction of **inout** will be drawn with mixed black and blue segments to indicate which segments will be driven by the transaction and which are inputs to the transaction.

By default all of the waveform segments on a bi-directional signals signal are assumed to have a direction of output and are colored black to indicate their direction. To change a segment to be an input segment (un-driven):

- Double-click on the input segment. This opens the *Edit Bus State* dialog.
- Uncheck the **Driven** check box. This indicates that the test bench does not drive this segment; this segment will be an input to the test bench.
- Click **OK** to close the dialog or use <Alt>-N or <Alt>-P (or the **Next** or **Previous** buttons) to edit other segments on the same signal. The segment for which you unchecked the driven flag should now be colored blue.

## 4.3 Driving Waveform States with Variables

Waveforms are normally driven to the drawn graphical state (high, low, tri-state, weak-high, and weak-low). However, waveforms with a graphical state of **valid** need to be driven to a distinct value during simulation. Using the *Edit Bus State* dialog you can hard code in a value, choose an existing variable in the timing diagram to drive the state, or in TestBench define a state variable. If a waveform segment is drawn with a graphical state other than **valid**, that graphical state will be used to drive the signal and any other state information entered through the *Edit Bus State* dialog will be ignored.

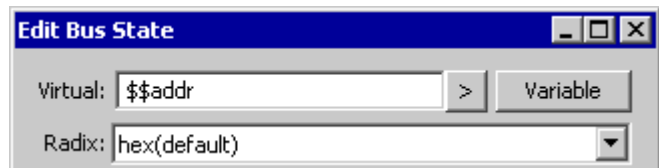
In TestBench, using variables to drive waveform states allows new values to be passed into the transaction each time it is called. This is convenient for timing diagrams that have data and address buses because each time the diagram is called new values can be passed into the timing diagram. **State variables** and **Diagram-level variables** can be used to drive a waveform state (see section *Section 3.3: Transaction Level Variables*). Parameter based variables should not be used to drive waveform states because their type is fixed to hold time values not state values.

TestBench state variables can be quickly defined in the *Edit Bus State* dialog by typing in a variable name that begins with two dollar signs like `$$addr`. State variables are automatically added to the parameter list for the transaction call. The type and size for these variables are determined by the signal that is being driven. Each time the transaction is called, a new state value can be passed into the variable. The same state variable can also be used on several signals and the maximum size will be determined by the min and max of all of the signals used. For example `$$addr` appears in `SIG0[3:0]` and `SIG1[12:9]`, then the `$$addr` will have a size of `[12:0]`.

Both TestBench and the Reactive TestBench option support Diagram-level variables. The type and size are controlled by the user during the declaration of the variables, so they require a little more setup work (see *Section 3.3 Transaction Level Variables*). Also diagram-level variables can be conditionally driven by different sources like samples and signal states within the timing diagram during simulation as well as being passed into diagram.


#### To edit the state of a valid signal segment:

- Double-click on the segment of the signal to open the *Edit Bus State* dialog.
- The **Virtual** edit box accepts values, variables, and Boolean equations that meet the format shown in *Appendix C: Language Independent Operators*. For example, `$$addr+@increment` is an acceptable equation for the **Virtual** edit box.



- To hard code a value, type the value in the **Virtual** edit box.
- To add a state variable, type the variable name using a \$\$ prefix into the **Virtual** edit box. For example, `$$data` might be the name of the variable for the value of data bus. This variable will appear in the timing diagrams apply call.
- To add a diagram-level variable:
  - Click the **Variables** button to open the *Select Variables* dialog. Double-click on the variable and click **OK** to close the dialog. The variable name with a @ prefix will be added to the Virtual box.

**OR**

- Click the Variables Menu Button  to display a list of variables that can be inserted into the equation.
- Click **OK** to close the dialog box.

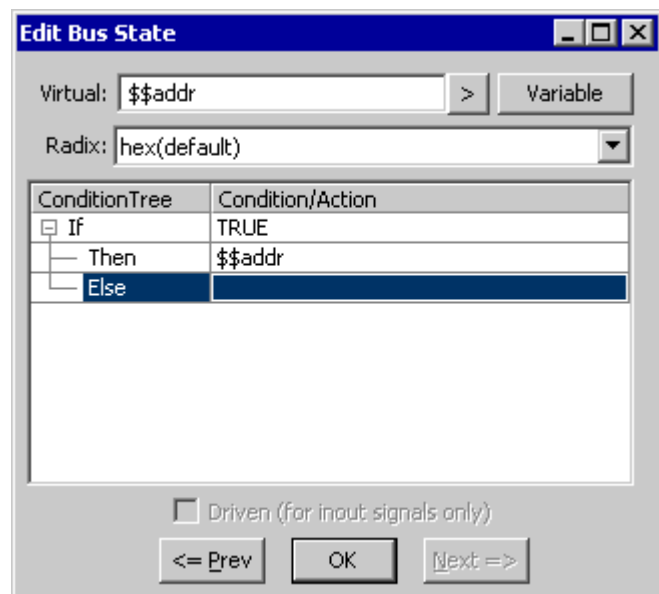
Note: State values can also be conditionally driven using the *Condition Tree* in the bottom of the *Edit Bus State* dialog. For more information see *Section 4.4: Driving Conditional State Values*.

## 4.4 Driving Conditional State Values

State values can be conditionally driven based on events and states that occur during simulation. The *Edit Bus State* dialog contains a *Condition Tree* that can be used to build conditional strings for the state value. If the state tree is not modified, the value will be unconditionally driven to the value in the **Virtual** edit box. The driven state can be made conditional by adding a condition to the State Condition tree.

#### To create a conditional drive for the state value:

- Double-click the segment that is to be conditionally driven to open the *Edit Bus State* dialog.
- Right-click on the **If** row and choose **Add Condition** from the context menu to open an edit box or double click in the **Condition/Action** column of the **If** row.



- Type the text for the condition. The condition must be written in the generated language of the transaction and it must equate to a Boolean equation when evaluated during simulation.
- Next, add the state values to the **Then** or **Else** rows by right-clicking choosing **Add Variable** or **Add State** menu option. Or double-click in the **Condition/Action** column and edit the state.
- Optional: Complex conditionals can be created using the **Add If...Then...Else** context menu. This option is available for any existing **Then** or **Else** row. Selecting this option causes a nested **If...Then...Else** to be added to the branch of tree that was selected.

Condition Tree	Condition/Action
[-] If	ADDR === 'hF0
[-] Then	\$\$addr
[-] Else If	ADDR === 'hAE
[-] Then	'h1
[-] Else	'h0

## 4.5 Adding Signals

Most signals will be automatically added by extracting the signal information from the model under test using the techniques that are discussed in *Section 3.2 Extracting MUT Ports into a Timing Diagram*. Signals can also be added manually by using the buttons on the *Signal Button Bar*. Certain types of signals like compare and internal signals are always added manually.

The generated bus-functional model can provide stimulus and monitor simulation outputs of the circuit that you are designing. In order to do this, the signals that will be exported by the bus-functional model have to match the signals that exist in your designs. If the signals in the timing diagrams are named the same as in your circuit model then the matching will be automatic. If the signal names do not match you will have to create a sub-project and use the *Signal and Ports* dialog to define the signal mapping as covered in *Section 2.3 Sub-Projects*.

Signals can be added manually by using the **Add Signal**, **Add Clock**, **Add Bus** and **Add Spacer** buttons on the signal button bar. The signal name, HDL type, and direction can be edited using the *Signal Properties* dialog.



### To add a Signal, Clock, Bus or Spacer:

- Click the appropriate button in the first group of four. This will add the Signal, Clock, Bus or Spacer to the timing diagram. Spacers are just for adding space to the diagram and do not generate code.
- If you added a signal, clock or bus, then double-click the name of the new object to open the *Signal Properties* dialog.
- Edit the **Name**. If the signal is to be hooked up to a signal in the HDL model, then use the same name.
- Edit the signal type using the **language Type** drop down list box in the bottom of the dialog.
- Edit the signal size using the **MSB** and **LSB** edit boxes. Clocks are always one bit wide.
- Edit the **Direction** using the drop down list box. The following directions are available:
  - **Output** indicates that the signal is output from the diagram.
  - **Input** indicates that the signal is what you expect the model under test to generate during simulation (these signals are inputs to the timing transactions, driven by the model under test). In the timing diagram, Sample parameters usually end on an input signal, indicating that the input signal should be checked for an expected value at that point on the signal.
  - **Inout** indicates that the signal is bi-directional (see *Section 4.2*). Inout signals contain driven and un-driven signal segments. Driven segments act like signals of type **output**.
  - **Internal** indicates that the signal will only be used internally to the diagram component.
- The **Clock** and **Edge/Level** specify the clocking signal for the waveform. In TestBencher, these will be automatically set by the *Project Wizard* options, however, you can pick a different system clock signal and edge using these controls. TestBencher users can also change the default clock using the *Diagram Properties* clock.

- For Clocks, the clock period, duty cycle, and clock offset can be changed by either clicking on the **Clock Properties** button or by Double-clicking on the clock waveform.

The default signal direction and language type for new signals can be set from the *Diagram Settings* dialog (see *Section 3.7: Diagram Settings Dialog - Overview* for more information).

## 4.6 Temporal Expressions for TestBench

Temporal expressions provide a method for looking for patterns of events within a transaction. TestBencher supports temporal expression through a direct text method described in this section and through graphical samples (see *Chapter 6: Transaction Samples*). Samples sometimes generate temporal expressions and other times they generate procedural code that produces the same functionality as a complex temporal expression. If a temporal expression is simple it is often better to describe it using the graphical samples because they are self-documenting. However if a temporal expression is complicated then the easiest method is just to type in the equation.

To add a temporal expression to a timing diagram, you will add a signal, set the type to temporal expression, and then type in the equation. TestBencher echoes the equation out to the proper section in the transaction code. The waveform of the temporal expression signal is ignored by TestBencher and should not be used as the end point for a sample or as the trigger for any parameters or markers in the diagram.

### To add a temporal expression:

- Press the **Add Signal** button to add a signal to the diagram.
- Double click on the signal name to open the *Signal Properties* dialog.
- Edit the name of the signal, this will be the name of the temporal expression.
- Choose the **te** radio button. This opens a different pane in the *Signal Properties* dialog where you can type in the temporal expression.
- Press the Ok button to close the dialog and then save the timing diagram file. This will generate the temporal expression in the transaction code.

## 4.7 Controlling the Triggering Order of Parameters

Edges on waveforms are responsible for triggering the markers and parameters that are attached to them. If more than one parameter or marker is attached to the same edge then the triggering order can be set using the *Edge Properties* dialog. By default the triggering order is the same as the order in which the objects were attached to the edge. The triggering order is especially important on edges that define the beginning and ending points of a marker loop, because the order determines whether the action occurs inside or outside of the loop.

Note: If a marker is relative to an edge, but not exactly on top of the edge, then order is based off of placement in the timing diagram and will not show up in the order dialog.

**To order Parameters and Markers attached to the same edge:**

- Double-click the edge that triggers the parameters and markers to open the *Edge Properties* dialog.
- Click the **Trigger Order** button to open the *Parameter and Marker Order* dialog.
- Drag and drop the rows to arrange the parameters and markers in the desired order. You can also use the arrow buttons on the right side of the dialog to move selected items up or down.
- If you need to review the properties of an item before setting the order, you can double-click the name of the object in the row to open the *Properties* dialog for that object.
- Click the **OK** button to close the *Parameter and Marker Order* dialog.
- Click the **OK** button to close the *Edge Properties* dialog.



**Displaying the order of parameter and markers in the timing diagram**

It may be useful to display the triggering order for parameters and markers in the timing diagram. This allows the order of execution to be determined at a glance, without opening the *Parameter and Marker Order* dialog. One of the display options for parameters and markers is **Name and Order**. This setting will display the order number for any parameter or marker with an order greater than 1, followed by the name of the parameter or marker. Note that the omission of the number one allows you to make this display setting the global default without displaying an order number when only one parameter or marker is triggered from an edge.

**To change the *Name and Order* display for a single marker or parameter:**

- Double-click the parameter or marker to open the *Parameter Properties* or *Marker Properties* dialog.
- Select the **Name and Order** option from the *Display Label* dropdown list.
- Click **OK** to close the dialog and apply the changes.

**To change the Global Settings for *Name and Order*:**

- Select the **Options > Drawing Preferences (Style Sheet)** menu option. This will open the *Drawing Preferences (Style Sheet)* dialog.
- Select the **Name and Order** selection from the *Parameter Display Label* dropdown list.
- Select the **Name and Order** selection from the *Marker Display Label* dropdown list.

Note: These two settings do not need to be the same. You may wish to set only one of these two as the global default.

- Click **OK** to close the dialog and apply the changes.

## 4.8 Sensitive Edges

The edges of signals can be made falling edge sensitive and rising edge sensitive using the check boxes in the *Signal Properties* dialog. Sensitive edges are usually placed on input signals and the code that gets generated causes the transaction to wait for the sensitive edge before continuing.

Sensitive edges cause wait statements to be inserted for that edge. These waits will block the clocking domain that contains the sensitive edge (see *Section 3.5: Transaction Architecture* for more detail on clocking domains).

**To enable sensitive edges on a signal:**

- Double-click the name of the signal that you want to watch for events on. This will open the *Signal Properties* dialog.

Note: Sequence Recognition watches the events on single bit signals only.

- Check the **Rising Edge Sensitive** checkbox or the **Falling Edge Sensitive** checkbox. Enabling both checkboxes will cause both rising and falling edges to be sensitive.
- Click the **OK** button to apply the changes and close the *Signal Properties* dialog.

Sensitive edges will have arrows instead of a line indicating the state transition.





# Chapter 5: Transaction Delays, Setups, and Holds

Timing diagrams can include graphical parameters like delays, setups, holds, and samples. These parameters generate transaction code that monitors and conditionally controls signal transitions. By combining and chaining together the parameters, you are graphically describing temporal expressions that will execute during simulation. In TestBench, Temporal Expressions can also be entered manually using a signal as described in *Section 4.6: Temporal Expressions for TestBench*.

This chapter will cover delays, setups, and holds that are parameters that perform actions between two signal transitions. Samples are placed on signal states (not transitions) and monitor the state of a signal. Samples are the main type of parameter used in TestBench timing diagrams and they are covered in detail in *Chapter 6: Transaction Samples*.

## Delays

Delays are used to specify a fixed time between signal transitions. The time between signal transitions can be a hard coded value or it can be a variable that is set during simulation. Delays can conditionally drive state values by triggering from a sample or by using an internal delay condition. The condition is checked after the delay is triggered, and before the delay time has been waited for. This is especially good for modeling control signals that go active after certain conditions in the transaction are met.

## Setups and Holds

Setups and Holds perform a check to determine if a signal is stable with respect to another signal. The graphical setup and hold parameters perform a one-time check between two signal transitions. A continuous check between two signals can also be created by using the properties for the signal.

## 5.1 Adding and Editing Parameters

Parameters are added by selecting a parameter button on the button bar, left clicking on the relative edge, and then right clicking on the second edge in the waveform window. After a parameter is added, its values can be edited by double-clicking on the parameter to open the *Parameter Properties* dialog. The properties for each parameter type are discussed in the section for that type.

To add a Delay, Setup, or Hold:

- Select the parameter button on the *Signal Button Bar* for the type of parameter you want to add.



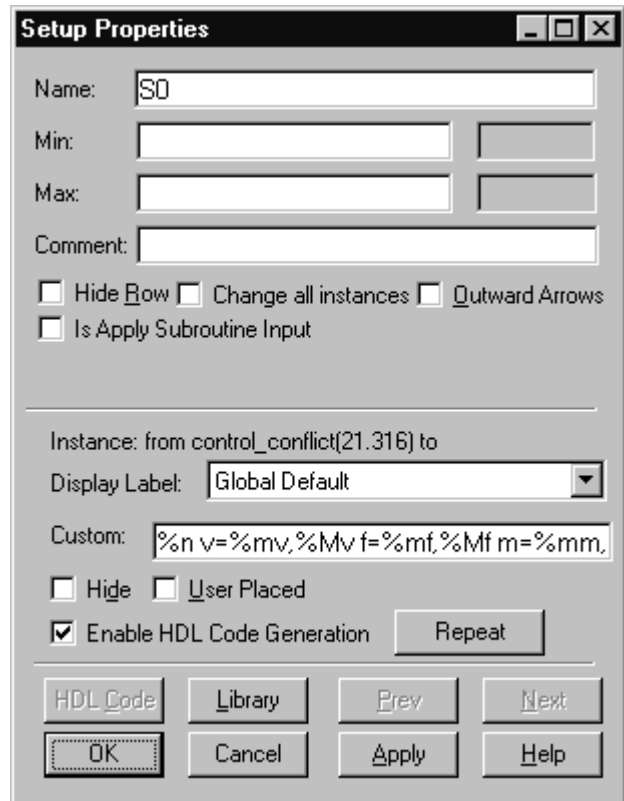
- Click on a transition to select it. For a delay this is the forcing transition. For a setup or hold this is the transition that will be monitored.
- Right-click on the second transition to add a parameter between the first and second transitions. For a delay this is the transition that will be moved. For a setup or hold this is the control signal.
- Double-click the name of the parameter to open the *Parameter Properties* dialog for that parameter and edit the properties of the parameter.

The *Parameter Properties* dialog has many settings that control how the parameter is displayed in the timing diagram and these features are covered in the Timing Diagram Editor on-line help *Section 4.4: Parameter Properties*. TestBench uses only a few controls for code generation and these are discussed below. A few additional controls are available for delays (discussed in *Section 5.2: Delays*) and samples (discussed in *Chapter 6: Transaction Samples*). The following controls are common to all parameters and are used in code generation:

- The **Name** edit box allows the user to specify the name of the parameter.

- The **Min** and **Max** edit boxes specify the minimum and maximum time for the parameter to execute. Each type of parameter handles the **Min** and **Max** values differently; for more information, see the sections on delays (Section 5.2), setups and holds (Section 5.4), and samples (Chapter 6).
- The **Is Apply Subroutine Input** checkbox, for TestBench, allows you to generate ports between the Component Model and the timing transaction with which to specify the values to use for the **Min** and **Max** settings of the parameter. If only one of the values is specified, then a port will only be made for that value. If there is no value specified for either setting, then a port will be made for the min value by default.
- The **Enable HDL Code Generation** checkbox allows you to turn the code generation for the parameter on and off without removing the parameter from the timing diagram. This checkbox must be checked in order to produce any HDL code for the parameter.

Note: The HDL code generation for all delays, samples, and markers in a timing diagram can be disabled through the *TestBench Diagram Settings* dialog. See Section 3.8: *Diagram Settings Dialog - General Tab* for more information on this feature.



## 5.2 Delays

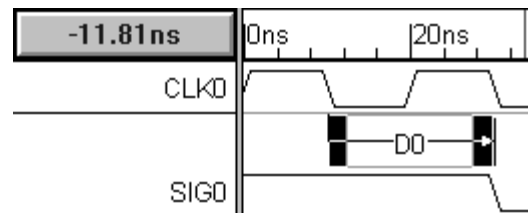
A delay specifies a fixed time between two signal transitions. Delays can also conditionally drive their second edge. In TestBench, the value for the delay time can be passed into the delay at simulation so that delays can be used to perform sweep tests to see when a circuit will fail.

The first edge (left most edge) that the delay is attached to is called the trigger edge. If the trigger edge is on a clocked signal then the delay will activate at the next clock edge if a level sensitive check of the trigger signal passes. If the trigger edge for the delay is on an unclocked signal, then the delay will activate when the signal transition occurs. If the level sensitive check fails, or if the unclocked trigger signal never transitions then the delay will not activate.

Once a delay is activated, then the delay process will wait for the amount of time (or clock cycles) specified in the min or max value of the parameter, and then drive the second edge. For more information, see Section 3.5 *Transaction Architecture*.

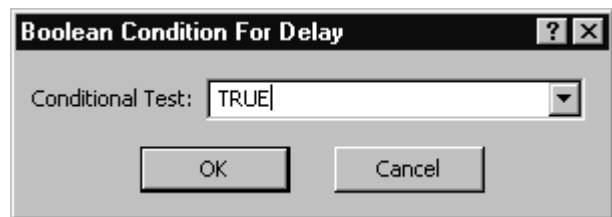
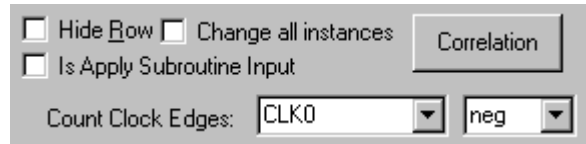
### To add a Delay to a Timing Diagram:

- Click the **Delay** button on the *Signal Button Bar*.
- Click on a transition to select it. This transition is the forcing transition.
- Right-click on the second transition to add a delay between the first and second transitions. This transition is the transition that will be delayed.
- Double-click on the delay to open the *Delay Properties* dialog. Most of the controls in the *Delay Properties* dialog were covered in Section 5.1 *Adding and Editing Parameters*.



The following controls are specific to delays:

- **Count Clock Edges** determines if the Min and Max settings are time or cycle based values. If the delay is *Unlocked* then the values are time. If a clock is specified then the values are numbers of clock cycles.
- **Min** and **Max** set the minimum and maximum time or number of clock cycles to be used for the delay. At simulation time only one value min, max, or typical (average of min & max) will be used. In TestBench, the *Diagram Settings* dialog (discussed in *Section 3.7: Diagram Settings Overview*), has the settings that determine which value will be used during simulation. If only one of the two settings has been given a value (min or max), the other setting will internally be given the same value.
- **HDL code** button opens the *Boolean Condition for Delay* dialog, that stores the condition that is checked before the delay drives the second edge. By default the condition is TRUE. You can type in the text for a new condition in the generated language. The condition can be any equation that evaluates to a TRUE or FALSE at simulation time. If the condition is not true after the triggering edge is detected, then the second edge will not be driven. The condition must be written in the generated language of the transaction. Note: If the condition is based on state values that occur during simulation, a graphical conditional delay can be constructed by triggering the delay from a sample parameter (see *Section 6.4: Samples Triggering a Delayed Transition or Another Delay*).

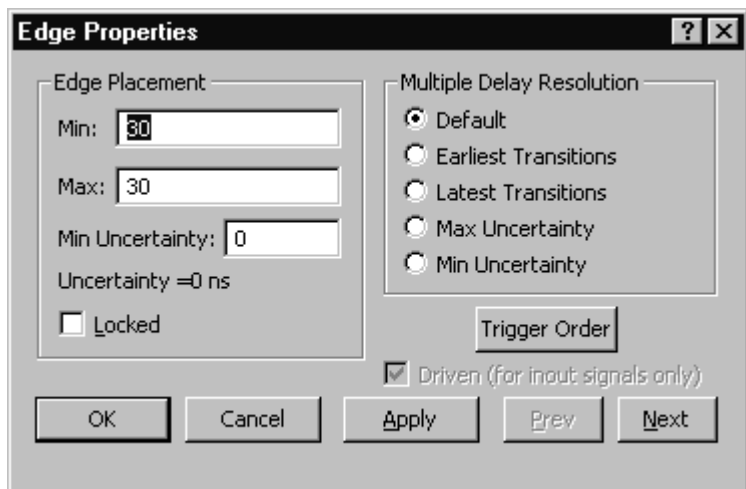


### 5.3 Resolving Multiple Delays

If the same edge is affected by multiple delays, there will be several possible ways for TestBench to resolve the actual delay. The value for the edge is calculated based on the **Multiple Delay Resolution** setting in the *Edge Properties* dialog. The default setting for the timing diagram is set in the **Options > Design Preferences** dialog.

To open the *Edge Properties* dialog:

- Double-click on the edge to open the *Edge Properties* dialog.
- In the **Multiple Delay Resolution** section, choose one of *Transition Settings*:
  - **Earliest Transitions** uses the delay that will place the edge as early in the diagram as possible.
  - **Latest Transitions** uses the delay that will place the edge as late in the diagram as possible.
  - **Max Uncertainty** and **Min Uncertainty** are not currently supported for TestBench Code generation.



## 5.4 Setups and Holds

Setups and Holds check timing requirements for a design. **Setups** are the minimum time necessary for a signal to be stable before a control signal transition. **Holds** are the minimum time that a signal must be stable after a control signal transition. Setups and Holds perform one check between two signal transitions. If the setup or hold fails then it outputs a warning in the simulation log file and prints the expected and actual values. If you want to perform a continuous check between two signals you can use the method described in *Section 5.5 Creating Continuous Setups and Holds*.

To create a setup or hold:

- Click the **Setup** or **Hold** button.
- Click on a transition to select it. This is the transition that will be monitored.
- Right-click on a second transition to add a setup or hold between the first and second transitions. This is the control signal.
- Double-click on the setup or hold to open the *Parameter Properties* dialog. Most of the controls in the *Parameter Properties* dialog were covered in *Section 5.1 Adding and Editing Parameters*.

The following controls are specific to setups and holds:

- The **Min** field sets the minimum time that the data transition can occur before a setup or after a hold on the control signal.
- The **Max** field sets the maximum time that the data transition can occur before or after the control transition. This field is optional and usually not specified for setups and holds.

If a time is specified for the **Max** field, then the data transition must occur between the **Min** and **Max** times.

- The **Outward Arrows** checkbox changes the direction that the arrows on the parameter are drawn. This does not affect code generation but it is a popular graphical feature.

## 5.5 Creating Continuous Setups and Holds

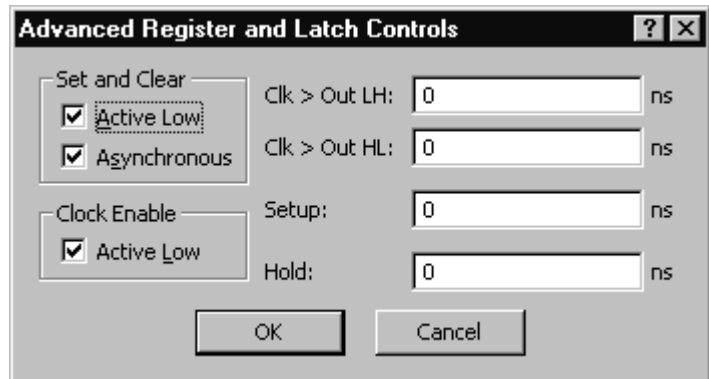
Continuous setups and holds can be created for any two signals in a timing diagram. A continuous setup or hold is created by using the *Advanced Register and Latch Controls* in the *Signal Properties* dialog. Continuous setups and holds can be useful in ensuring that a data signal remains stable long enough to be written at every clock edge, for example.

This feature is currently only supported in VHDL and Verilog, because it uses a feature called simulated signals. The simulated signals can be used to create registered and latch logic, so there are several of optional controls that you can choose to use. This feature is not supported in Waveformer Lite. To perform a continuous setup or hold just specify the clocking signals, edge type, and either the sample or hold value.

### To Create a Continuous Setup or Hold:

- Double-click on one of the signals to monitor (pick the non-clock if one of them is a clock). This opens the *Signal Properties* dialog.
- Select the clocking signal from the **Clock** drop-down list box. The clocking signal can be any clock or signal in the timing diagram.
- Select the type of edge or level triggering from the **Edge/Level** list box. For a Register circuit, choose **neg** for negative edge triggering, **pos** for positive edge triggering, or **both** for edge triggering. For a Latch circuit choose either low or high level latching.
- (OPTIONAL) The **Set**, **Clear**, and **Clock Enable** are optional signals that model the set, clear, and clock enable lines of the register or latch. If "Not Used" is chosen for a line, then that line is not modeled. These lines can be active low or high and synchronous or asynchronous depending on the settings in the *Advanced Register and Latch Controls* dialog.

- Click the **Advanced Register** button to open the *Advanced Register and Latch Controls* dialog that determines how this individual register is generated. The global defaults can be defined using the **Options > Simulation Preferences** menu. This dialog controls the following options:
  - **Setup**: Describes the time for which the input must be stable before the clocking event. If a **min/max** time pair is entered, **Setup** will use the **min** time. Any violations of this setup time will be reported to the simulation log file verilog.log, shown in the report window.
  - **Hold**: Describes the time for which the input must remain stable after the clocking event. If a **min/max** time pair is entered, **Hold** will use the **min** time. Any violations of this hold time will be reported to the simulation log file verilog.log, shown in the report window.
  - (OPTIONAL) **Clock to Out**: Describes the delay from the triggering of the clock signal to a change on the output edge.
  - (OPTIONAL) **Set and Clear Active Low**: If checked, the set and clear lines will control the output when they are low. If unchecked, then the set and clear lines will control the output when they are high.
  - (OPTIONAL) **Set and Clear Asynchronous**: If checked, then the set and clear lines will control the output anytime they are active. If unchecked, the model is synchronous and an active set or clear line does not affect the output until the next clock trigger event.
  - (OPTIONAL) **Clock Enable Active Low**: If checked, the clock will be enabled when the clock enable line is low. If unchecked, the clock will be enabled when the clock enable line is high.



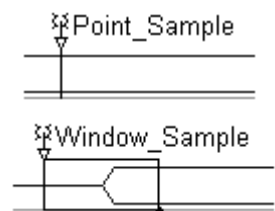


## Chapter 6: Transaction Samples

Samples generate the self-testing code within a transaction using either temporal expressions or procedural code that produces the same functionality as a complex temporal expression. In TestBench, temporal expressions can also be entered manually as described in *Section 4.6: Temporal Expressions for TestBench*. Samples are used to monitor the signal values coming back from the model under test. Samples can be run at a specific time, triggered from an event, or triggered from another sample. The value that is sampled can be exported to the top-level module. This could be used, for instance, to provide an input value for a state variable in another timing transaction or to determine if a specific timing transaction is to be executed or not. Samples can also be used to trigger a delay based on its success or failure. Below are the terms used to describe the different monitoring times and triggering events of a sample.

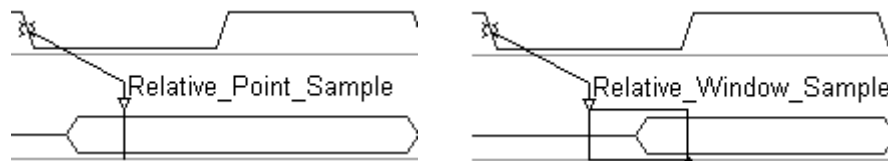
### Monitoring Time

Samples that monitor a signal at a specific time are called **Point Samples**. And samples that monitor a signal over an interval of time are called **Window Samples**. Window samples are useful for testing that the value of a given signal does not change over a specified time frame, or for verifying that the signal goes through a specified sequence of states. Window samples draw themselves with a box indicating the monitoring interval. If you need to sample over a large window and you do not want to display it graphically then you can use the *Multiplier* control in the *Code Generation Options* dialog described in *Section 6.2*.



### Triggering Process

Point or Window Samples can be either triggered at a specific time in the diagram (Absolute Sample) or they can be triggered by a transition on a signal or another sample (Relative Sample). The point and window samples shown in the above image are both absolute samples. The images below show relative samples that are triggered by a transition on a signal. If the triggering event is on a clocked signal, then at the next clock edge a level sensitive check will be performed and if it fails the sample will not execute. If the triggering event is on an unclocked signal, then if the transition does not occur during simulation then the sample will not execute.



### Check for Condition and Trigger an Action

The Sample's *Code Generation Options* dialog is used to define the condition the sample checks for and the actions it performs on the success and failure of the condition. *Section 6.3: Interpreting Sample Conditions and Blocking Points* describes how to control how the sample's condition is tested.


### Sample Variables and Files

Samples generate several diagram-level variables that can be accessed by other graphical elements in the diagram (*Section 6.5*). Sampled values can also be written out to a file (*Section 6.6*).

## 6.1 Adding a New Sample

To create a sample you will define the triggering event by how you draw the sample. The monitoring time or interval will be set using the *Samples Properties* dialog. The sample actions to take if it succeeds or fails will be set using the *Code Generation Options* dialog discussed in *Section 6.2*.

**To add a new sample:**

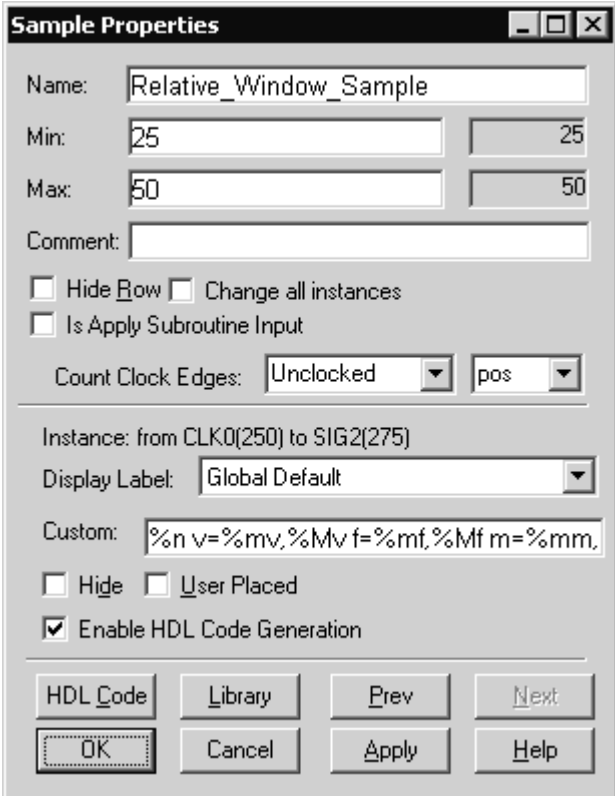
- Click the **Sample** button  on the *Signal Button Bar*.
- If you want the sample to be relative, then click the edge that you want the sample to be relative to.
- Right-click on the signal to be sampled. This will add the sample to the timing diagram. The exact time at which the sample is placed can be changed using the *Samples Properties* dialog discussed in the next step.

**To edit the monitoring time and properties of the sample:**

- Double-click the sample name to open the *Sample Properties* dialog.
- Type real time or clock cycles into the **Min** and **Max** edit box. If the min and max are different than the sample will be a Window Sample.

The *Sample Properties* dialog has many settings that control how the sample is displayed in the timing diagram and these features are covered in the Timing Diagram Editor on-line help *Section 4.4 Parameters Properties*. TestBench uses only a few controls for code generation and these are discussed below.

- The **Min** and **Max** edit boxes are used to specify the beginning and ending times or clock cycles for a sample window.
- Checking the **Is Apply Subroutine Input**, for TestBench, generates input ports to the timing transaction that can be used to specify the values to use for the min and max settings of the sample. (*Section 6.2* describes how the monitored value can be made to be an output port of the transaction.)
- Samples can be cycle-based instead of time-based. The **Count Clock Edges** settings allow a clocking signal and edge to be specified for the sample.
- The **Enable HDL Code Generation** checkbox must be checked for any code to be generated for the sample.
- The **HDL Code** button opens the *Code Generation Options* dialog that defines the actions of the sample. This is covered in *Section 6.2: Sample Condition and Actions*.



**Sample Properties**

Name: Relative\_Window\_Sample

Min: 25 25

Max: 50 50

Comment:

Hide Row  Change all instances

Is Apply Subroutine Input

Count Clock Edges: Unlocked pos

Instance: from CLK0(250) to SIG2(275)

Display Label: Global Default

Custom: %n v=%mv,%Mv f=%mf,%Mf m=%mm.

Hide  User Placed

Enable HDL Code Generation

HDL Code Library Prev Next

OK Cancel Apply Help

**6.2 Sample Condition and Actions**

When a sample is triggered, the sample will test for a condition and then perform an action based on the success or failure of the condition. Both the condition and the actions can be changed using the *Code Generation Options* dialog. You can choose from several predefined conditions and actions or directly enter the HDL code. The user defined condition and action usually call class methods that have been defined for the transaction (*Section 3.3: Transaction Level Variables*) or are short HDL expressions that make use of the internally generated sample variables (*Section 6.5: Using Sample Variables*).



**To define the condition and actions of a sample:**

- Double-click on the sample name to open the *Sample Properties* dialog.
- Click the **HDL Code** button in the lower left-hand side of the dialog to open the *Code Generation Options* dialog.
- The **If Condition** drop down list box controls what the sample checks for. Select one of the following conditions:
  - *Sample state matches*: If the monitored value matches the expected value the *Then Action* will be taken otherwise the *Else Action* will occur.
  - *Sample State doesn't match*: If the monitored value matches the expected value the *Else Action* will be taken otherwise the *Then Action* will occur.
  - *User-defined condition*: Directly enter the HDL code to execute see *Section 6.5: Using Sample Variables*

- The **Then Action** and **Else Action** drop down list boxes control which actions are taken on the success or failure of the sample condition. Select one of the following actions:

- *Do nothing*: take no action if this branch is executed.
- *Display Message*: Display a message in the simulation log using the severity level defined by the radio buttons below the action.
- *Restart Diagram*: Resets and restarts the transaction execution.
- *End Diagram (set status to Done)*: Ends execution of this particular transaction. The bus-functional model will continue to execute as if this transaction had normally ended.
- *Pause Simulation (Verilog only)*: Stops the entire simulation.
- *Do Delayed Transition*: Creates a delayed state transition (see *Section 6.4: Samples Trigger Delayed Transition or Another Sample*) that triggers based on the results of the *If Condition*.
- *Trigger Sample*: creates a triggered sample (see *Section 6.4: Samples Trigger Delayed Transition or Another Sample*) that will fire based upon the results of the *If Condition*.

- *Break Loop*: stops the loop that immediately surrounds the sample.
- *Continue Loop*: returns to the beginning of the loop immediately surrounding the sample, skipping the last part of the loop
- *User-defined action (enter below)*: lets the user directly enter VHDL or Verilog code for the action into the edit box below the action drop-down list box. See *Section 6.5: Using Sample Variables* for more information.

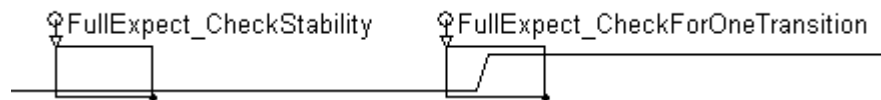
- When the **Full Expect** checkbox is checked, every transition that occurs within the window range is checked. If the option is not enabled, the sample will check for the condition to be true (*Simple Expect*), or the event to occur (*Restricted Expect*) depending on how the waveform is drawn. See *Section 6.3: Interpreting Sample Conditions and Blocking Points* for more information.
- The **Multiplier** property extends the window of time for a sample. The difference between the **min** and **max** values will be multiplied by the value of the multiplier to determine the length of the sample. This also provides a method to indirectly specify a timeout for a sample. Since this method of extending the sample window does not appear graphically it can be used for very large windows that would not be very pretty to look at.
- In TestBench, the **Enable Variable** control enables the sampled value to be output to a file in a spreadsheet-like format or stored in a variable. Set to *Then* if you want the data to be stored if the sample condition succeeds or *Else* if you want the data to be stored if the sample fails. Select *Always* if you want the data to be stored regardless of the condition. See *Section 6.6 Storing Sample Values in User Defined Variables* for more information.
- In TestBench, the **Store Sampled Value As Subroutine Output** checkbox creates an output port to the transaction and when the transaction terminates it passes the sampled value out to the port. How this is implemented depends on the generation language:
  - In Verilog, TestBench will automatically create a variable in the top-level project that the transaction is stored. The variable is named *transactionName\_sampleName* and at the end of the transaction the sample value will be passed out to this variable.
  - In all the other languages, you must create a variable in the calling project that has the same type as the signal that is being sampled. This variable is then passed into the transaction apply call. The variable will be set during the transaction execution.
- The **Blocking** option determines whether or not the triggering process or sequence of the sample will wait for the sample to complete before continuing execution. If the option is enabled, the triggering process will trigger the sample and then wait until the sample process is complete before continuing execution. Otherwise, the two processes will execute concurrently once the sample is triggered. Samples, by default, are non-blocking. *Section 6.3: Interpreting Sample Conditions and Blocking Points* discusses this feature.

### 6.3 Interpreting Sample Conditions and Blocking Points

The drawn waveform and the **Full Expect** check box in the *Code Generation Options* dialog determine when Windowed Samples execute an action. The sample can also be made to block other graphical elements in the diagram by using the **Blocking** check box in the *Code Generation Options* dialog. If **Blocking** is enabled then other elements in the same clocking domain as the sample will be paused until the sample condition executes an action. If **Blocking** is disabled, the other graphical elements will continue to function regardless of whether the sample condition is satisfied. Below are some examples of different types of samples.

#### Full Expect Samples

If the **Full Expect** box is checked then the sample will wait until the end of the sampling window to determine if all conditions were met. This is called a **Full Expect** sample. This type of sample will test every state transition drawn within the sample window. For instance, if a Full Expect sample has a condition of *Sample state matches*, it will test that every expected transition matches what is drawn in the sample window. The appropriate Full Expect sample action is executed at the end of the sample window. This is indicated visually by the dot at the end of the sampling window.



If the sample is clocked then the value of the waveform will be sampled at each clock edge in the sample window.

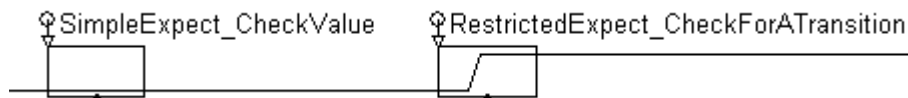
### Simple Expect and Restricted Expect Samples

Simple and Restricted Expect samples are created when the **Full Expect** setting is disabled. The manner in which the expected waveform is drawn determines whether the sample is a Simple Expect or a Restricted Expect sample.

A sample that is drawn above a stable section of a waveform will test for the condition to be true at any time during the sample window. This is called a **Simple Expect** sample. A sample that is drawn over a stable low waveform, for example, will watch for the condition to be true at any time during the sample window. This means that a Simple Expect sample will trigger its action at the beginning of the sample window if the expected state matches the driven state during simulation.

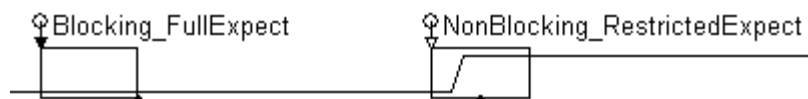
If a sample is drawn above a waveform with one or more transitions, the sample will test for each transition in the window. This is called a **Restricted Expect** sample. A Restricted Expect sample will test the first transition to see if it matches the first transition in the drawn waveform. If the transition matches, then the next transition is evaluated in the same manner. Once all of the transitions are found, the sample condition will pass and execute the *Then* action. If a wrong transition is found, the condition will immediately fail and execute the *Else* action. If not enough transitions are detected then the sample times out at the end of the window and the *Else* action is executed.

Both **Simple Expect** and **Restricted Expect** samples are drawn with a dot in the middle of the sample window to indicate that the trigger time is determined at simulation time and can occur before the end of the window.



### Blocking Sample

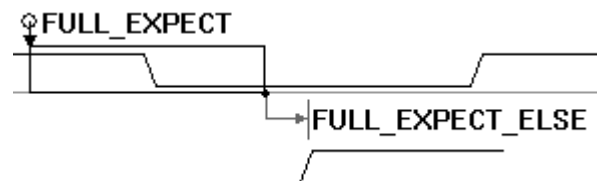
The **Blocking** setting in the *Code Generation Options* dialog controls whether or not a sample blocks other constructs in the same process. Samples with this setting enabled prevent other constructs from proceeding until the sample condition triggers an action. *Section 3.5: Transaction Architecture* discusses how blocking samples will pause a portion of the timing transaction. If a sample has **Blocking** enabled, then the clocking domain will pause until the *Then* or *Else* action is executed. Blocking samples are shown visually with a solid arrowhead. Non-blocking samples display with a hollow arrowhead.



## 6.4 Samples Triggering a Delayed Transition or Another Sample



Samples can be used to trigger delayed state transitions or other samples. These actions are performed by use of triggered delays and samples. These constructs are triggered when the appropriate action is called for the *Then* or *Else* segment of a sample. Several samples can be chained together to test for a complex set of conditions.

If a sample is triggering a delay, then that sample conditionally controls the signal transition. This is especially useful if several conditions must be met prior to a transition on a control signal. An alternate, non-graphical method for conditionally triggering transitions is discussed in *Section 5.2: Delays*.



There are two different methods you can use to add a parameter to a sample. The recommended way is to add a parameter that is relative to a sample. This is the fastest way to add samples and delays to the *Then* and *Else* actions. The other method is to use the Sample's *Code Generation Options* dialog. Either method will set one of the sample actions to **Do Delayed Transition** or **Trigger Sample** and attach a graphical parameter to the sample.

**Method 1: (Recommended) Add a Delay or Sample to relative to a Sample**

- Click the **Delay** button  or the **Sample** button  in the *Diagram* window.
- Click on the sample name to select the sample that will trigger the new parameter.
- For delays, right-click on the state transition that you want to be conditionally delayed. This will open a context menu. Choose either **Then Sample Delay** or **Else Sample Delay** to create the new conditional delay.
- For samples, right-click on the waveform you want to sample. This will open a context menu. Choose either **Then Triggered Sample** or **Else Triggered Sample** to create a chained sample.

**Method 2: Using the *Code Generation Options* dialog to add a triggered delay or sample. Note only add one delay or sample at a time:**

- Double-click the name of a sample to open the *Sample Properties* dialog.
- Click the **HDL Code** button to open the *Code Generation Options* dialog.
- For Delays, choose **Do Delayed Transition** from the **Then Action** or **Else Action** drop down list box.
- For Samples, choose **Trigger Sample** from the **Then Action** or **Else Action** drop down list box.
- Click the **OK** button to close the *Code Generation Options* dialog.

Note: When you close the *Code Generation Options* dialog you will enter a special select mode. While you are in this mode, the *Sample Properties* dialog will disappear. When you exit the select mode, the *Sample Properties* dialog will reappear.

- Right-click the state transition that will be delayed or the waveform that will be sampled. This will add the delay or sample to the diagram.

The delay ending position can be moved to other signal transitions by selecting the delay then dragging and dropping the right handle of the delay to the new transition. Triggered samples can be edited just like regular samples.

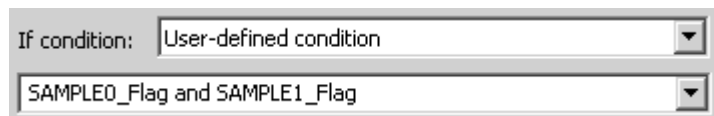
## 6.5 Using Sample Variables

Two sample variables are automatically generated for each sample: *sampleName\_Flag* and *sampleName*. The *sampleName\_Flag* variable is a Boolean flag that indicates whether the sample condition was true or false. And *sampleName* is a state variable that contains the value of the sampled signal at the time the sample's condition was met or timed out. These are diagram-level variables and can be referenced anywhere in the timing diagram including other sample's actions and conditions, HDL Code Markers, and Class Methods

In TestBencher, the sample value, *sampleName*, can also be exported from the transaction by checking the **Store Sampled Value as Subroutine Output** checkbox in the *Code Generation Options* dialog as described in *Section 6.2: Sample Condition and Actions*.

**Example of using Sample Flag Conditions**

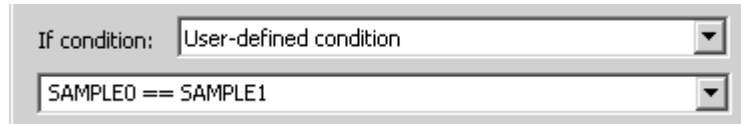
It is frequently desirable to define a sample condition in terms of previously executed samples. For example, you might wish to execute an action if two different previous samples were both true. This can be accomplished by writing HDL code accessing the flag variables that store information about previously executed samples. Assume you have a diagram with three samples (SAMPLE0, SAMPLE1, and SAMPLE2) where the first two samples test the values of two signals. To make SAMPLE2 true if both SAMPLE0 and SAMPLE1 are true, you would enter the *User-Defined Condition* of **SAMPLE0\_Flag and SAMPLE1\_Flag**.



Assume you have a diagram with three samples (SAMPLE0, SAMPLE1, and SAMPLE2) where the first two samples test the values of two signals. To make SAMPLE2 true if both SAMPLE0 and SAMPLE1 are true, you would enter the *User-Defined Condition* of **SAMPLE0\_Flag and SAMPLE1\_Flag**.

### Example of using Sample Values in the Diagram

You can also use the sample values to build user-defined conditions for samples. For example, to test that the value sampled by SAMPLE0 is equal to the value sampled by SAMPLE1, enter the following *User-Defined Condition* for SAMPLE2.



**Note:** the types of the signals sampled by SAMPLE0 and SAMPLE1 must be the same, or you will get a type mismatch error when you compile your test bench.

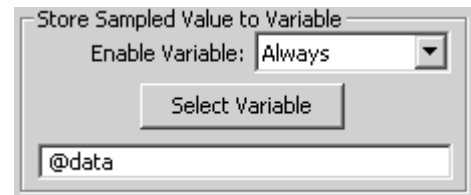
## 6.6 Storing Sample Values in User Defined Variables

In addition to the automatically created sample variables, a sampled value can be stored in a user-defined variable. The stored sample value can be used in the diagram to define a marker loop expression or a conditional delay equation.

In TestBench, the sampled value can be used to drive another signal or stored in a file. By enabling and selecting a variable, the sampled value will be stored each time the sample completes. File Output variables write the sampled value to the specified file when the transaction completes.

To store a sample value in a user defined variable:

- Open the *Sample Properties* dialog by double-clicking the name of the sample.
- Click the **HDL Code** button to open the *Code Generation Options* dialog.
- Select the desired enable option from the **Enable Variable** drop-down. This option will determine the condition under which the sampled value will be stored in the variable. This option can be set to **Always**, **Never**, **Then**, and **Else**. The **Then** and **Else** options specify that the data will be stored only if the *Then Action* or *Else Action* is executed, respectively.
- Click the **Select Variable** button to open the *Select Variable* dialog.
- Click a field name or the variable name in the **Name** column of the selection tree to select a variable. Note that default *Index*, *MSB*, and *LSB* values are defined.



**Note:** For any given transaction, only one sample can output to a specific column in the file. If more than one sample is using the same field name within the same timing diagram, only the last instance to occur during simulation will output to the column.

Any item that cannot be edited will have a gray background in the tree (except the name). To edit a value in the tree:

- Double-click the text that needs to be edited.
- Edit the text
- Click the **Insert Into Equation** button to set the variable property for the sample.
- Click **Close** to close the dialog. This will set the variable property for the sample.

You will be able to change the variable or field name at any time by opening the *Code Generation Options* dialog for the sample and repeating this process.



# Chapter 7: Transaction Markers


Markers can be added to timing diagrams to specify specific actions to be taken by the transaction during execution. These actions can include identifying the end of a transaction, creating loops in the transaction, executing HDL code, blocking, and pausing the simulation.

Markers are triggered either by the unlocked process or by the clocked process of edge they are relative to. Loop and Wait Until markers act on their triggering process so it is important when using these types of markers to setup the triggering event correctly (*Section 3.5 Transaction Architecture*).

## 7.1 Adding a Marker to a Diagram

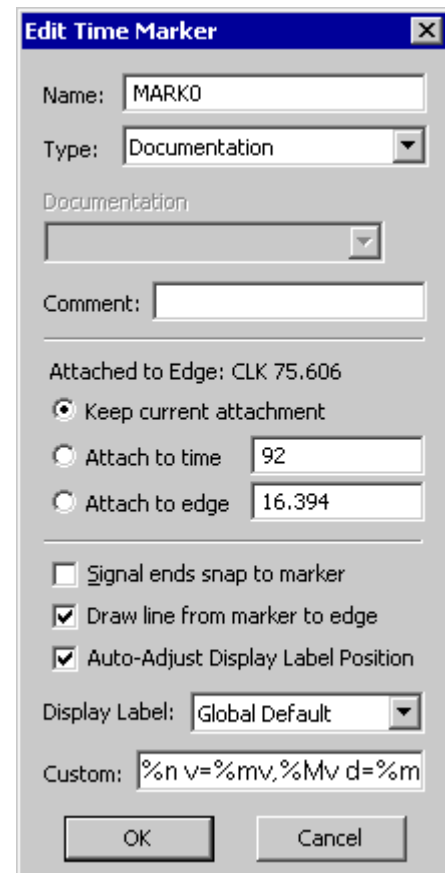
As with samples, markers can be absolute or relative. An absolute marker is attached to a specific time, while a relative marker is attached to a specific edge. Relative markers will be triggered by the process associated with the clocking domain (See *Section 3.5 Transaction Architecture*). Double clicking the marker opens the *Edit Time Marker* dialog that is used to control the code generation options for the marker.

### To place a marker in a diagram:

- Click the **Maker** button  on the *Signal Button Bar*.
- If you want the marker to be relative, then select the edge that you want the marker to be relative to.
- Right-click in the *Diagram* window to place the marker. This will add a documentation marker to the diagram window.

### To Edit a Marker:

- Double-click on the marker line or on the marker name to open the *Edit Time Marker* dialog.
- The **Marker Type** controls the function of the markers. The rest of the chapter is devoted to the details of each of the marker types:
  - *End Diagram* causes the transaction to terminate at that point.
  - *Pause Simulation (Verilog only)* stops the entire simulation.
  - *While Loop, For Loop, Repeat Loop, Loop End, and Exit Loop When* are used to create loops for a single process in the transaction.
  - *HDL Code* marker inserts user written source code.
  - *Wait Until* causes the process that triggers the marker to block until the condition becomes true.
  - *Semaphore* used to define critical regions in a transaction.
  - *Pipeline Boundary* is used to specify a pipeline region in a transactor. This is used when multiple instances of a transactor are running in parallel.
  - *Documentation* markers are used to annotate the timing diagram.
  - *Time Break Markers* are used to hide sections of the timing diagram but do not cause code to be generated.
- The **Attach to time** controls are used to change the attachment or placement of a marker.
  - To move a relative marker to the exact edge time, type **0** into the **Attach to edge** edit box.



- To attach to a new edge, check the **Attach to edge** radio button and click **OK** to close the dialog and enter into an edge selection mode. As you move the cursor a green bar will hop to the closest edge. Left click on the edge that you want to attach the marker.
- To attach to a new time, check the **Attach to time** radio button and enter a time into the edit box.
- The **Snap Signal Ends to Marker** feature is generally for documentation purposes - the ends of all drawn waveforms will be attached to the marker and move with it.
- **Draw Line From Marker To Edge** will cause a dotted line to be drawn for markers that are attached to an edge. This is a nice feature to be able to quickly see that the marker is attached to an edge and not a time, and also which edge the marker is attached to.
- **Auto Adjust Display Label Position** allows the diagram editor to automatically adjust the position of the marker display to ensure that it does not over-write or get overwritten by other items in the diagram window.
- Click the **OK** button to close the dialog.

## 7.2 End Diagram Markers

End Diagram Markers are used to indicate the execution end of a timing diagram. These markers are useful for extending a transaction past the last drawn waveform. In TestBencher End Diagram markers are especially useful for syncing up multiple timing diagrams that share the same clock. For example, it is convenient to place an End Diagram Marker at the exact ending transition of a clock cycle.

If there are no End Diagram markers then the longest non-clock signal will determine the end of the timing diagram. If there is more than one End Diagram Marker then the earliest one will determine the end of the timing diagram. End Diagram Markers are displayed using a purple line.

### To modify a time marker to be an end diagram marker:

- Add a marker and then double-click on the marker to open the *Edit Time Marker* dialog.
- Select **End Diagram** from the *Marker Type* drop down list.
- If the Marker is not located at the exact location or attachment that is needed, then use the **Attach to** radio buttons to move the marker. In this example the edge is attached to the CLK0 edge at exactly time 250ns.
- (OPTIONAL) Choosing **Type** from the **Display Label** control causes the marker to display the words *End Diagram* instead of the marker name.
- Click the **OK** button to close the dialog.

## 7.3 Pause Simulation Marker (Verilog Only)

A Pause Simulation marker will pause the entire simulation when it reaches the marker. This provides a graphical breakpoint. While the diagram is paused you can check variables and signal states. When you are done, use your simulator run button or run command to continue the simulation.

This feature is not supported in VHDL because there is no language construct that can stop the simulator. However, some simulators can be configured to pause on assert failures. If your simulator supports this feature, then you can use an HDL code marker to place an assert in the timing diagram.



To specify a **Pause Simulation** marker:

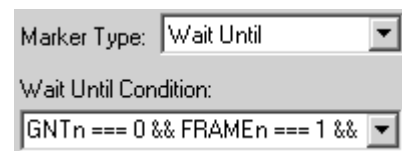
- Add a marker to the timing diagram. The exact placement or attachment does not matter because the marker will pause all processes in the entire model.
- Double-click on the marker to will open the *Edit Time Marker* dialog.
- From the *Marker Type* drop-down list, choose **Pause Simulation (Verilog only)**.
- Click **OK** to close the *Edit Time Marker* dialog.

## 7.4 Wait Until Marker

Wait Until markers provide a mechanism for indefinitely pausing the execution of one clocked process within a transaction. This type of marker pauses the transaction until its condition becomes true. Blocking samples also pause the execution of a process, but they have a time out built into the window and multiplier settings. Wait Until markers will not time out. The process that gets paused will be the triggering process of the Marker (see *Section 3.5: Transaction Architecture*).

To specify a **Wait Until** condition:

- Add a marker that is attached to some signal transition in the diagram.
- Double-click on the marker to open the *Edit Time Marker* dialog.
- From the *Marker Type* drop-down list, choose **Wait Until**. This relative marker it will pause the execution of all signals and graphical elements that are relative to the same signal and edge type.
- In the **Wait Until Condition** edit box, enter a condition. The condition can be any equation in the generation language that evaluates to a TRUE or FALSE at simulation time.
- Click **OK** to close the *Edit Time Marker* dialog.



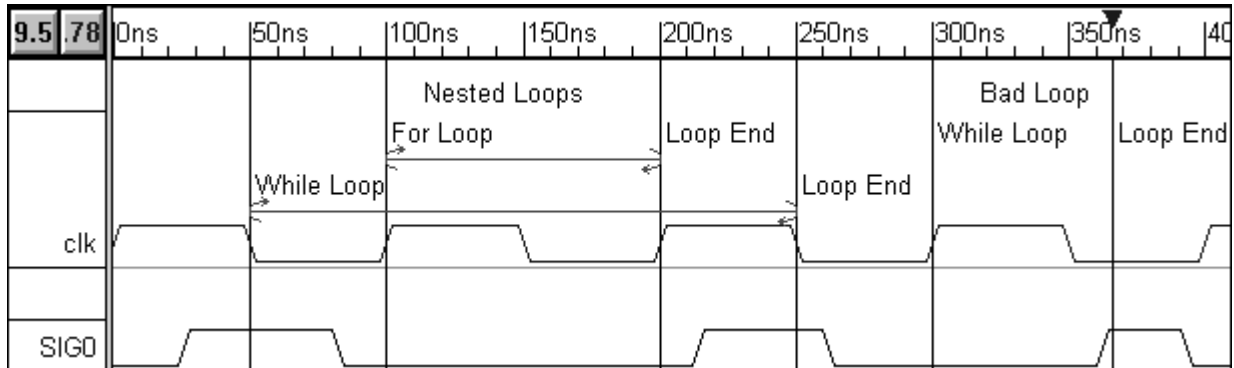
When this transaction is applied, it will now pause execution (of the transaction, not the simulation) at the time that the marker is placed until the specified condition has occurred.

## 7.5 Loop Markers

Loop markers are used to create sections in the transaction that are repeated during simulation. For example if you were designing a burst read transaction that would need to determine at simulation time the number of cycles needed to complete the read cycle, then you could use a while loop. The transaction could be setup to continuously loop until a certain ending condition was met. TestBencher supports while loops, for loops, and repeat loops. The *Exit Loop When* marker can be used to terminate a loop in the middle of a cycle. Loop Markers can also be used with samples whose *Break Loop* and *Continue Loop* actions affect the operation of the loop.

The same process must trigger both the beginning loop marker and the end loop marker (see *Section 3.5 Transaction Architecture*). For clocked transactions, this means that the begin marker and the end marker need to be attached to the same edge type of a given signal. When TestBencher recognizes the beginning and ending of a loop it will draw a green loop line between the markers. In the example below the bad loop will not work because the while marker is triggered by the rising edge clk process while the loop end is triggered by the unlocked process.

Often the signal edges that trigger the beginning and end of loop markers are also triggering other markers and samples. When several graphical elements are triggered off of the same edge then the order determines whether the other graphical elements occur inside or outside of the loop. The order is set by double-clicking on the edge and using the *Edge Properties* dialog (see *Section 4.7: Controlling the Triggering Order of Parameters*).



To add a loop to a timing diagram:

- Add two markers to the timing diagram. Both should be relative to the same signal and edge type, or both should be absolute time markers.
- Double-click on the marker on the left to open the *Edit Time Marker* dialog. Choose one of the following loop types and define the beginning of the loop:
  - **While Loop** marker when matched with an *End Loop* marker will execute continuously over a sequence of test vectors either forever or until a defined condition is met. The condition can be any equation in the generation language that evaluates to a TRUE or FALSE at simulation time.
  - **For loop** marker will execute for a specified number of iterations. The *Index* variable will be automatically created. Each loop the index variable will be incremented by the *Inc* number. The loop will end when the index becomes greater than the *End* number.
  - **Repeat Loop** marker will execute for a specified number of iterations.
- Click **OK** to close the dialog.
- Double-click on the marker on the right to open the *Edit Time Marker* dialog. Choose the **Loop End** marker type.
- Click the **OK** button to close the dialog. If the markers are triggered by the same process, TestBench will draw a loop line between the markers. If there is no loop line then check the attachments of each marker.

Type:

Loop while condition:

Marker Type:

Index:  Inc:

Begin:  End:

Marker Type:

Repeat number:

### Exit Loop When

The *Exit Loop When* marker will terminate the inner most loop that graphically surrounds the *Exit Loop When* marker and that is triggered off of the same process. The condition can be any equation in the generation language that evaluates to a TRUE or FALSE at simulation time.

Marker Type:

Exit when condition:

## 7.6 HDL Code Markers

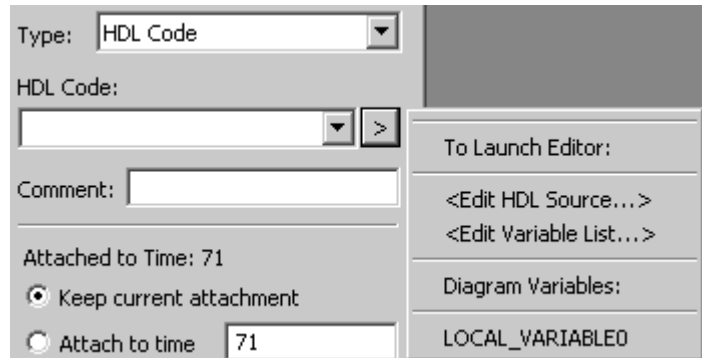
HDL code markers are used to make calculations and execute code that is not represented graphically. HDL code markers have a limited amount of space for typing, so it is usually just used to type in the name of a function to call. The code box accepts direct HDL code in the transaction generation language. You can make calls to class methods (*Section 8.7: Class Methods*), library subroutines (*2.8 Libraries and Use Clauses*), or insert any code that is valid within the context of a process (VHDL) or method (TestBuilder).

**To add an HDL Code marker:**

- Add a marker to the diagram and double-click on the marker to open the *Edit Time Marker* dialog.
- From the **Marker Type** drop-down list, choose **HDL Code**.
- Type in the source code into the **HDL Code** edit box.

**OR**

- Select the **<Edit HDL Source...>** menu option to enter multiple lines of source code for this type of marker.
- Click **OK** to close the dialog.

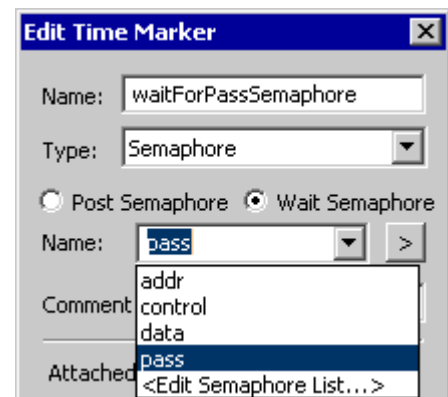
**7.7 Semaphore Markers**

Semaphore markers are used to secure critical regions during a transaction. Semaphore markers can be created for transactions that are part of a project that contains semaphores. See *Section 8.11: Semaphores* for information about creating Semaphores. There are two types of Semaphore Marker - **wait** and **post**. A **Wait Semaphore Marker** waits for the semaphore variable to be free, then takes 'possession' of the semaphore and enters the critical region. The **Post Semaphore Marker** defines the end of the critical region and causes the release of the semaphore.

**To add a Semaphore Marker:**

- Add a marker to the diagram and double-click on the marker to open the *Edit Time Marker* dialog.
- From the **Marker Type** drop-down list, choose **Semaphore**.
- Select either the **Post** or the **Wait** semaphore marker type.
- Enter the **Semaphore Name**. There are two ways to specify the semaphore name. The name can be typed into the *Semaphore Name* history list or selected from the *Variable Button Menu*.

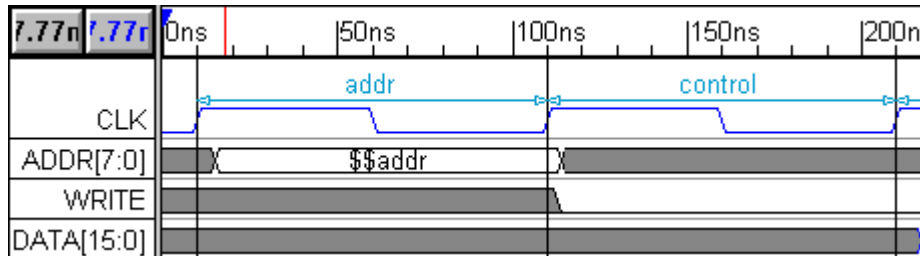
New Semaphores can be created in the *Marker Dialog*. Either enter the name of the new Semaphore in the *Semaphore Name* combo or select the **<Edit Semaphore List...>** menu item from the *Variable Button Menu*. The **<Edit Semaphore List...>** menu item will launch the *Classes and Variables* dialog for projects. New Semaphores entered directly in the combo will be given an initial value of 1.

**7.8 Pipeline Boundary Markers**

Pipeline Boundary Markers are used for pipelined transactors. These markers will automatically create the semaphore needed to handle a critical region (if the semaphore does not already exist in the project). Loop markers can be placed within the critical region - but only if the complete loop is inside the region. In other words, the loop cannot overlap the pipeline phase - it can be entirely inside the phase or around the phase.

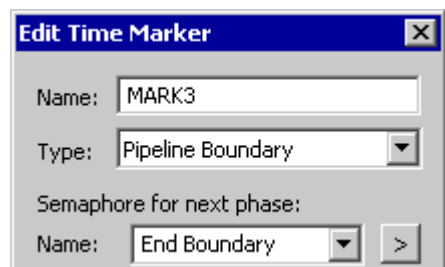
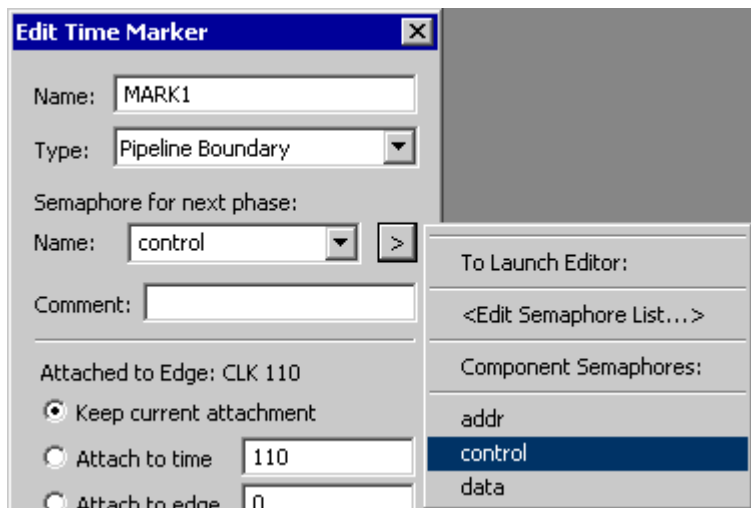
The **Instance Count** of a diagram represents the number of instances of the transaction that are created during simulation. TestBench will automatically set the **Instance Count** for the diagram during generation. This setting can be overridden, but it is not normally necessary to do this. Should this setting be overridden, the pipeline depth must be less than or equal to the **Instance Count** of the timing diagram. The Instance Count is set in the *Diagram Settings* dialog - see *Section 3.8: Diagram Settings: General Tab* for more information on setting the **Instance Count** for a diagram.

The **Semaphore Name** specified in the Marker Properties dialog represents the next pipeline phase and is displayed in the diagram between the start and end boundaries of the phase. The next Pipeline Boundary marker in the diagram (either the start of the next phase, an **End Boundary** pipeline marker or an **End Diagram** marker) represents the end of the phase. In the diagram below, for instance, the first marker is a Pipeline Boundary marker with **addr** selected as the semaphore. This phase is ended by the **control** Pipeline Boundary marker - which also started the control phase.



### To Create a New Pipeline Phase:

- Place a marker at the time or edge that the phase should begin. Double-click the marker to open the *Marker Properties* dialog.
- Enter the name of the phase in the **Semaphore For Next Phase** combo. If the name is not already in the project Semaphore list, then a new Semaphore will be created with **Initial Value** of 1 and placed in the project.
- Place a second marker at the time or edge that will end the pipeline phase. This marker must be one of the following three types (otherwise a start phase without one of these to end it will produce an error):
  - 1) **End Boundary**: Place a **Pipeline Boundary Marker** and select **End Boundary** from the **Name** drop down. Use this to end the last phase in the transaction when there are other events after the last phase.
  - 2) **Pipeline Boundary**: Place a **Pipeline Boundary** marker to end this phase and begin the next. This method should be used when two pipeline phases are consecutive.
  - 3) **End Diagram**: An **End Diagram** marker will automatically end the phase. This is a nice feature to use when the last pipeline phase is the last event in the diagram.



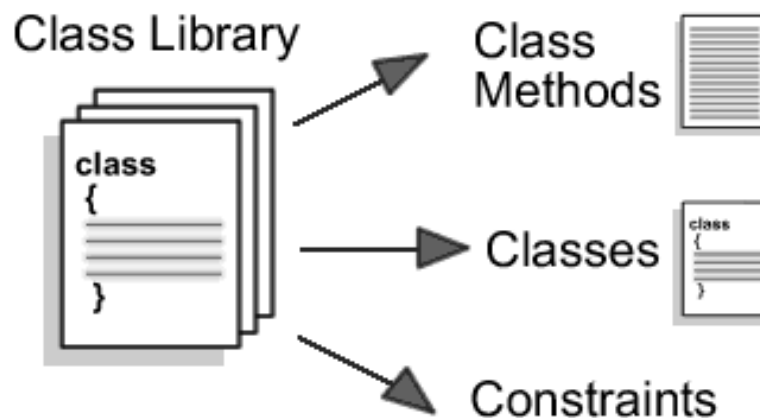
## 7.9 Documentation and Time Break Markers

Documentation and Time Break markers can be used to split the visual image of the timing diagram for whatever purpose may be needed. For example, it may be useful to visually highlight a point of change in the timing diagram. The time break markers can also hide sections of the timing diagram. These markers generate a comment line in the source code for the transaction. If **Verbose Markers** is checked in the *Diagram Settings* dialog a message is displayed during simulation (see *Section 3.8: Diagram Settings Dialog - General Tab*).

## Chapter 8: Classes and Variables

In addition to the graphical elements of a project, TestBench also supports generation of user-defined classes and variables. Classes, which are stored in class libraries, contain data fields and methods. These elements let you pass data around and compose algorithmic functions that are not easy to define graphically.

Class Libraries are used to store classes and their constraints and can be shared between projects. These libraries can represent various protocols or commonly used classes. This prevents the need to duplicate class definitions for use with multiple projects. Once a Class Library is included in a project, all of the classes that are defined in the library are considered local to the project.



Class fields can use any of the generation language's native data types or a user-defined data type (another class). Each field can have a structure of a simple element or depending on the language a queue, an array, or an associative array. Classes can be used to define a packet class, where each field represents a different portion of the packet. Class methods can also be added to the class to act on the fields. Classes are supported in all of the generation languages. The interface for designing and accessing the classes is the same across the generation languages, but the implementation varies radically depending on the language's support for classes and user defined data types.

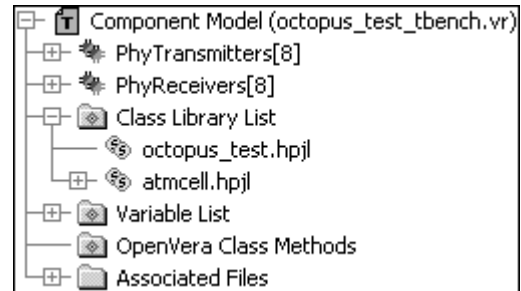
Variables are used to store data that can be set and accessed during simulation. The data type of a variable can be any of the generation language's native data types, or any user defined class that is local to the project. Variables can also be defined to have structures like queues, arrays, associative arrays, input files and output files which are dependent on the generation language.

During simulation, a random number generator can drive the values for variables. The *Constraints* dialog is used to define limits of the random number generation for random variables.

### 8.1 Class Libraries

Class Libraries are used to store classes and their constraints. Class Libraries can be defined and shared between projects. Each project also creates a default library, which is named after the project and stored in the project directory. This default library is independent and can be included in other projects.

All of the class libraries for a project are listed in the **Class Library List** folder in the *Project* window. Double clicking on a library opens it for editing. All of the class libraries for a particular project share the same scope, so the class names must be unique for a particular project. For example, if two libraries contain a class named *ATMCell*, they cannot be added to the same project. Test-Bencher will check for this each time you add a class or add a library, and warn you about any scoping errors.



Class Libraries are created, edited, and added to the project using the *Classes and Variables* dialog. If you open this dialog while a project is opened you will automatically be editing and creating libraries for the current project. Note that whether or not a project is open, modifying a class library will affect any project that includes the library.

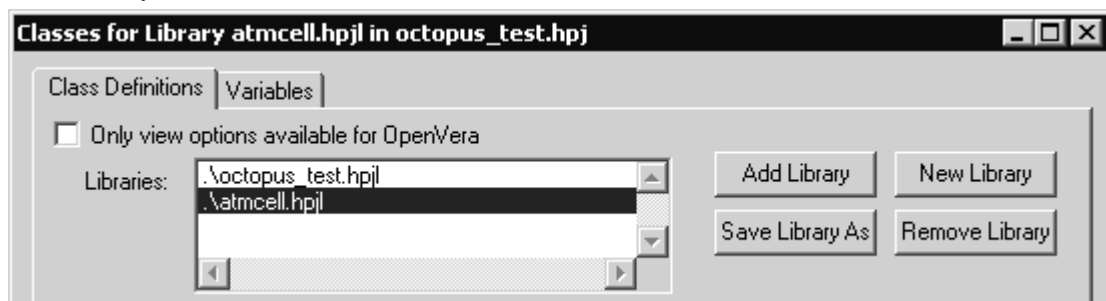
#### To create, edit, add, or remove a Class Library:

- Open the *Classes and Variables* dialog by performing one of the following actions:
  - Select the **Project > Classes and Variables** menu option. This is the only way to open the dialog if no projects are opened.
  - In the *Project* window double-click the **Class Library List** folder.

By default, the *Class Definitions and Variables* dialog is displayed in a language independent view. This means that options may be shown that are not available for the currently selected language. This is provided because class definitions are language independent and can be shared between projects. A language specific view is available for the currently selected language.

To change between the language independent and language specific views:

- Enable the **Only view options available for <Language>** checkbox for the language dependent view. Disable the checkbox for the language independent view.
- Select the **Class Definitions** tab to display the library functions.
- To create a new library, click the **New Library** button to open a file dialog. Name the library and save it. The new library will be listed in the **Libraries** list box.



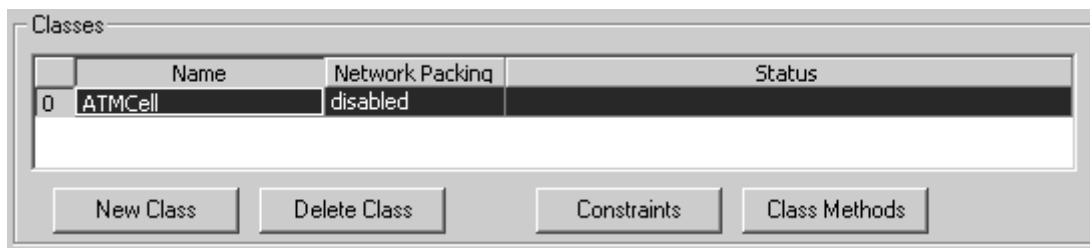
- To add an existing library, click the **Add Library** button to open a file dialog. Browse to find the library and click **OK** to add it to the **Libraries** list box.
- To remove a library, select the library in the **Libraries** list box and click the **Remove Library** button. Note the default library cannot be removed.
- To edit a library, select the library in the **Libraries** list box. All of the classes for the selected library will be displayed below in the **Classes** list box. The selected library is also displayed in the title bar of the dialog.
- *Section 8.2: Classes* describes how to create new classes for the selected library.
- When you are done editing click the **OK** button to save the library and close the *Classes and Variables* dialog.

## 8.2 Classes

To edit or add a class you will open the *Classes and Variables* dialog and select the library that will contain the class. The **New Class** button adds new classes to the *Classes* list. Selecting a class will cause the fields of the class to be displayed in the *Class Fields* list. Clicking the **New Field** button adds a new field to the selected class. The **Constraints** and **Class Methods** buttons are used to open dialogs that allow the respective elements to be created, edited, and removed from the selected class definition. Any cell in the *Classes or Class Fields* list that is not grayed out can be edited by either double clicking or by selecting the cell and then typing.

Classes fields can be defined manually using the **New Field** button or automatically using a file and the **Define from Template** button. Each field can either be a native data type or a class type that is defined in the same class library. Each Field can have the structure of a single element or depending on the generation language it can have complex structures like array, associative array, and queue.

Once a Class is created, it is used when creating variables just like any native data type of the language (*Section 8.3: Variables*).



### To create or edit a class:

- Open the *Classes & Variables* dialog and select the Library that will contain the class. (*Section 8.1: Class Libraries* has more information on libraries).
- Click the **New Class** button in the *Classes* area to add a new class to the list in this area. The class will have default name and status.
- To edit a class or class property, double-click on the cell to open an edit box. For Verilog, the *Network Packing* cell reverses the byte order of the class after packing.

### To add a field to a class:

- Select the name of the class that is being edited in the *Classes* list. This will cause the fields of the class to be displayed in the *Class Field* list.
- Fields can be added manually or by reading in from a file:
  - Click the **New Field** button to add a field.
  - Click the **Define from Template** button to open a file dialog. Find the template file and click **OK**. *Section 8.10 Importing Fields from a Template File* describes the format of the file.
- Double-click on the cell for any field property that needs to be edited. This will cause an edit box, drop-list, or dialog to be opened, and this can be used to edit the property. *Section 8.4 Variable and Class Field Properties* defines different field columns.

	Field Name	Packing	Static	Random	Structure	Size	MSb	LSb	Data Type
0	GFC	big/normal	non-static	rand	element	1	3	0	4_state_vector

**To add a method to a class:**

- Select the name of the class that is being edited in the *Classes* list. Click the **Class Methods** button to open a dialog for entering the class method (*Section 8.7: Class Methods*).

**To add a constraint to a class:**

- Select the name of the class that is being edited in the *Classes* list. Click the **Constraints** button to open a dialog for entering the constraints (*Section 8.8: Constrained Random Number Generation*).

## 8.3 Variables

Variables are used to store data that can be set and accessed during simulation. Variables can be used anywhere in TestBench that you type in HDL code including: Boolean equation and Condition boxes, Sample values, Loop control variables, HDL code markers, class methods, and HDL states.

Variables can be local to either the diagram or to the project component. Diagram-level variables are declared within the transaction source code, and project-level variables are declared within the corresponding project Component Model. The following scope rules are applied to Variables:

- Diagram-level variables are only accessible outside of the diagram if they can be hierarchically accessed in the given language.
- Project-level variables are accessible from the project's transaction, but the access depends on the language you are using. *Chapter 12: Language Specific Details* covers the details on accessing variables.

Each variable has various properties including the data type and the structure. The data type can be any of the generation languages native data types, or the type of any class that is a part of the project class library or any imported class library. The structures for variables can be simple elements or depending on the language queues, arrays, associative arrays, file input and file output structures. The *Classes and Variables* dialog is used to add Variables and edit their properties. The scope of the variables being edited (diagram-level or project-level) is determined by the way in which the dialog is opened.

**To Create Variable:**

- Open the *Classes and Variables* dialog from either the Project for project-level variables or from the diagram for diagram-level variables:
  - For Project-level variables, select the **Project > Classes and Variables** menu option or double-click on the **Variable List** folder in the *Project* window. Another way to open this dialog is to right click the *Component Model* folder and select **Classes and Variables** from the context menu.
  - For Diagram-level variables, click the **View Variables** button in the *Diagram* window.
- Select the **Variables** tab near the top of the dialog. Note the title bar of the dialog indicates which timing diagram or project file that you are editing the variables for.
- Click the **New Variable** button near the bottom of the dialog. You can also just click the blank variable line and start typing a new variable. Either method creates a new variable with default properties in the *Variables* list box.
- Double-click on the column for any property that needs to be edited. This will cause either an edit box or a drop-down list to be opened that can be used to set the field property. The variable properties are defined in *Section 8.4: Variable and Class Field Properties*.
- When you are done click **OK** to close the dialog.



## 8.4 Variable and Class Field Properties

The properties of variables and class fields are added and edited using the *Classes and Variables* dialog as described in *Section 8.3 Variables* and *Section 8.2 Classes*. The section defines the different properties. Certain property settings like size will depend on the structure property. For instance, an element type forces the size to be 1. Other properties will change depending on the generation language. Any item that cannot be edited will have a gray background in the list. The properties include:

- **Variable Name** or **Field Name**: used when referencing the variable or field of the class.
- **Direction**: (Variable Only) this property determines whether or not the variable is an input to the diagram or component. An applied diagram-level variable will become an input to the transaction apply call, just like a state value. An applied project-level variable will become an input to the project Component Model, and each instance of the component can specify an initialization value for the applied project-level variable.
- **Packing**: (Field Only) allows classes to be converted into bit streams. Double-clicking on this field will cause the *Packing Properties* dialog to be opened. This dialog allows packing to be enabled and disabled, and it also allows Bit and Byte order specification. Note that the packing options that are allowed will depend on the language being generated and are discussed in *Section 8.6: Data Packing*.
- **Static**: (Field Only) determines whether or not all instances of the class will use the same copy of the data (static) or if they will maintain their own copy of the data member (*non-static*). By default, this property is set to *non-static*.
- **Random**: Acts on variables and fields that have a native data type. This setting determines whether or not the values assumed by the data member will be randomly generated, and if so what method is used. This property can be set to *non-rand* and *rand*. The *non-rand* selection indicates that the variable is not randomized. If it is set to *rand* then random values will be generated for the class. Random fields are generated when the randomize method is called on the variable. This method is automatically generated when needed, and can be called from the *Sequencer Process* of the Component Model. The generation language may limit the options that are available for randomization. *Section 8.8 Constrained Random Number Generation* shows how to constrain the random values that are generated for random variables.
- **Structure**: determines the type of structure for the field. The types available are *element*, *array*, *associative array*, and *queue*. An element type will cause the field to equate to a simple variable in the class. By default, the structure type for a field is *element*.
- **Size**: specifies the number of elements in a complex structure type field. This setting is available for arrays and queues. Because the default structure type is an element, the default for this property is 1.
- **BitSize, MSB, and LSB**: are used to determine the bit size for field elements. Depending on the language being used, the bitsize may be specified using an LSB and an MSB. Note that some types, such as a string, may not use a bitsize, and that others, like bool, may have a limited bitsize.
- **Data Type**: determines the type of the elements of the field. The possible settings for this property are the available language independent types (such as bool, 2\_state, or 2\_state\_vector) and may also include other classes that have been defined in Class Libraries that are included in the project. The default for this property is *int*. Note that only language independent types that are available for at least one of the licensed languages will be shown. If a language independent type is selected it will be converted to the appropriate type for the generated language in the generated test bench. *Section 8.5: Language Independent Types* provides more information about the language independent types as well as a chart showing the conversion values from these types to the generated language types.
- **Initial Value** (variables only): allows an initialization value to be specified. The variable will be initialized on creation during test bench simulation. An initial value can be specified for both diagram and component level variables. The assignment to this value will occur once at the beginning of simulation execution. The string entered in this field will be placed directly in the generated code without modification.

## Structure Types

**Element:** a single data item (like a single integer).

**Array:** a series of elements of the same data type. Arrays allow data to be randomly accessed through a numeric index. Arrays have a fixed number of elements.

**Associative Array:** stores a series of key-value pairs. The key is used to index a particular element in the array (the value). All of the elements are of the same data type. The size of an associative array grows as data is added to it. Associative arrays are referenced in the same way that arrays are, with the key representing the index. Any integer or value that resolves to an integer can be used as a key for an associative array. The two primary differences between an associative array and an array are that the associative array can grow dynamically (during simulation), and that the numeric keys used to look up a data element do not have to be sequential.

**Queue:** is a FIFO (first-in, first-out) access to data. Queues have a fixed number of elements of the same data type. This type of field is useful if two timing transactions are running concurrently and one of the transactions needs to process data that is collected in another transaction.

**File Structure Types:** allow data to be read from or written to a file during transaction execution. File Structure variables need the Data Type property set to one of the user-defined Classes. Further more, any Class that is used in this manner should have fields that correspond to the columns in the file. A file that is being used for input can be used as the Class Template file (*Section 8.10: Importing Fields from a Template File*). The two File structure types that are available are described below.

**File Output:** is used to write information accumulated during transaction execution out to a file. This information is written out in a spreadsheet like format, and the column headers describe the field information (name, bit information, and radix) in the same manner that is used by the Class Template file. These Variables are used to store state information. The data is written out to the file sequentially, with one row for each transaction that uses the variable.

**File Input:** is used to read data from a file using a spreadsheet like format. This type of Variable can be used to drive state information. The data is written out to the file sequentially, with one row for each transaction that uses the variable.

## 8.5: Language Independent Types

SynapticAD has defined a set of language independent types that is used by TestBench Pro in place of the native types for a given language. This is done to facilitate the development of language independent class definitions and variables. During test bench generation the language independent type is converted to the appropriate native type for the language being generated. Note that not all of the language independent types are supported by all of the generation languages. The dialogs that allow selection of these types, such as the Class Definitions & Variables dialog, will only display the language independent types that are supported for at least one of the currently licensed languages. Additionally, these dialogs support a view that will display only the items that are available for the currently selected language.

The chart below provides a description for each of the language independent types (shown in the Syncad Types column). Following that is a chart that describes the conversion from the language independent types to the native types for language generation.

Syncad Type	BitSize	Description/Values
bool	1	Truth values (1 or 0)
2_state	1	0, 1
2_state_vector	variable	0,1 in vector format
byte	8	Unsigned integer represented by 8 bits

Syncad Type	BitSize	Description/Values
int	32	Signed integer represented by 32 bits
unsigned_int	32	Unsigned integer represented by 32 bits
real	64	Floating poing numbers
fixed_len_string	variable	Series of characters enclosed by quotes
variable_len_string	n/a	Series of characters enclosed by quotes
time	64	Simulation time quantities
4_state	1	0, 1, X, Z
4_state_vector	variable	0, 1, X, Z in vector format
event	n/a	Synchronization item
std_logic	1	U, X, 0, 1, Z, W, L, H, -
std_logic_vector	variable	U, X, 0, 1, Z, W, L, H, - in vector format
std_ulogic	1	Unresolved version of std_logic
std_ulogic_vector	variable	Unresolved version of std_logic_vector
signed_logic	variable	Signed version of std_logic_vector
unsigned_logic	variable	Unsigned version of std_logic_vector

### Type Conversion

The chart below provides conversion information for converting between the language independent types (shown in the Sncad Types column) and the generated language types. Cells that are grayed out represent items where no conversion is available between the language independent type and the native language types.

Syncad Type	Verilog	VHDL	TestBuilder
bool	reg	boolean	bool
2_state	reg	bit	tbvSmartSignal2StateT
2_state_vector	reg	bit_vector	tbvSmartSignal2StateT
byte	reg	bit_vector	tbvSmartSignal2StateT
int	integer	integer	tbvSmartIntT
unsigned_int	integer	natural	tbvSmartUnsignedT
real	real	real	tbvSmartDoubleT
fixed_len_string	reg	string	char[]
variable_len_string			tbvSmartStringT
time	time	time	
4_state	reg	std_logic	tbvSmartSignal4StateT

Syncad Type	Verilog	VHDL	TestBuilder
4_state_vector	reg	std_logic_vector	tbvSmartSignal4StateT
event	event		
std_logic		std_logic	
std_logic_vector		std_logic_vector	
std_ulogic		std_ulogic	
std_ulogic_vector		std_ulogic_vector	
signed_logic		signed	
unsigned_logic		unsigned	

Note that not all language types are perfectly equivalent to the language independent type. Variances are as follow:

- Verilog reg type is a four state type.
- Verilog integer type is signed.
- VHDL natural is a limited version of the VHDL integer type, so it's max value is  $2^{31}$ , not  $2^{32}$ .
- Some languages do not provide an unsigned integer type.

## 8.6 Data Packing

Data packing is the method used to convert data structures to bit streams or byte streams. Data packing is used when you want to work with data at a higher level of abstraction. Instead of passing data around the test bench using byte arrays or bit streams, you can create a structure definition which defines the data in a way that is easier more human readable way. Then, when you want to actually use this data to drive values onto a bus, the data structure can be packed into a bit stream or array of bytes. Basically, when you pack a data structure, all of the fields will be concatenated in the way that you specify through the packing options.

When creating structure definitions, packing properties can be individually specified for each field of the structure. Packing can be enabled individually for each field and when enabled there are several different ways the field can be packed. These options, which are listed below, control the order in which bits, bytes, and words are packed. Note: the order in which fields are packed is determined by the order in which they are shown in the structure definition. The first field (that has packing enabled) is the first field to be packed.

- **Endianess:** Little Endian (default) or Big Endian. If Big Endian is used, then for each field:
  - each pair of bytes is swapped. Note: if the size of the field is not a multiple of 16 then this step will not be performed.
  - each pair of 16 bit words are swapped. Note: if the size of the field is not a multiple of 32 then this step will not be performed.
- **Networking:** if this option is selected then the order of the bytes in the resulting packed byte array will be reversed.
- **Bit Normal:** selecting this option causes the data to be arranged so that the most significant bit of each byte is first in the vector.
- **Bit Reverse:** this option causes bit vector to be reversed, resulting in the least significant bit of the vector to be packed first and the most significant bit to be packed last.

*Chapter 12: Language Specific Details* discusses the language specific features of data packing that TestBench supports.

## 8.7 Class Methods

Class methods are user defined functions and tasks that let you add HDL algorithms to the timing diagram or bus-functional model. These methods can be added to individual timing diagrams, project components, or user-defined classes. Once a class method is added to an object it shares the same scoping level as the object. It can be called during simulation to perform activities that are difficult to describe graphically.

Diagram-level class methods, like diagram-level variables, are local to the timing diagram in which they are created. These methods can be accessed using HDL Code Markers and Sample Actions. They are generated in the diagram transaction source file so they share the same scoping level of other diagram-level objects. The code for the methods is stored in the timing diagram file so that the methods are available for other projects (if the diagram is included in multiple projects).

For TestBench, project-level class methods can be accessed from the Sequencer Process in the Component Model. These methods are specific to the project for which they are defined. The top-level module of the bus-functional model (generated from the top-level template file for the project) contains these methods.

For TestBench, class-level class methods can be called from the same scoping level as the variable that instantiates the class. These methods are a part of the class definition and are stored in the respective class library. The methods are generated in the class source code definition.

The *Class Methods* dialog is used to create and edit user-defined methods. Class methods are defined in three different sections of the dialog: name, parameters, and source code. Selecting a different class method name changes the contents of the other sections of the dialog. The parameters represent data that is passed into the method. The source code is the actual code that will be placed in the generated method definition. This code is written in the generation language. Class Methods can be specified for every licensed language, allowing diagrams, class definitions, and even the Component Model to be language independent. To define a new class method:

- Open the *Class Methods* dialog from the object that you want to define the class method for:
  - For diagram-level methods, click the **Class Methods** button located in the *Diagram* window.
  - For class-level methods, click the **Class Methods** button located *Classes and Variables* dialog for the selected class.
  - For project-level methods, right click on the **Component Model** folder in the *Project* window and choose **Class Methods** from the context menu.
- Create a new class method.
  - Select the Language that the method is to be defined for from the **Language** drop-list.
  - Click the **New Method** button to create a new method with default values.
  - Double-click on the cell for any property that needs to be edited. A class method's properties are:

- **Method Name:** the name of the method.

- **Method Type:** Test-Bencher supports two types of methods. *Tasks* perform an operation on the parameters that they are passed, but do not specifically return any value.

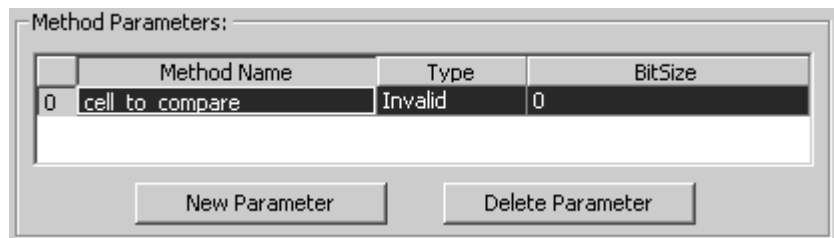
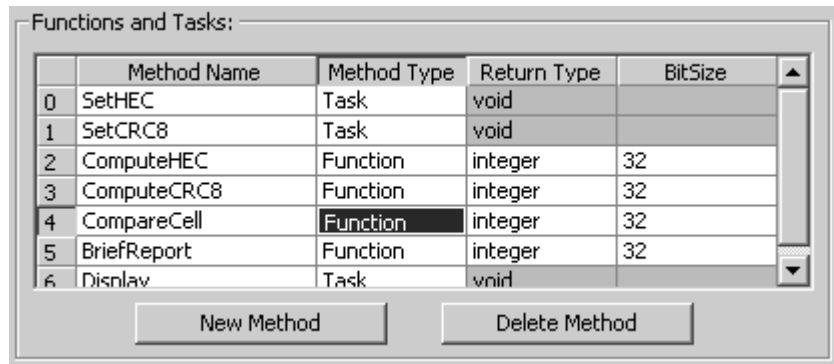
*Functions* perform an operation on the parameters that they are passed and return a value.

- **Return Type:** the data type of the value that is returned by a function. This can be any of the language independent types defined for the language, or any of the user-defined classes. *Section 8.5: Language Independent Types* provides more information about the language independent types and their conversion to the native languages.

- **Bitsize:** the size in bits of the value that is returned by a function. This value is only editable if the *Return Type* is a bit type.

- Add parameters to the class method:

- Select the class method in the *Functions and Tasks* list. This will cause the class methods parameters and source code to be displayed in the rest of the dialog.



- Click the **New Parameter** button to create a new parameter with default properties.

- Double-click on the column for any property that needs to be edited. A parameter's properties are:

- **Name:** the name of the parameter.

- **Type:** the data type of the parameter. This can be any of the Syncad types defined for the generation language or any of the classes in the project (*Section 8.2: Classes*).

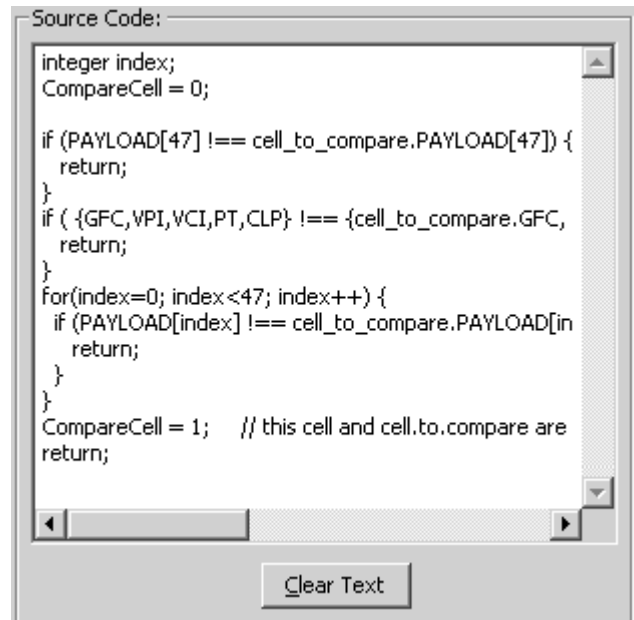
- **Bitsize:** the size in bits of the parameter. This value is only editable if the *Type* is a bit type.

- Add the class method source code:
  - Select the class method in the *Functions and Tasks* list. This will cause the class methods parameters and source code to be displayed in the rest of the dialog.
  - Type your source code into the *Source Code* edit box.

## 8.8 Constrained Random Number Generation

Random values can be generated for variables. When randomization is used, constraints are usually needed to specify the valid ranges of the random numbers generated. Constrained random number generation is supported in TestBuilder.

To use constrained random number generation, you must first define a variable or the fields of a class to have a random property. Simple variables that have a native data type can be set to random by checking one of the random properties when you define the variables (*Section 8.3 Variables*). Complex variables that have a user-defined class as the data type can also be randomly generated, but the randomization properties are set during the definition of the class. Individual fields of the class that have a native data type can be set to be random (*Section 8.2 Classes*). Next the *Constraints* dialog is used to define the limits of the random number generation. And finally, during simulation you will call the randomize functions of your generation language to generate the random numbers.

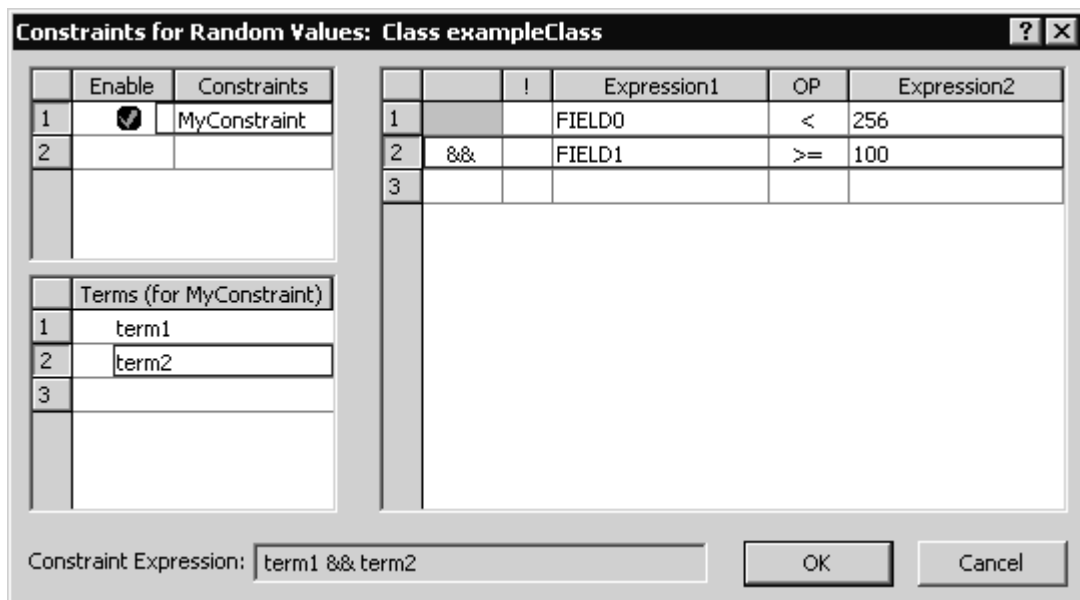


```

Source Code:
integer index;
CompareCell = 0;

if (PAYLOAD[47] != cell_to_compare.PAYLOAD[47]) {
  return;
}
if ( {GFC,VPI,VCI,PT,CLP} != {cell_to_compare.GFC,
  return;
}
for(index=0; index<47; index++) {
  if (PAYLOAD[index] != cell_to_compare.PAYLOAD[in
  return;
}
}
CompareCell = 1; // this cell and cell.to.compare are
return;
  
```

Clear Text



	Enable	Constraints
1	<input checked="" type="checkbox"/>	MyConstraint
2	<input type="checkbox"/>	

	!	Expression1	OP	Expression2
1		FIELD0	<	256
2	<input checked="" type="checkbox"/>	FIELD1	>=	100
3				

	Terms (for MyConstraint)
1	term1
2	term2
3	

Constraint Expression:

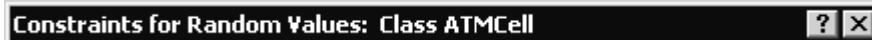
OK Cancel

Constraints have the form of *ConstraintName = term1 && term2 && term3 ...* Where each term consists of a Boolean expression. Constraints are entered using the different sections of the *Constraint* dialog. By entering the information in this manner the Constraints are somewhat language independent. The individual expressions are language dependent, so they may have to be modified slightly if moving to a different language. The Constraint dialog image shows a complex expression for *term2 = (FIELD0 < 256) && (FIELD0 >= 100)*.

There are four basic steps required for adding a constraint:

1) First, open the *Constraints* dialog for a random variable or class with random fields by:

- Open the *Classes and Variables* dialog and select the **Class Definitions** tab or the **Variables** tab depending on what you are editing (*Section 8.3: Variables* or *Section 8.2: Classes*).
- If you working with a class, select the library and class that you want to constrain.
- Click the **Constraints** button in to open the *Constraints* dialog. The title bar of the dialog will indicate which constraints you are editing, either a specific class or the list of variables.



2) Create new *constraint* and enable it:

- Double-click in the empty constraint cell to open a edit box and type the name of the new constraint.
- Constraints with a green checkmark  are enabled and will affect the random number generation for the project. Click on this column to toggle to the Disable State. A red X  indicates that a constraint is temporarily disabled.

	Enable	Constraints
1	<input checked="" type="checkbox"/>	EnabledConstraint
2	<input type="checkbox"/>	DisabledConstraint
3		

3) Create a new *term* for the constraint.

- Select the constraint for which you want to create a new term.
- Double-click in the empty term in the *Terms* section to open an edit box and type in the name of a new term. By default, each term that you add will be added together with the other terms for the constraint and displayed the *Constraint Expression* edit box.
- Edit the *Constraint Expression* directly by adding parentheses, && and || operators between the terms.

Terms (for MyConstraint)	
1	term1
2	term2
3	

4) Define *expressions* and *compound expressions* for each term in the constraint:

- Select the *term* that will contain this expression.
- Double-click in each of cells to build an expression. At least one complete expression needs to be entered for each term in the constraint. The rules for building expressions are given below.

	!	Expression1	OP	Expression2
1		FIELD0	<	256
2	&&	FIELD1	>=	100
3				

- Click **OK** to save the constraints and close the dialog.

### Expressions in Constraints

Expressions are added using the expression grid in the *Constraints* dialog. The Expression columns can contain any combination of variables and variable fields that share the same scoping level of that the constraint will be used at, constants and mathematical operators like: +, -, and \* operators. The rest of the columns can contain operators: == (is equal to), != (is not equal to), < (is less than), > (is greater than), <= (is less than or equal to), and >= (is greater than or equal to).



Some possible expressions for a class with three fields, **FIELD0**, **FIELD1** and **FIELD2**, include:

```
FIELD0 < 10
```

```
FIELD0 + FIELD1 <= 15
```

```
FIELD1 * 3 == FIELD2 - 2
```

The *Expressions* section of the *Constraints* dialog contains the following columns:

		!	Expression1	OP	Expression2
1			FIELD0	<	10
2	&&		FIELD0 + FIELD1	<=	15
3		!	FIELD1 * 3	==	FIELD2 - 2
4					

- The first column contains the Boolean operator that will precede the expression in the term. This box is grayed out for the first expression, because it is unnecessary. The choices for this dropdown are **&&** (Boolean AND) and **||** (Boolean OR). **&&** is the default selection.
- The second column (headed by a **!**) indicates whether to apply a Boolean NOT to the expression. If the **!** is in the column, then the expression will be inverted.
- The third column (**Expression1**) contains the left-hand side of the expression.
- The fourth column (**OP**) contains the comparison operator that the expression will use to compare **Expression1** and **Expression2**.
- The fifth column (**Expression2**) contains the right-hand side of the expression.

## 8.9 File Input and Output Variables

Variables with a structure type of **File Output** or **File Input** are used to write or read data to or from a file. In order to create a file variable you must first define a class that represents the structure of the file and then define a variable with the structure type set to a file type. There is an easier automated method to making the file variables. Instead you just add the file to the project's *Test Vector List - File Input* or *Test Vector List - File Output* folder and TestBench will automatically parse the file and create a *Class* definition with a field for each column of the file. TestBench also instantiates the class as a variable in the project. From anywhere in the Project or a transaction of the project you can access the file variable. File Input and File Output files use the WaveFormer Pro spreadsheet file described in *Section 8.10*. To add a file and create file variables:

- Define a file with the header information described in Section 8.10.
- In the *Project* window, open the **Test Vectors Lists** folder.
- For file output variable, right click on the **File Output** folder and choose the **Add Files to File Output folder** from the context menu.
- For file input variable, right click the **File Input** and choose the **Add Files to File Input Folder** from the context menu. Note file input is not available in all languages.
- Choosing one of the above menu options will open a file dialog that lets you choose the file that defines the data structure.
- Click **OK** to close the dialog and generate the class definition and variable. Choose **Project > Classes and Variables** menu to view the new variable and class definition.

## 8.10 Importing Fields from a Template File

A Class Template File can be used to specify the name, MSB, LSB and radix for the fields of a class. The format of the template file is a tab-separated file that can be generated using a spreadsheet program. Note: The format used for the class template files is compatible with WaveFormer's spreadsheet format. This format can be exported from and imported to timing diagrams. This is also the file format used for File Input and File Output in TestBencher (*Section 8.9: File Input and Output Variables*).

To define a class the just the first two header lines of the file are used. Other lines are ignored. *Section 8.2: Classes* describes how to use the file to define the fields of a class. The format of the class template file should be as follows:

```
[Vectors] Radix=defaultRadix
    fieldName1[MSB:LSB](Radix)    fieldName2    fieldName3
```

The Vectors statement is required so that TestBencher will recognize the file format. A default radix can be specified for both file input and output. This specification is made with the Radix= assignment. The supported radices are **hex** (hexidecimal), **bin** (binary), **dec** (decimal), or **real**. If no default radix is provided, a hex radix is assigned. The default radix can be overridden for a specific column by placing the radix information for that column in the field name.

The second line of the file contains the field names and optionally the size and radix of a signal. Any other lines contain data. For input files, a line of data will be read in from the file each time a file input variable needs more data. If you are using the file to create a file output variable the data will be ignored. During simulation the specified output file will be opened and the data overwritten.

The example below shows the formatting for a template that contains two columns. The first line contains the Vectors statement and a default radix of **hex**. The second line defines the data columns. If a class is created using this file t will have two fields named Addr[0] and Data[7:0].

```
[Vectors] Radix=hex
Addr    Data[7:0](bin)
A4      10110011
```

## 8.11 Semaphores

Semaphores are a special variable type that is used to help secure a critical region. Semaphores do not have the same properties as a normal variable - the only property that can be defined is the **Initial Value**. Semaphores are defined at the project level and can be accessed by transactors in the project. Multiple transactors can access the same semaphore, so that a critical region can be defined for diagrams that are running concurrently.

### Creating New Semaphores

To add a new semaphore to the project:

- Select the **Project > Classes and Variables** menu item. This will launch the *Classes and Variables* dialog.
- Click the **Semaphores** tab. This will display the Semaphores that are available in the current project.
- Click the **New Semaphore** button to add a new Semaphore to the project. The **Name** and **Initial Value** can be edited by double-clicking the values in the tree. New Semaphores can also be created from the Marker dialog by using the name of a new semaphore when creating a Semaphore or Pipeline Boundary marker (see *Section 7.7: Semaphore Markers* or *Section 7.8: Pipeline Boundary Markers* for more information).
- The **Delete Semaphore** or <delete> key will delete the currently selected Semaphore.

### Using Semaphores with a Marker

Semaphores can be used in a transactor that is part of a project using the **Semaphore Marker** or the **Pipeline Boundary** marker type. These two marker types are described in more detail in *Section 7.7: Semaphore Markers* and *Section 7.8: Pipeline Boundary Markers*.





# Chapter 9: Project Component and Transaction Sequencer

When TestBencher creates a bus functional model from a project, it generates the Component Model from a user-specified template file. The Component Model is the top-level control file for the project. For top-level projects, this is the place where the transactions are instantiated and if there is a MUT it is instantiated. The template file also maintains the Sequencer Process, which holds the *Transaction Sequencer* logic for applying transactions to the MUT. For sub-projects, this is the place where any initialization class methods are handled.

The template file is specified in the *Project Wizard* when the project is first created (*Section 2.1: Creating, Opening, and Saving Projects*). The template file is a generation language source code file where both the user and TestBencher insert source code. When the generation language for a project is changed, the template file must be changed using the technique described in *Section 9.6: Changing a Project Template File*. TestBencher generates the bus-functional model code within the pairs of commented key words called macros. Each time the bus-functional model is generated the code within the macro lines is overwritten. An example of a macro keyword pair is:

```
-- $ComponentInstantiationsForAllDiagrams
-- End $ComponentInstantiationsForAllDiagrams
```

The Sequencer Process is the area in the template file where you will construct the **Transaction Sequencer** that defines how to apply transactions to the model under test. Transactions can be called directly using the *Insert Diagram Calls* dialog (see *Section 9.1: Transaction Calls*), read from a file using the *Transaction Manager* queue (see *Section 9.6: Transaction Manager and Test Reader*), or randomly generated using the *Transaction Generator* (see *Section 9.3: Transaction Generator*). The *Transaction Monitor* (see *Section 9.5: Transaction Monitor*) creates a coverage report on the operation of the Transaction Generator. Structural calls like loops and conditional statements can be added to control the test execution and generate data (see *Section 9.2: Writing Code in the Template File*). The sequencer process is outside of any macro statements so the code will be preserved during code generation process. By default the top-level template file contains one Sequencer Process, but you can add more sequencer processes if you have sets of transactions that need to execute completely asynchronously.

## 9.1 Transaction Calls

The *Insert Diagram Subroutine Calls* dialog generates diagram apply calls so you do not need to memorize the function syntax. Each timing diagram generates three task calls: Apply, Apply-nowait, and Abort. Apply runs the transaction in a blocking mode, and Apply-nowait runs the transaction concurrently with other transactions. The Master/Slave *Diagram Setting* (see *Section 3.8: Diagram Settings Dialog - General Tab*) determines how many times a transaction executes. Master Transactors run once and stop. Slave Transactors and Global Clocks run in a looping mode until an Abort call is received.

In addition to these task calls, you can also place HDL code in the sequencer. One example where this would be useful is if you wish to place conditions on whether or not a timing transaction is called, or on the parameter values that you wish to have applied.

An alternative method to placing transaction calls in the sequencer process is to create a file external to the bus-functional model with transaction calls and during simulation read the transaction calls from a file (*Section 9.3: Transaction Manager and Test Reader*).

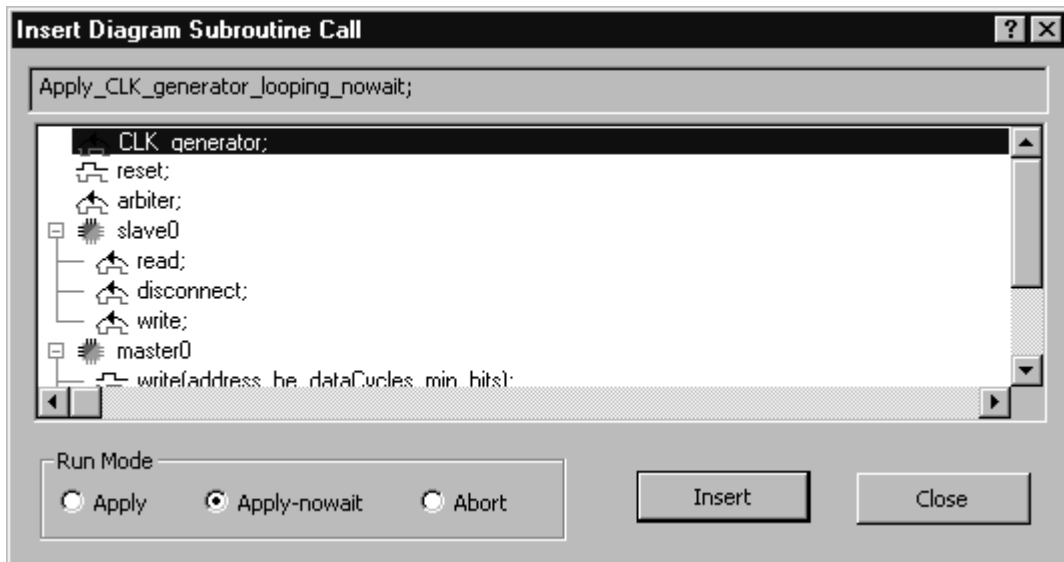
### To edit the sequencer process:

- Double-click on the **Component Model** folder in the *Project* window to open the template file in an editor window. *Appendix A: Editor Commands* has information on the editor commands and using external editors.

- Scroll down in the template file until you find the Sequencer Process. You will see a comment in the code that looks like:

```
Transaction Sequencer - After this comment, define how to
    apply transactions to the model under test using:
```

- Click in the template file just below this comment, then right click and select **Insert Diagram Calls...** from the context menu to open the *Insert Diagram Calls* dialog. This dialog contains a list of available 'Apply' statements, one for each diagram in the project. If you have not added any diagrams to your project the list will be empty.



- Select a timing diagram name. The *Run Mode* radio buttons will default to **Apply** for Master transactions to run them in a blocking mode. For Slaves the default is **Apply-nowait** to run the transaction concurrently with subsequent apply calls.
- Either double click on the timing diagram name to accept the defaults OR choose a *Run Mode* radio button and press the **Insert** button. Either of these actions will insert an apply call.

**Note:** The Apply statement was inserted at the same line as your cursor. The *Insert Diagram Call* dialog is a modeless dialog it can remain open while you perform other actions. Inserting additional Apply statements causes those statements to be added on successive lines.

- If any of the applied transactions contain variables, then edit the template code to provide values for variable names (see *Section 9.2: Writing Code in the Template File* for more details). In the example Apply statement below, a value of three is assigned to `stateVar`.

```
// Apply_verySimpleCyclic(stateVar);
Apply_verySimpleCyclic(3);
```

## 9.2 Writing Code in the Template File

Generation language source code can be added anywhere in the template file, except for inside a macro statement. The macro statements are places in which TestBench will place the generated code. The most common place to add source code to a template file is in the sequencer process. This process is the area where diagram calls are placed in the template file (see *Section 9.1: Transaction Calls*).

You can write source code to place conditions on whether or not a diagram call is made based on information returned from previous transactions. You can define variables, loops, and other control structures. This is also the place where calls to the randomization functions are made.

The following is a Verilog example of a sequencer process that performs a sweep test on an SRAM. The variable *delay* is defined to be a counter for a *For* loop. Each time the loop executes a different delay is passed into the Write and Read cycles of the SRAM. This delay conditional controls a signal in the Write and Read transaction. Variables could have been defined for the transactions, but we choose to pass in values. The transaction apply calls were written using the *Insert Diagram Calls* dialog as described in *Section 9: Transaction Calls*. The user wrote all of the other code in this example.

```
// Sequencer Process
real delay0; // delay0 will serve as the index and the delay value
initial
begin
  for (delay0 = 32.0; delay0 > 5.0; delay0 = delay0 - 5.0)
  begin
    // Apply_Tbwrite( addr , data , $realtobits(delay0_min) );
    Apply_Tbwrite( 'hF0 , 'hAE , $realtobits(delay0) );
    // Apply_tbread( addr , data , $realtobits(delay0_min) );
    Apply_tbread( 'hF0 , 'hAE , $realtobits(delay0));
  end
  Abort_tbglobal_clock;

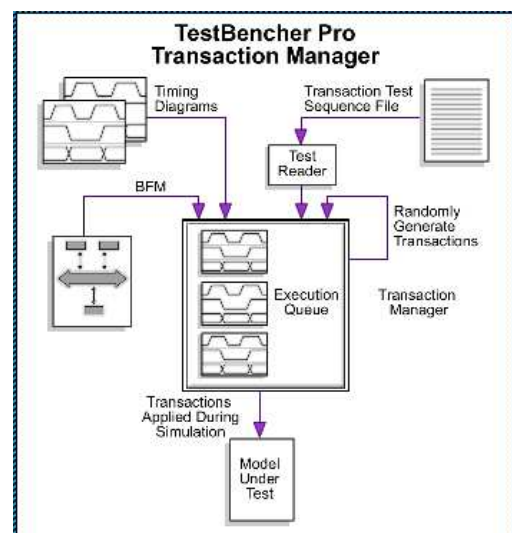
  $fclose(logfile);
end
```

### 9.3 Transaction Manager and Test Reader

In addition to sequentially executing transaction calls that are placed in the template file, TestBencher can generate a transaction manager module that maintains a queue of transactions to be executed. Transactions can be generated randomly, posted to the queue during simulation, or read in from a file using the Test Reader component.

TestBencher automatically generates Transaction Manager and Test Reader code from the transactions included in the project. Attempting to create and maintain this type of code manually is difficult because the code changes each time you add a new transaction type or change the number and types of parameters for a transaction.

During simulation, the Transaction Manager maintains a queue of transactions to be executed. The manager can randomly generate transactions to fill the queue based on a weighted function. Transaction Diagrams can dynamically post other transaction calls to the queue based on responses from the model under test. And the transaction manager technology also supports TestBencher's hierarchical bus-functional model feature by allowing BFMs to post



transactions to sub-component BFM queues. In particular, a top-level BFM model can generate test sequences and post them to the child BFM transaction managers. All calls to transactions can be specified either as relative or fixed path, allowing any transaction to be referenced from anywhere in the project.

The transaction manager can also fill the queue by reading the transaction test sequences from a file. Different test sequence files can be applied to a test bench without having to recompile the test bench. Transaction test sequence data files are also easier to create and maintain because they are smaller and contain less repetitive information.

With TestBencher's transaction manager and test reader feature, the user simply defines the inputs and outputs for each transaction type as part of his normal test bench creation process and the test reader code is automatically created. The user inserts Post Diagram calls into the template file or into HDL code segments in a transaction. These Post Diagram calls place the transaction apply calls in the transaction manager's queue. The user can also make Apply File calls to hand an entire file to the Transaction Manager.

#### To enable the Transaction Manager generation:

- In the *Project* window, right click on the **Component Model** folder and choose **Project Generation Properties** from the context menu. This opens the *Project Generation Properties* dialog.
- Check the **Enable Transaction Manager** checkbox click **OK** to close the dialog.
- During the next code generation TestBencher will generate a transaction manager module. *Section 10.1: Generate the Bus-functional Model* describes how to force a code generation.
- Next add apply file calls and post diagram calls to the template file and code segments in the timing diagram.

*Chapter 12: Language Specific Details* discusses the syntax for apply calls and post diagram calls.

#### Format of the Transaction Manager Test Sequence Files

An input file can be created to specify the test sequence used by the Transaction Manager module. This input file contains a row for each transaction to be called. Below is an example of test sequence file.

```

CLK_generator 1
arbiter 1
slave0.write 1
slave0.read 1
slave0.disconnect 1
reset 0
master0.write 0 f0000000 f 10
master0.read 0 f0000000 f 10
slave0.write 2
slave0.read 2
slave0.disconnect 2
arbiter 2
CLK_generator 2

```

Each row has the following information in the following order:

- Instance Transaction Name - the name of the transaction and the relative path from the top-level module to the project that contains the transaction manager. Some options are *transactionName*, *subProjectInstanceName.transactionName*, *projectName.transactionName*, and *projectName.subProjectInstanceName.transactionName*.



- Transaction Mode - an integer to specify how the transaction should be executed. The different types of modes are shown below.
  - 0 => Apply - will block the transaction manager from executing any more transactions until completed.
  - 1 => Apply\_nowait - will trigger the transaction and immediately execute the next apply call in the queue.
  - 2 => Abort - aborts the specified transaction.
- Parameters - there can be any number of parameters depending on the transaction that is specified in the transaction path. TestBencher will read in the appropriate parameters based on the transaction type.

Once the file is created it can be applied by calling the `ApplyFile` or `ApplyFile_nowait` method for the transaction manager. This varies slightly depending on which language you use (see *Chapter 12: Language Specific Details* for more information). When `ApplyFile` is called, it will immediately close the current file that is being used by the transaction manager (if there was one specified earlier) and open the new file that is specified. The transaction manager will then read one line at a time from the file whenever the apply call queue becomes empty. `ApplyFile` will not return control to the calling process until all transactions have been read from the file. `ApplyFile_nowait` will return as soon as the file is opened. So, if `ApplyFile_nowait` is called twice on the same transaction manager without any delay between the calls, the first call will be overridden by the second call since only one file can be opened at a time by one transaction manager.

### Transaction Manager Modes

The Transaction Manager can also be run in different modes by calling a *SetApplyCallMode* function (see *Chapter 12: Language Specific Details*). By default the mode it is set to looping so that transactions are executed when they are available in the queue. There are three modes:

- TB\_LOOPING - run the next transaction from the queue whenever there are apply calls in the queue.
- TB\_ONCE - only run the next transaction from queue when **ApplyNextTransaction** is called.
- TB\_SUSPEND - finish running the current transaction and stop reading apply calls from the queue until the apply call mode is changed.

## 9.4 Transaction Generator

The Transaction Generator creates random master transaction calls and feeds them to the Transaction Manager during simulation. A weightings table defines the state machine and probability of executing transactions one after another. Calls to **RunRandomTransaction** apply transactions to the Transaction Manager. Each time a random transaction is applied the arguments for the transaction are randomly generated using the constraint definitions of the variables. The Transaction Generator is supported in Verilog with TestBuilder, VHDL with TestBuilder, and pure VHDL.

When the Transaction Manager is enabled the code for the Transaction Generator is automatically made available (Section 9.4). The weightings table and function calls **SetTransactorWeightings** and **RunRandomTransaction** will also be generated. You will use these table and function calls to control how the Transaction Generator works.

The weightings table is a state matrix where the rows define the most recent master transactor to be applied and the columns define the next transaction to execute. The first row/column is reserved for the **reset** or starting state of the BFM. The order of the master transactions is the same as they appear in the *Project* Window. A zero in the weightings table indicates that a specific transaction will never follow another. The higher the number the more likely a transaction will follow. You must copy and paste this table from the generation macro to the code section below the **Insert Diagram Calls** comment, because the original table within the macro is rewritten each time the test bench is generated.


Below is an example of some TestBuilder code for the sequencer process. The first call starts the clock. Next the weightings table has been edited so that it is 5 times more likely that diagram **write** will run after a **reset** than diagram **read**. The `SetTransactorWeightings` call registers the table. The *for loop* calls `RunRandomTransactor` 10 times and randomly applies read and write transactions. The data and address arguments for the transactions are automatically randomly generated using the constraint settings for the variables.

```

//*****
// Transaction Sequencer - After this comment, define how to
//   apply transactions to the model under test using:
//
// - Transaction calls (Insert Diagram Calls in right-click menu)
// - Transaction Manager to read transactions from a file
// - Transaction Generator's RunRandomTransactor() randomizes calls
// - Source code in C++ and TestBuilder
//*****
Tvm.Apply_clk_generator.spawn( );
           // to |           |
           // reset | write | read
int weightings[3][3] = { { 0,    5,  1 }, // from reset
                        { 0,    1,  1 }, // from write
                        { 0,    1,  1 } }; // from read
SetTransactorWeightings( weightings );
or (int i=0; i < 10; i++)
{
    RunRandomTransactor();
}

```

To use the Transaction Generator:

- Make sure that the transaction manager generation is enabled, by right clicking on the **Component Model** in the project node, and choosing **Project Generation Properties** menu and then confirm that the **Enable Transaction Manager** is checked.
- Press the **Make TB** button, , to generate the test bench.
- In the *Project* window, double click on the **Component Model** node to open the project component template file in an *Editor* window.
- Scroll down to the *Sequencer Process* and read through the code until you find the **weightings** table. TestBencher automatically generates a table of 1's that means that there is an equal chance of each transaction being randomly generated.
- Copy and paste the **weightings** table and the `SetTransactorWeightings` function to the area below the **Insert Diagram Calls** comment.
- Edit the weightings table to indicate that the chance of a transaction being generated after another one occurs. The larger the number the higher the chance of being generated.
- Use the *Insert Diagram Calls* dialog to add the calls to start the slave diagrams like the clock generator (*Section 9.1: Transaction Calls*).

- Then make calls to the **RunRandomTransactor()** function to apply transactions to the model under test.

## 9.5 Transaction Monitor

The Transaction Monitor works with the Transaction Generator to ensure that the randomly generated transactions adequately cover the test space. The transaction monitor will monitor the generated transactions and create a coverage log. During simulation, the monitor will use a series of test criteria change the Generator's weightings table to force certain cases to execute. The Transaction Monitor is supported in Verilog with TestBuilder, VHDL with TestBuilder, and pure VHDL.

To use the Transaction Monitor:

- Setup the Transaction Generator as described in *Section 9.5: Transaction Monitor*.
- Edit the Transaction Monitor test strategy
- After a simulation the coverage log is displayed in the *Report Window*

## 9.6 Changing a Project's Template File

A top-level template file is automatically created for a project when the project is created. Sometimes it is necessary to change the template file for a project, such as when the language of the project is changed (see *Section 2.5: Component and Component Instance Generation Properties* for information on changing the project language). The *Copy TestBench Template* dialog is used to copy a new template file into a project. The *Set TestBench Template* dialog is used to switch between different template files.

Some generation languages or language combinations require multiple template files. In these cases, the template file that contains the Component Model and Sequencer Process is the template file that is named in the *Copy* and *Set TestBench Template* dialogs. Any secondary template files are copied automatically when the primary template file is copied. The secondary template files are displayed under the **Component Model** folder of the *Project* window.

**To copy a new template file and use it in a project:**

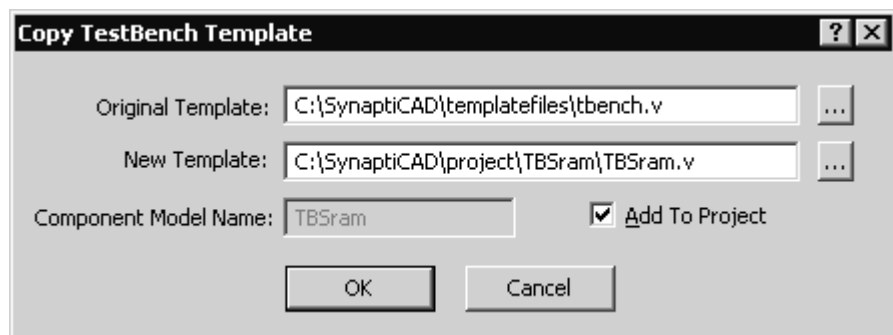
- Before you change the template file for a project, it is recommended that you copy the Project directory to another place. For multiple languages, the template files are not language independent. Create separate projects with the language dependent template files in different directories, and include the same transactions and class libraries.

- Choose **Project > Copy TestBench Template** menu option to open a dialog with the same name.

- Check the **Add to Project** checkbox so that the template will be included in the project.

- Use **Original Template** box to choose a template file that matches the language that you are working or use the Browse (...) button to find one on the hard drive. If the project language is a language dialect combination, such as TestBuilder/Verilog, then the selected template should be for the primary language - TestBuilder in this case. The following is a list of the template files shipped with TestBench:

- **TestBuilder** uses *tbuidOnlyTBench.cpp* as the primary and *tbuidOnlyTBench.h* will be included as a secondary template file.



- **Verilog** uses *tbench.v* or *isotbench.v* as the primary file. The **isotbench** template file dumps just the top-level signals of the MUT to the waveform window, while the **tbench** template file also dumps all of the transaction status signals. The **tbench** is the default test bench template file because it provides more information and the file is simpler.
- **Verilog with TestBuilder Integration** uses *tbuildTBench.cpp* as the primary, with *TbuildTBench.h*, *tbuildMain.cpp*, and *tbench.v* included as secondary template files.
- **VHDL** uses either *tbench.vhd* or *isotbench.vhd* as the primary file. The **isotbench** template file dumps just the top-level signals of the MUT to the waveform window, while the **tbench** template file also dumps all of the transaction status signals. The **tbench** is the default test bench template file because it provides more information and the file is simpler.
- **VHDL with TestBuilder Integration** uses *tbuildTBench.cpp* as the primary, with *TbuildTBench.h*, *tbuildMain.cpp*, and *tbench.vhd* included as secondary template files.
- In **New Template** edit box, enter a file name for your new template file (this file will become your test bench).
- Click **OK** to copy the template and add it to the *Project* window. Notice that the template file appears in the project window.

**To switch in a different template file:**

- Choose **Project > Set New Component Template** menu option to open a file dialog.
- Choose a template file that you created earlier using the *Copy TestBench Template* dialog.
- Click **OK** to close the dialog and switch in the new template file.

# Chapter 10: Generation and Simulation


This chapter discusses how to generate the top-level test bench from the template file, errors during test bench generation and simulation of the test bench. TestBench can control external simulators, compilers, and HVL tools to build, link and simulate a design.

## 10.1 Generate the Bus Functional Model

The last thing that you do in TestBench before simulating the model is to generate the test bench. TestBench verifies that the source code files for each of the timing diagrams in the project is up-to-date. The diagram transaction files are updated each time the timing diagrams are saved. All class libraries are regenerated. And the macro statements in the project's template files are expanded.


Any errors and warnings that may occur during test bench generation will be reported in the test bench log file, **waveperl.log** in the *Report* window. Some possible errors include marker loops don't match, invalid marker attributes, unable to find input and output files, and invalid file formats. TestBench will also warn you of potential problems, such as two distinct signals within the same timing diagram using identical names but different directions (this may not always be a problem, but if the signal is being used as an output of the diagram, it may cause confusion). The log file will also list of the total number of warnings and errors that have occurred. If there are no errors in code generation then you are ready to simulate.

### To generate the bus-functional model:

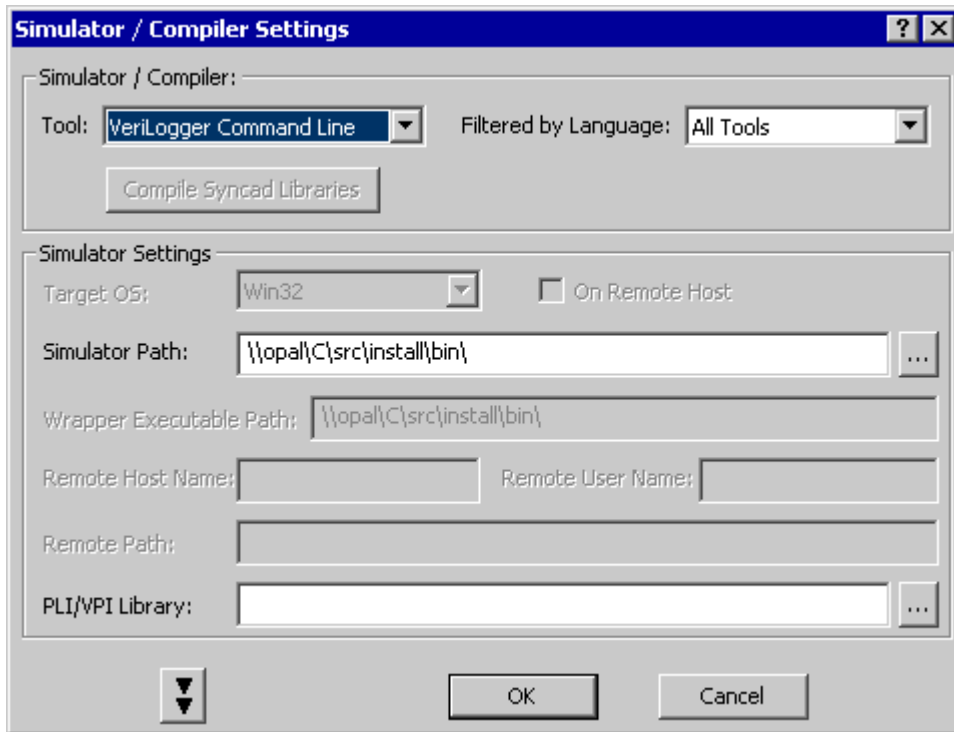
- Click the **Generate Test Bench** button  on the *Simulation* button bar.
- In the *Report* window, click the **waveperl.log** tab and check for errors. If you can't see the *Report* window, select the **Window > Report** menu to bring the window to the top of the screen.
  - If there are errors, fix them and regenerate the test bench.
  - If there are no errors, then you are ready to simulate.

## 10.2 Simulator and Compiler Settings Dialog

The external simulator and compiler paths are set using the *Simulator / Compiler Settings* dialog. These settings are saved in the syncad.ini file. Before you simulate, you will also need to set which tool to use for a particular project by using the *Project Simulation Properties* dialog covered in *Section 10.3: Project Simulation Properties Dialog*. To change the path information for external tools:

- Choose the **Options > Simulator / Compiler Settings** menu option to open the *Simulator/Compiler Settings* dialog.
- In the **Tools** drop-down choose your simulator or compiler. This will enable options based on the selected Tool.
- The **Compile Syncad Libraries** button allows you to compile libraries needed by external simulators and compilers for SynaptiCAD projects.
- In the **Simulator Path** edit box either type in the path name or use the **browse button**  to search for the path.
- The **PLI/VPI Library** setting is for the full path to the PLI or VPI library that the selected tool will use.

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

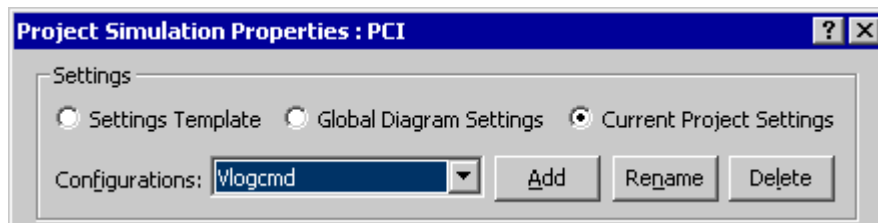


### 10.3 Project Simulation Properties Dialog

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

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

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

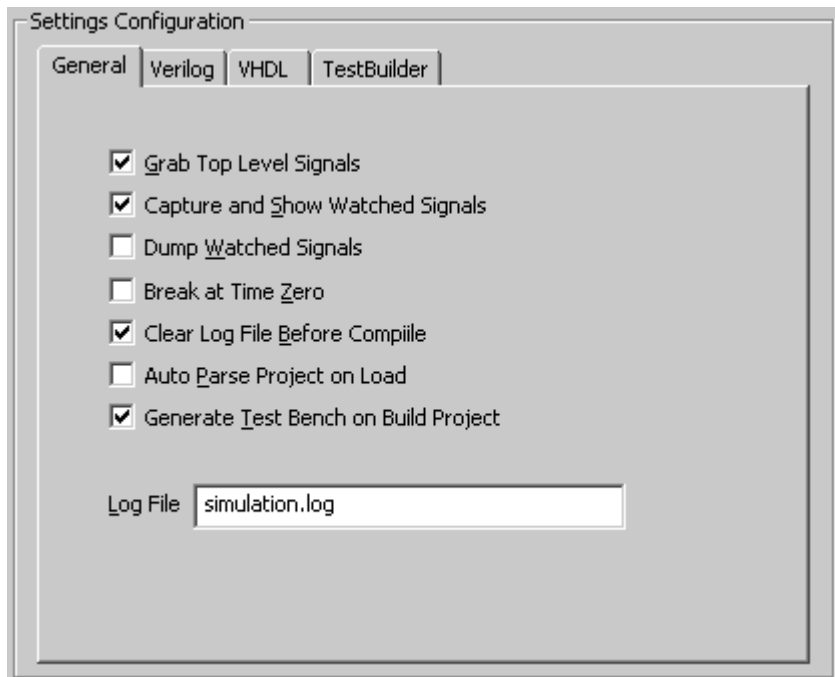


- If the **Settings Template** radio button is selected, then you are editing the default settings that are used by new projects. These are stored in the INI file each time the program is closed. The **Restore Default Templates** button at the bottom of is used to reset the INI file to the factory default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

- By default the *Settings Template* and the *Current Project Settings* use the **Debug Configuration**. If you are moving projects to different machines or if you want to have different settings for debugging and releasing a project you may want to create a new configuration to store the different settings. If you need to define a new configuration:
  - Press the **Add** button to open the *Add New Configuration* dialog, that lets you specify a name and the default configuration to copy the settings from.
  - **Rename** button lets you change the name of the current configuration.
  - **Delete** removes the current configuration.
  - Use the **Configurations** drop-down to choose which configuration you will be editing.

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

- The **Grab Top Level Signals** check box turns on the automatic monitoring of ports or internal signals in the top-level module.
- The **Capture and Show Watched Signals** check box enables the display of waveform results from a simulation run.
- The **Dump Watched Signals** check box will generate a dump file for any watched signals in the diagram. The generated file will have the same name as the .btim file, only with an extension of .VCD.
- The **Break at Time Zero** check box is the equivalent of setting a breakpoint at time zero. This starts the simulator and allows you to enter commands into the console window that will be executed during simulation.



- The **Clear Log File Before Compile** checkbox clears the simulation log just prior to a new compilation being performed. This log maintains compilation notes, as well as some simulation notes. Note that in this dialog you can also change the name of this log (see Logfile below).
- When the **Auto Parse Project on Load** box is checked, user source files are automatically parsed and built when the project is loaded. The top-level module is the first module that is not included by another for Verilog; it is the first entity/architecture pair parsed for VHDL. This is mainly used by Actel Libero customers with WaveFormer Lite.
- The **Generate Test Bench on Build Project** automatically updates the test bench for changes to timing diagrams. Turn this off if you want to temporarily change some of the generated source code manually or to avoid updating the test bench on diagram changes.
- The **Log File** specifies the name of the log file that receives all the simulation results and information. By default TestBench uses *simulation.log*.

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

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

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

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

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

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

```
.v;.vo
```

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

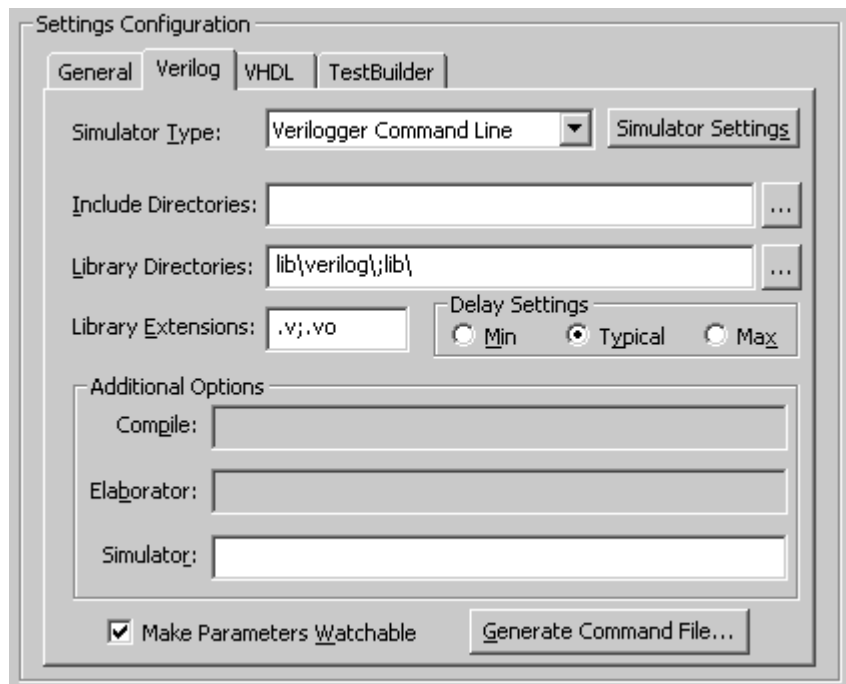
- **Compile**, **Elaborator**, and **Simulator** option edit boxes

allow you to write additional command line options that will be passed to the tool when it is run. Most simulators do not support all three phases of command line options.

- When the **Generate Command File** button is pushed, the text contained in the Simulator Options edit box along with the list of Verilog files specified in the *Project* window are written to a Command File. This file can then be used with the Command Line version of your simulator to run a simulation without the TestBench GUI.
- The **Make Parameters Watchable** determines whether or not parameters will be included with the automatic monitoring of ports and internal signals in the top-level module.

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

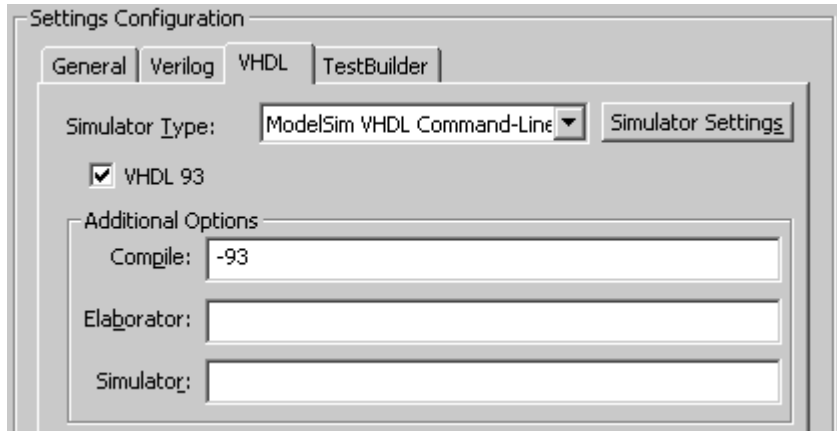
- The **Simulator Type** drop-down determines the simulator.





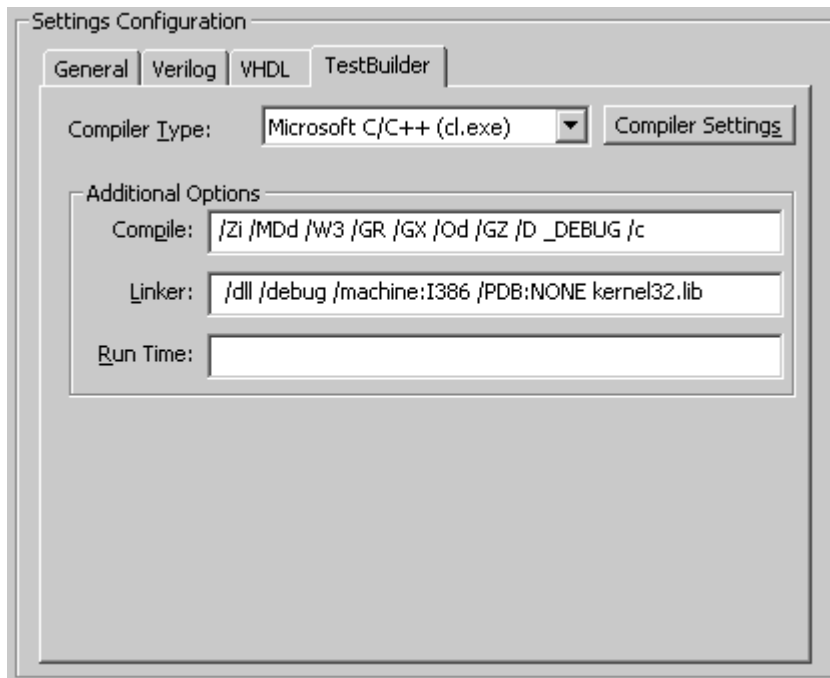
- The **Simulator Settings** button opens the *Simulator / Compiler Settings* dialog where you can edit the simulator paths.
- The **VHDL 93** checkbox specifies that the project dialect for the generated files is **VHDL 93**.

- 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.



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

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



## 10.4 Simulating the Bus Functional Model



The test bench that you have created can now be simulated by taking all of the files produced by TestBencher Pro and placing them into a simulator.

TestBench can also remotely control VHDL and Verilog simulators and C++ compilers. The settings for SynaptiCAD's simulators are controlled through the *Project Simulation Properties* dialog described in *Section 10.3*. The settings for external simulators and tools are controlled using *Simulator/Compiler Settings Dialog* dialog described in *Section 10.2*.


During simulation, if the simulator resides on the same computer as TestBench, the External Program Integration feature will use the project directory for the simulated TestBench project as the working directory for the simulator. If the simulator is on a different computer, the External Program Integration feature will use the directory specified in the **Remote Path** edit box as the working directory for the simulator.

Both the External programs and the internal SynaptiCAD simulators are controlled through the simulation button bar.

#### To build and simulate a bus-functional model:


- Follow the normal steps to generate the bus-functional model (*Section 10.1*).
- Follow the steps in *Section 10.3: Project Simulation Properties Dialog* and *Section 10.2: Simulator and Compiler Settings Dialog* to set up the simulator settings.
- Click the yellow **Compile the Active Project**  button. This builds (parses) the project using the tools specified in the *Project Simulation Properties* and *Simulator/Compiler Settings Dialog* dialogs. For mixed language projects like TestBuilder several tools will be needed to compile different parts of the project. For example, TestBuilder project will require a C++ compiler and an HDL simulator.
- In the *Report* window, look at the **Errors** tab for any compile errors. If there are errors then fix them, regenerate the test bench, and recompile. If there are no errors continue on with the project.
- Run the Simulation. There are three ways to run the simulation within TestBench:
  - Click the **Run** button  on the simulation button bar.
  - Select the **Simulate > Run** menu option, OR
  - Click the <F5> key.



- During Simulation the status of the simulation is displayed in the bottom right hand corner of the TestBench Pro main window. When the simulation is complete **Simulation Good** will be displayed. 
- When the simulation is complete the following information will be displayed:
  - The simulation waveforms will be displayed in the *Stimulus and Results* diagram window.
  - The simulation log file is shown in the *Report* window. Any notes, warnings and errors reported by the simulator will appear in this log.
  - In the *Project* window, the **Simulated Model** folder contains the compiled MUT and the *Stimulus & Results* diagram. If there are any archived stimulus/results files, they will be in the *Stimulus & Results Archive* folder. If any extra files are necessary for TestBench to build the project, they will be automatically added to the project, and contained in the *Compiled Library Files* folder.

#### ModelSim Details

Once the design is loaded into ModelSim, three windows will be opened automatically - the Structure, Signals, and Wave windows. Additionally, all of the top-level signals from the component entity will be placed in the Wave window. These signals will allow you to monitor the MUT and the transactions during simulation.

- To restart the simulation with ModelSim, click the black restart button  on TestBencher's simulation button bar.

## 10.5 Generating Command Files for Third Party Simulators

A command file for Verilog simulators can be generated for a project that will be simulated using a third party simulator. This file can be used when the third party simulator is invoked from the command line. TestBencher can also control the simulators graphically using the techniques described in *Section 10.4: Simulating the Bus Functional Model* and *Section 10.2: Simulator and Compiler Settings Dialog*.

### To generate the command file:

- Select the **Project > Project Simulation Properties...** menu option to open the *Project Simulation Properties* dialog.
- Click the **Generate Command File** button.
- Enter the filename for the command file in the *Filename* edit box. (The file extension will be “.vc” by default.)
- Click the **Save** button to save the file.

To use this file with a third party simulator, you then use the ‘-f’ switch followed by a space and then the filename entered above to simulate the project.

For example, consider a project named test. If the generated command file is named ‘test.vc’ this file would be used with VeriLogger’s command line simulator using:

```
vlogcmd -f test.vc
```

## 10.6 TestBencher Simulation Modes

TestBencher supports two methods for generating test benches: system level BFM generation and unit-level testing. The unit-level testing features are used to create quick stimulus based test benches for smaller designs and models. These features are covered in the VeriLogger Pro on-line help. The system-level features are covered in this manual.

To support these methods of test bench generation, TestBencher has two simulation settings, **Auto Run** and **Debug Run**, that determine how the internal simulator reacts when diagrams are modified. If **Auto Run** is active then changes in the diagram are automatically re-simulated, and if **Debug Run** is active then simulations are run only when specified by the user. The simulation settings are important if you are using simulated signals to construct the Transaction Diagrams. TestBencher should be set to **Debug Run** when you are developing new timing diagrams.

Set your simulation mode to **Debug Run**:

- If the mode button is set to **Auto Run**, click the button to toggle the mode to **Debug Run**. The mode button is located on the simulation toolbar, below the main menu in TestBencher.



# Chapter 11: Test Bench Techniques

TestBench can generate bus-functional models that represent many different bus transactions and functions. Here we have gathered some of the techniques that we use to model different types of functionality.

## 11.1 Master and Slave Transactions

Many bus specifications are described using the terms Master and Slave transactions. Master transactions are usually applied to the MUT, perform a function and then stop executing. And slave transactions run in a continuous looping mode. A slave waits for a certain set of conditions to become true so it can perform a function, and then return to the waiting mode.

For TestBench, any timing diagram can be a Master or a Slave transaction based on the **Execution** setting in the *Diagram Settings* dialog (see *Section 3.8: Diagram Settings Dialog - General Tab* for more information about the Diagram Execution settings). TestBench generates appropriate transaction calls for each timing diagram. The following table describes the typical settings available for master and slave transactions:

Master Run Once	Apply
Master in Pipeline	Apply-nowait
Stop Master	Abort
Slave and Clock Processes	Apply-nowait
Stop Slave and Clock Processes	Abort

*Section 9.1: Transaction Calls* has more information about the *Insert Diagram Subroutine Call* dialog. The PCI example in the **Examples** directory has both the master and slave diagrams drawn for the specification. You can load the project for your particular language and experiment with the transactions.

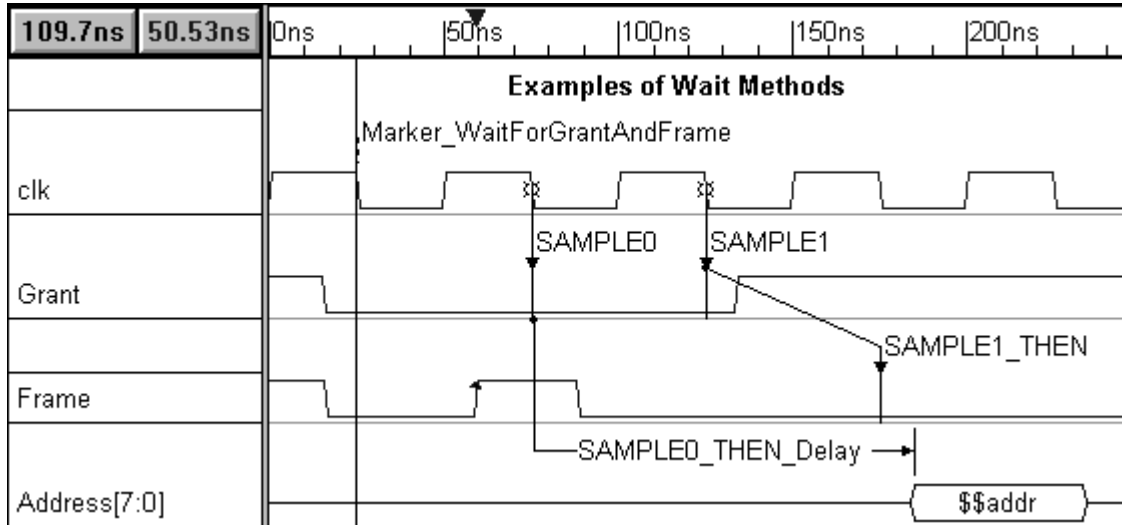
## 11.2 Waiting for Signal Transitions

You can use either samples or markers to make a transaction wait for an event or series of events before continuing to execute. Below is a chart of the different methods of waiting and the recommended usage for each method. *Chapter 6: Transaction Samples* and *Chapter 7: Transaction Markers* have more information about Samples and Markers.

Wait On	How Long	Construct Used	Settings
One or more conditions (e.g., signal states)	Block until all conditions are true	Wait Until Marker	
One event or condition	Block until time out	Sample with Multiplier	<i>min</i> == <i>max</i> <i>multiplier</i> > 1 check <i>Blocking</i> unchecked <i>Full Expect</i>
One event or condition	Block until time out	Sample with window	<i>min</i> != <i>max</i> <i>multiplier</i> == 1 check <i>Blocking</i> unchecked <i>Full Expect</i>
One event	Block indefinitely or until diagram times out	Sensitive Edge	

Wait On	How Long	Construct Used	Settings
Several events or conditions across several clock cycles	Each sample may block with time out	Several samples chained together (first samples will block subsequent samples)	check <i>Blocking</i> unchecked <i>Full Expect</i>

Below is an example of timing diagram that demonstrates these techniques for waiting.



- The Marker called **Marker\_WaitForGrantAndFrame** is a *Wait Until Marker* type with the condition of **(Grant===0 && Frame===0)** {the condition code is in the generated language, in this example Verilog}. This marker will block the transaction until the condition becomes true.
- The rising edge on **Frame** is sensitive. This will cause the diagram to wait for that edge to occur.
- The Sample called **Sample0** is setup as *blocking* and *non-full expect* with a multiplier of 3. The Multiplier is the sample's time out. Checking the *blocking* box causes the sample to block the triggering clocked sequence until it times out or until the condition becomes true. Disabling the *Full Expect* box means that the sample will not expect the drawn condition to be true during the entire window. Instead it will continue sampling as long as the condition is NOT true and the time out has not been reached.
- This sample also has a conditional delay, **SAMPLE0\_THEN\_Delay**, so that when it passes it will cause the value passed into `$$addr` to be written out to the Address signal. If **Sample0** times out then the Address signal never gets driven.
- The samples **Sample1** and **Sample1\_THEN** check for Grant and Frame to be true over successive clock cycles. They are defined using the same settings as Sample0 in the previous example except the multiplier is set to 1.

**Note:** If a Sample has a multiplier of 1 and no window defined at simulation then the **blocking** check box has no effect on the behavior of the Sample. The Sample will execute and then immediately pass or fail depending on the condition.

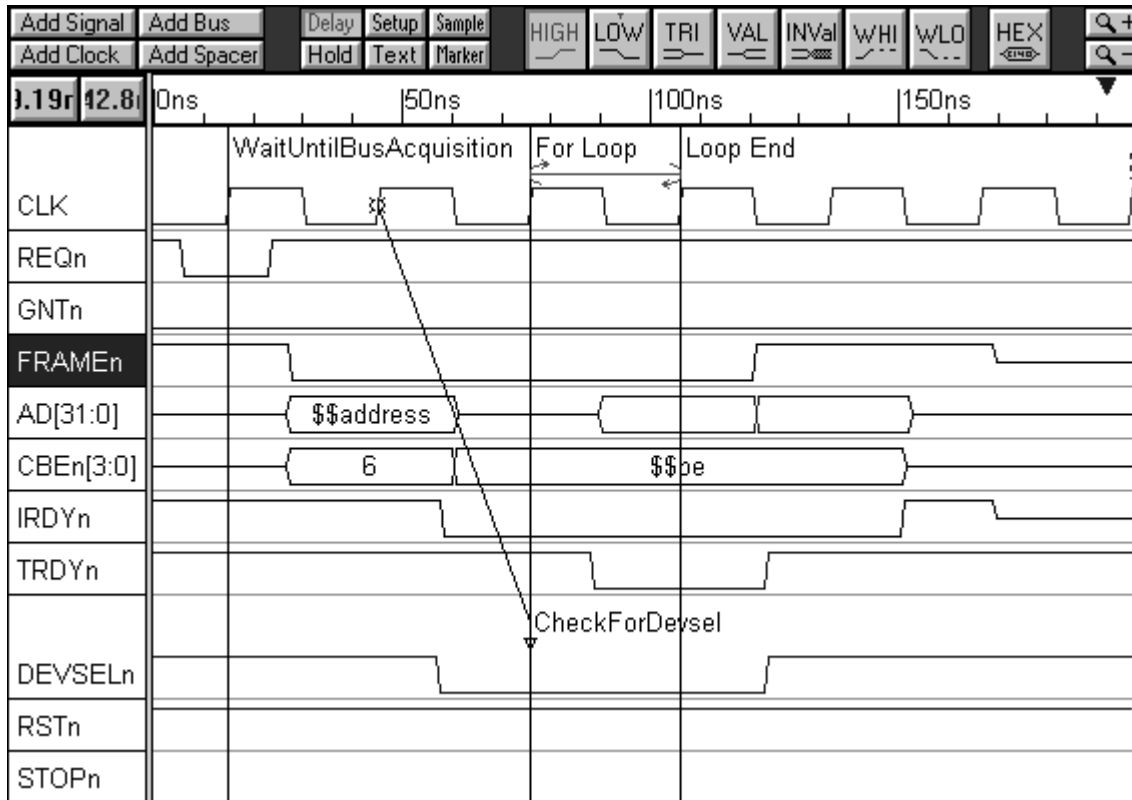
### 11.3 Burst Mode Transactions

Burst Mode Transactions usually process 1 or several cycles depending on information received at simulation time. TestBench can model burst mode transactions using a combination of marker loops, parameter variables, and samples. The PCI example projects that are shipped with TestBench have examples of both master and slave burst mode read transactions. The basic design for a burst mode transaction is as follows:

- 1) Draw a timing diagram that represents the smallest possible transaction.

- 2) Use a marker loop to define the beginning and ending of the repetitive section.
- 3) If it is a master transaction, use an input variable to pass in the number of cycles to process. This variable will be used in the condition statement of the Marker loop.
- 4) If it is a slave transaction, you will use one or more samples chained together to determine if the burst transaction should continue.

Below is an example of a simple burst mode master read cycle.



- The marker, **WaitUntilBusAcquisition**, will cause the transaction to wait until the condition ( $GNTn === 0 \ \&\& \ FRAMEn === 1 \ \&\& \ IRDYn === 1$ ) is true (Section 7.4: Wait Until Markers).
- The sample, **CheckForDevsel**, checks the  $DEVSELn$  signal and if it is active low then the transaction proceeds normally. If it fails the transaction aborts (Chapter 6: Transaction Samples).
- A diagram variable called  $dataCycles$  is defined as a transaction input with a type of integer. When the transaction is called,  $dataCycles$  will carry the number of cycles that the master read will execute (Section 8.3: Variables). The For Loop uses the  $dataCycles$  variable to determine how many read cycles to perform (Section 7.5: Loop Markers).
- Two state variables,  $address$  and  $be$ , are also transaction inputs (Section 4.3: Driving Waveform States with Variables).

## 11.4 Conditionally Moving Signal Edges (Sweep Tests)

Sometimes it is desirable to move an input signal edge each time it is applied to the MUT. This allows testing the MUT under different timing conditions. Sweep tests can also be performed that move the edge until a setup failure is generated.

Delays are used to move and conditionally trigger signal edges. The min and max values of a delay specify either a fixed time or the number of clocking signal edges between two signal transitions. These values can be defined as variables that can be passed into the delay each time it executes. *Chapter 5: Transaction Delays, Setups, and Holds* has information on adding and editing delays. There are two types of variables that can be used to drive the min and max value: parameter variables and diagram-level variables.

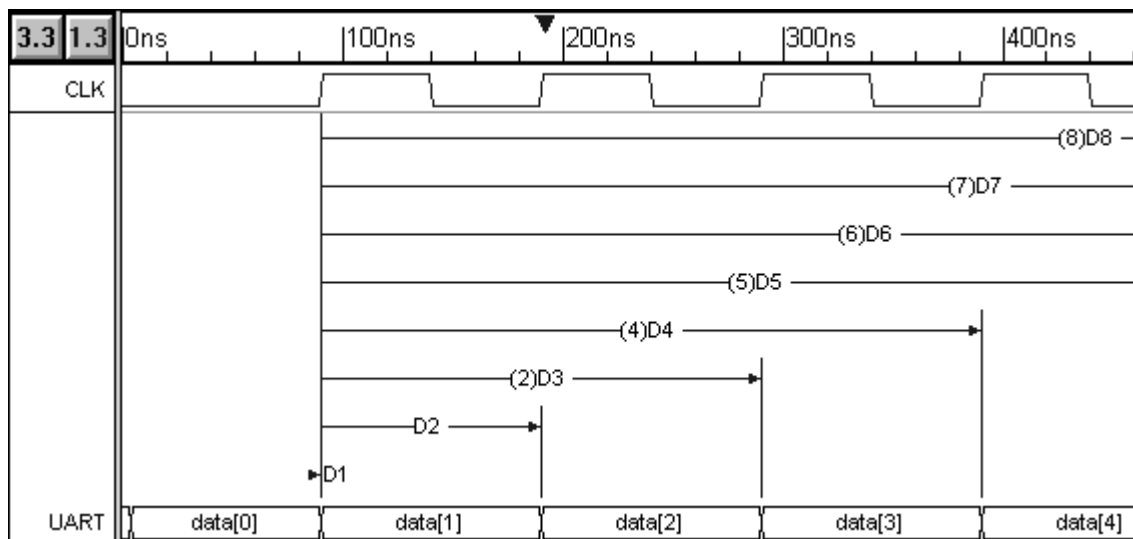
- If the delay value is to be passed into the transaction, check the **Is Apply Subroutine Input** checkbox in the *Delay Properties* dialog. This will create two variables named **delayName\_min** and **delayName\_max**. These variables will be inputs to the transaction and will automatically drive the value of the delays (*Section 5.2: Delays*).
- If the delay value is to be calculated within the transaction, create a diagram-level variable using the **Variables** button in the diagram (*Section 3.3: Transaction Level Variables* and *Section 8.3: Variables*). Type the name of the variable into the min or max value of the delay.

For sweep tests you will want to monitor the reaction of the MUT using either a setup, a continuous setup signal, or a sample, depending on exactly what you want to check:

- Graphical setups monitor the distance between two specific edges in a transaction. (*Section 5.3: Setups and Holds*).
- Continuous setups monitor all of the edge transitions between any two signals in a timing diagram. A continuous setup or hold is created using the *Advanced Register and Latch Controls* in the *Signal Properties* dialog as described in *Section 5.5: Creating Continuous Setups and Holds*.
- A windowed, full-expect sample can be used monitor the MUT response. This type of sample checks to make sure that the MUT signals transition within the correct time period (*Chapter 6: Transaction Samples*).

## 11.5 Reading and Writing Serial Data

To read and write serial data you will use diagram-level variables and manipulate the data coming in or going out of the transaction. The UART (Universal Asynchronous Receiver/Transmitter) project in the *Examples* directory is an excellent example of manipulating serial data.



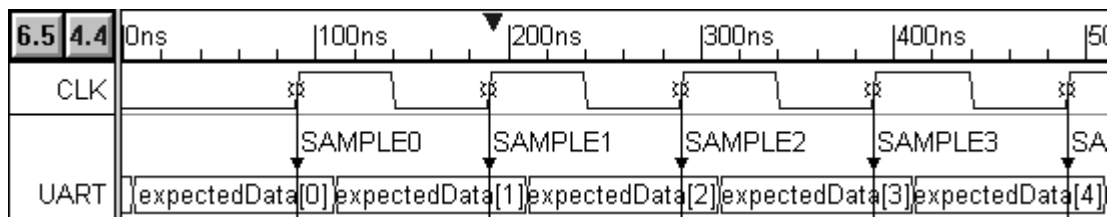


The segment of the UART write timing diagram shown above takes a byte it receives from the top-level project and serializes the data to the MUT. In this diagram we have:

- Defined a diagram-level variable *data* that is an input to the transaction that will hold the byte data to serialize. Also a parameter variable called *speed* that determines the number of clock cycles needed to write out a bit of data.
- On each clock cycle (or number of clock cycles) a new data bit is driven to the MUT. The min values of the delays are defined using equations with the variable *speed* that controls how many clock cycles that they should delay.

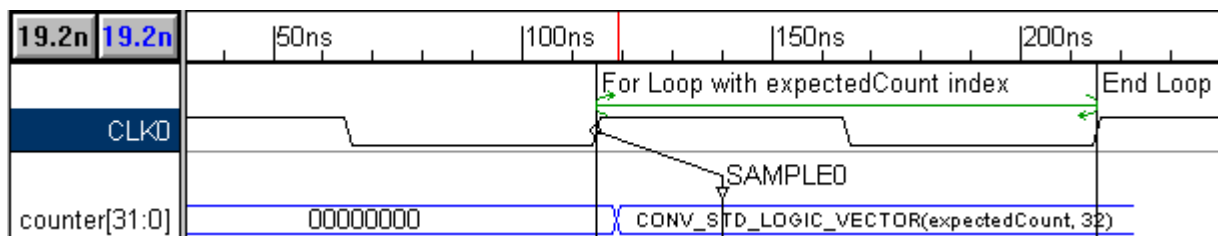
Below is a section of the UART read timing diagram converts a serial bit stream from the MUT into bytes that are passed out of the transaction. In this diagram we have:

- Defined two diagram-level variables: an output variable called *data* to hold the data coming back from the MUT, an input variable called *expectedData* to compare against the actual data. Also a parameter variable called *speed* that determines the number of clock cycles needed to read a bit of data.
- On each clock cycle (or number of clock cycles) a new piece of data is read in. The *speed* variable is used to define the sampling window for each *full-expect, blocking* sample. At the end of the sample window, each sample compares the actual data with the expected data and logs any errors. The samples also store the actual data into the correct bit of the *data* variable. This is done using the *Store Sample Value to Variable* section of the sample's *HDL Code Generation* dialog (Section 6.2: *Storing Sample Values in User Defined Variables*).
- Since *data* is an output to the transaction it is passed out of the transaction to the top-level project.



## 11.6 Testing a Counter Model

Testing a counter with a reactive test bench is a lot easier than it is with traditional stimulus based test benches. The test bench can be designed with just a small two cycle timing diagram with a loop. Below is an image of a diagram that tests a 32-bit counter.



Discussion of the Counter Test Bench:

- **Initialize the counter:** the first cycle in the diagram is used to initialize the starting value of the counter.
- **Counter Loop:** two loop markers surround the second cycle. The first marker starts the For-Loop and initializes an index variable called *expectedCount*. Each time through the loop *expectedCount* will be incremented. The For-loop is defined in the *Marker* dialog of the first marker.

- **Expected Counter Output:** The signal `counter` is blue to indicate that it is the output of the model under test and an input to the test bench. Each time the counter is incremented we expect the counter model to increment and to be equal to the index of the For-Loop. The `SAMPLE0` compares the actual simulation output with the value generated by the test bench. The bus state of the counter signal contains code that defines how the model output should change with each loop. It is language dependent:

**VHDL:** The image shows a VHDL test bench that converts integer *expectedCount* to a 32-bit standard logic vector. The `CONV_STD_LOGIC_VECTOR` is necessary because VHDL does not automatically convert integers to standard logic vectors.

**Verilog:** The code would just be `expectedCount`, because the language is able to automatically do the conversion.

## 11.7 External Model Support

TestBencher's BFM's can be used with external models from any of the languages supported by TestBencher, with other TestBencher BFM's, and with TestBencher Reference Models. TestBencher automates the process of hooking up the project BFM to other models that exist in your system by using information contained in the different folders of the project.

### VHDL and Verilog Models

External VHDL or Verilog models are first added to the **User Source Files** folder and then the **Extract MUT Ports** function extracts the model information and makes it available to TestBencher. This is discussed in *Chapter 1: Step 2 Add the MUT to the Project* and *Step 3: Extract Port Information*.

### Other TestBencher BFM's

Other TestBencher BFM's are first added to the **Project Library** folder and then instantiated using the context menu functions. This is *Section 2.3: Sub-Projects* and *Section 2.4: Component Instances of Sub-Projects*. The BFM's should be of the same generation language as the main TestBencher project, so in this case they should all be **TestBencher** BFM's.

### TestBencher Reference Models

TestBencher can generate golden reference models that run in parallel with a VHDL or Verilog RTL model. Golden reference models are high-level behavioral descriptions of a design and are used to compare to the results of an RTL-level model during simulation. When this feature is turned on, TestBencher handles all of the code generation and hook-up of the model. This is covered in *Section 2.7: Golden Reference Models*.

## Chapter 12: Language Specific Details

TestBencher generates test benches from language independent timing diagrams. Even though the timing diagrams look the same, the implementation may be significantly different from language to language. This is why we have refrained from discussing implementation earlier in the manual. The template file code is dependent on the language. This chapter discusses how the transactions execute, the generated files needed for simulating the model, and the transaction manager post and apply file calls.

### 12.1 Verilog

This section lists the files generated for pure Verilog models. Combination projects using Verilog and the C++ TestBuilder library are covered in the TestBuilder section. Some advanced features like Transaction Manger generation, random constrained data generation, and advanced data structures are supported using the combined TestBuilder and Verilog languages.

#### Files needed for simulation:

***projectName.v*** – contains top-level module for project. This is the Component Model file (the expanded template file).

One timing transaction file for each timing diagram in the project:

***projectName\_diagramName.v*** – contains transaction module.

All of the MUT files listed in the **User Source Files** folder in the *Project* window.

***syncad.v*** - contains base constants that are used throughout the generated code.

***wavelib.v*** - (optional) contains register and latch module definitions that are used by timing diagrams with clocked “Simulated” signals. If there are no clocked simulated signals in the entire test bench then this file is not needed.

Library files - When a new project is created a lib/verilog directory is created in the project directory that contains various modules that may or may not be used by the project. Currently, these libraries consist of different clock models that are used based on the types of output clocks used in the timing diagrams.

***projectName\_emulator.v*** - (optional) If using a reference model (see *Section 2.7: Golden Reference Models*) then a file named *projectName\_emulator\_skeleton.v* is generated and placed in the Associated Files folder for the Component Model. This file should be copied to another file named *projectName\_emulator.v*, placed into the User HDL Files folder and modified to represent the reference model.

***tb\_projectName\_tasks.v*** - (optional) If using File I/O then this file will be generated and added to the Associated Files folder for the Component Model.

***tb\_libraryName\_user\_classes.v*** - (optional) one file is generated for each imported (Non-default) library in the project hierarchy. All of these library packages are placed in the Associated Files folder for the Component Model of the top level project during test bench generation.

#### Accessing Project Level Variables from a Diagram in Verilog

In Verilog, only variables that are instances of user defined classes are accessible from the diagram. In these cases, the variable is just referenced by its name. An upward search through the scopes will be done to find the variable instance. For example, if you had an **ATMCell** class definition and an instance named **cell** either at the project-level or the diagram-level, then you could access the **HEC** field of **cell** by using **cell.HEC** syntax.

#### Data Packing in Verilog

When packing is enabled for at least one field of a class definition, that class will have an additional field that is an array of bytes which is used during packing operations. This array is called **packed\_array**. The **pack** task will pack the fields to **packed\_array**. The **unpack** task will fill the data fields of the class with data from **packed\_array**. For ex-

ample, if you had a variable named **VAR0** that was an instance of a class definition with fields that had packing enabled, you would call **VAR0.pack** to pack those fields into **VAR0.packed\_array**. This byte array could then be accessed by an index to drive a signal. If you were reading data from a bus you would use **Sample(s)** to store the data one byte at a time into the **packed\_array**. Then once you were done reading, you would call **VAR0.unpack** to convert the data to its class form.

## 12.2 VHDL

This section lists the files generated for pure VHDL models. Combination projects using VHDL and the C++ TestBuilder library are covered in the TestBuilder section. Some advanced features like random constrained data generation, and advanced data structures are supported using the combined TestBuilder and VHDL languages. Also several files are generated for top-level projects that are not generated for sub-projects. These files are marked as **top-level project only**.

### Files needed for simulation:

**projectName.vhd** - contains top-level module for project. This is the Component Model file (the expanded template file).

One timing transaction file for each timing diagram in the project:

**projectName\_diagramName.vhd** – contains transaction module.

All of the MUT files listed in the **User Source Files** folder in the *Project* window.

**syncad.vhd** - (top-level project only) contains package declaration and body that has various types, functions, and procedures defined that are used throughout the generated test bench code.

**wavelib.vhd** - (top-level project only) contains register and latch module definitions that are used by timing diagrams with clocked “Simulated” signals. If there are no clocked simulated signals in the entire test bench then this file is not needed. The files in the User HDL Files folder.

**tb\_control\_types.vhd** - (top-level project only) contains signal and variable declarations that are used by the apply call procedures throughout the test bench.

**tb\_diagram\_types.vhd** - (top-level project only) contains package that defines enumerated type for diagram types. Each diagram in project hierarchy is given a different value.

**tb\_transaction\_manager.vhd** - (top-level project only) contains entity/architecture for transaction manager (if enabled anywhere within project hierarchy). This transaction manager is instantiated once in each project that has a transaction manager enabled.

Associated Library files that are used by TestBencher. When a new project is created a lib/verilog directory is created in the project directory that contains various modules that may or may not be used by the project. Currently, these libraries consist of different clock models that are used based on the types of output clocks used in the timing diagrams.

**tb\_projectName\_tasks.vhd** - (optional) If using File I/O then this file will be generated and added to the Associated Files folder for the Component Model.

**tb\_projectName\_parameters.vhd** - (optional) contains package that encapsulates the parameters that are passed into each diagram. This package is used by the apply calls to communicate data to/from diagrams.

**tb\_projectName\_user\_classes.vhd** - (optional) this file contains record type declarations for user defined classes that exist in the project. This file is added to the Associated Files folder for the Component Model during test bench generation.

**tb\_libraryName\_user\_classes.vhd** - (optional) one file is generated for each imported (Non-default) library in the project hierarchy. All of these library packages are placed in the Associated Files folder for the Component Model of the top-level project during test bench generation.

TestBench assumes that your VHDL compiler includes pre-written libraries for packages **std\_logic\_1164**, **std\_logic\_textio**, and **std\_logic\_arith**. If you do not have these library files, you will need to acquire them. (SynaptiCAD can provide these files – contact our sales department.) Analyze (compile) these files and run your simulator.

### Procedures for Apply Calls, Transaction Manager, and File I/O

The procedures generated for diagram Apply Calls and Transaction Manager calls have similar syntax and parameters. These procedures can be used in the project template file. In VHDL there are two parameters, **tb\_control** and **instancePath**, that are used to control the transactions. If you type in the diagram calls by hand you must pass in values for these parameters.

#### Diagram Apply Calls

The diagram apply calls are defined in the **tb\_projectName\_tasks** package. These are usually added to the template file using the *Insert Diagram Calls* dialog (*Section 9.1: Transaction Calls*). However you may type in the apply calls by hand. To apply a diagram that is not in the current project, specify the package path when calling the apply call (i.e. `work.tb_projectName_tasks.Apply_diagramName(...)`). An alternate method is to add a *use clause* statement at the top of the component model that brings in the task package for the project that contains the diagram. If you use the *Insert Diagram Calls* dialog this is handled for you. Each timing diagram generates several apply call procedures:

#### Master Transaction Apply Calls:

```
Apply_diagramName ( signal tb_Control : inout tb_Control_Type;
                    instancePath : in string;
                    <parameters> );

Apply_diagramName_nowait ( signal tb_Control : inout tb_Control_Type;
                           instancePath : in string;
                           <parameters> );

Abort_diagramName ( signal tb_Control : inout tb_Control_Type;
                   instancePath : in string);
```

#### Slave Transaction Apply Calls:

```
Apply_diagramName_looping ( signal tb_Control : inout tb_Control_Type;
                            instancePath : in string;
                            <parameters> );

Apply_diagramName_looping_nowait( signal tb_Control : inout tb_Control_Type;
                                   instancePath : in string;
                                   <parameters> );

Abort_diagramName ( signal tb_Control : inout tb_Control_Type;
                   instancePath : in string);
```

#### Transaction Manager Procedures

When the transaction manager is enabled for a project there are additional procedures defined in the **tb\_projectName\_tasks** and the **tb\_control\_types** packages (*Section 9.3: Transaction Manager and Test Reader*). These procedures allow you to post the apply calls to any transaction manager in the project using an interface that is very similar to the normal apply calls. Basically, you just need to specify which transaction manager you want to post to. There are also procedures that allow you to apply a file that contains a list of transactions to post to the transaction manager. Listed below are the transaction manager procedures.

**Procedures for posting transactions individually to the transaction manager (defined in *tb\_projectName\_tasks.vhd*):**

**Master Transaction Manager Apply Calls:**

```

Post_diagramName ( signal tb_Control : inout tb_Control_Type;
                    TransactionManagerPath : in string;
                    InstancePath : in string;
                    <parameters> );

Post_diagramName_nowait ( signal tb_Control : inout tb_Control_Type;
                           TransactionManagerPath : in string;
                           InstancePath : in string;
                           <parameters> );

Post_Abort_diagramName ( signal tb_Control : inout tb_Control_Type;
                          transactionManagerPath : in string;
                          instancePath : in string );

```

**Slave Transaction Manager Apply Calls:**

```

Post_diagramName_looping ( signal tb_Control : inout tb_Control_Type;
                            TransactionManagerPath : in string;
                            InstancePath : in string;
                            <parameters> );

Post_diagramName_looping_nowait ( signal tb_Control : inout tb_Control_Type;
                                    transactionManagerPath : in string;
                                    instancePath : in string;
                                    <parameters> );

Post_Abort_diagramName ( signal tb_Control : inout tb_Control_Type;
                          transactionManagerPath : in string;
                          instancePath : in string );

```

**Procedures for posting transactions from files (defined in *tb\_control\_types.vhd*)**

```

ApplyFile ( signal tb_Control : inout tb_Control_Type;
             fileName : string;
             transactionManagerPath : in string )

```

This procedure applies all of the transactions listed in the file. It will not return to the calling process until all transactions have been read from the file and placed into the transaction manager's queue. If the file contains all concurrent (nowait) transactions then this function will return in zero simulation time since the transaction manager will start all of the transactions without blocking the next one.

```

ApplyFile_nowait ( signal tb_Control : inout tb_Control_Type;
                   fileName : string;
                   transactionManagerPath : in string )

```

This procedure will perform the same actions as `ApplyFile` except that it returns to the calling process immediately regardless of the type of apply calls listed in the file.

Procedures used to handle transaction manager mode (defined in `tb_control_types.vhd`). Note: At the printing of this manual the following procedures were not supported. Check the on-line documentation for actual implementation.

```
SetApplyCallMode( transactionManagerPath : in string; mode : in TStatus )
```

Sets the mode of the transaction manager. By default it is set to looping so that transactions are executed when they are available in the queue. The **mode** can be any of the following:

TB\_ONCE - only run the next transaction from queue when **ApplyNextTransaction** is called.

TB\_LOOPING - run the next transaction from the queue whenever there are apply calls in the queue.

TB\_SUSPEND - finish running the current transaction and stop reading apply calls from the queue until the apply call mode is changed.

```
ApplyNextTransaction( transactionManagerPath : in string )
```

Applies the next transaction from the queue if the apply call mode is set to TB\_ONCE. Otherwise, calling this function has no effect on the transaction manager.

### Parameters for Apply Calls and Transaction Manager Procedures

**tb\_Control** – is a global record that contains control signals needed to trigger transactions. Pass this record into the apply or post calls.

**instancePath** – the hierarchical path to the project which contains the transaction you wish to apply. You can type in the full instance path or use the generic **tb\_InstancePath** that contains the project instance path for transactions relative to the current entity.

**transactionManagerPath** - path to project that contains the transaction manager that you want to post the apply call to. Typically, this will be **tb\_InstancePath** which is a generic available in all project components and diagrams that specifies the full instance path to the current scope. So, `tb_InstancePath` would be used if you want to post the apply call to the transaction manager that is owned by the project in which you're calling the Post method from.

**fileName** – the name of the file on disk that contains the list of apply call data.

**<parameters>** - there will be additional parameters to the apply calls based on the type of parameters that exist for your diagram. Note that the “nowait” apply call procedures will NOT contain any parameters that are outputs from the transaction since the apply call will return immediately to the calling process once the diagram starts.

### Examples of Apply Calls and Transaction Manager Procedures

Example 1. Apply diagram contained in local project

```
Apply_write( tb_Control, tb_InstancePath, x"AB", x"EF" );
```

- writes data EF to address AB

Example 2. Apply diagram in another project by specifying relative instance path to current project

```
work.tb_master_tasks.Apply_write( tb_Control, tb_InstancePath&"master0",  
                                  x"AB", x"EF" );
```

Example 3. Apply diagram in another project by specifying full instance path from root

```
Work.tb_master_tasks.Apply_write( tb_Control, "PCI.master0", x"AB", x"EF" );
```

### Transaction Generator

If the Transaction Manager is enabled then the following functions and weighting table will be generated for the project. These are covered in *Section 9.5: Transaction Monitor*.

- The weightings table is a matrix of the master transactions. The order of the transactions is the order in which they appear in the Project window. Copy this out of the generation macro to the code section of the Sequencer process before editing. An example table might look like:

```
good := SetTransactorWeightings( tb_InstancePath, ((0, 5, 1),
                                                    (0, 1, 1),
                                                    (0, 1, 1)));
```

- The **SetTransactorWeightings** function registers the weightings matrix. You can change the weightings at any time during the simulation.
- The **RunRandomTransactor** function randomly creates a transaction call and hands it to the Transaction Manager to put on the transaction queue. Each time a transaction is created with this function the transaction arguments are randomly generated using the constraint settings for the variables. Usually this function is used inside of a loop for example:

```
for i in 0 to 9 loop
    transactor := GetRandomTransactor(tb_InstancePath, transactor);
    PostRandomTransaction(tb_Control, tb_InstancePath,
                        T<projectName>MasterTransactor' image(transactor));
end loop;
```

### Accessing Project Level Variables from a Diagram in VHDL

In VHDL, from a diagram, you have access to any variable that is defined in the owning project. To get access to a project level variable, use the following syntax:

```
work.tb_projectName_parameters.tb_ProjectVariables(tb_ProjectID).variableName
```

Note: **tb\_ProjectID** is an integer variable that is defined in the project. Each project in the hierarchy receives a unique project id during initialization of the simulation.

For example, if you had a project named **master** and a project level variable named **address**, you enter something like the following:

```
work.tb_master_parameters.tb_ProjectVariables( tb_ProjectID ).address
```

### Data Packing in VHDL

The following two functions are generated in the user class package when packing is enabled for a class:

```
function pack( dataStructure : className ) return std_logic_vector;
function unpack( packed_data : std_logic_vector(packed_size downto 0) )
    return className;
```

**className** - a variable that is an instance of a user defined class.

**packed\_size** - the sum of all the packed field bit sizes.

When using the packing options in VHDL, you typically will want to create a variable of type **std\_logic\_vector** that has a size that matches the total number of bits in all of the packed fields. This variable is then used when packing and unpacking.

## 12.3 TestBuilder

This section lists the files generated for pure TestBuilder models. For combination TestBuilder and VHDL or Verilog projects, the top-level files and transaction calls will be the same as the pure TestBuilder projects, but the transaction-level files will be in the native VHDL and Verilog languages.



**Files needed for simulation:**

**projectName.cpp/h** – contains top-level module for the project. This is the component model file (the expanded template file).

For pure TestBuilder projects: one timing transaction file and header file for each timing diagram in the project:

**projectName\_transactionName.cpp/h**– contains transaction module.

All of the MUT files listed in the **User Source Files** folder in the *Project* window.

**syncad\_tb.cpp/h** – contains base classes and enumerated types that are used through out the generated code.

**tbvMain.cpp** - (top level project only) contains tbvMain function which is the entry point for TestBuilder. The tbvTvmTypes array is also defined in this file.

**projectName\_library.cpp/h** – contains the user defined classes defined in the project library.

**libraryName\_library.cpp/h** – one file is generated for each imported library in the project hierarchy. All of these library packages are placed in the Associated Files folder for the Component Model of the top-level project during test bench generation.

**tb\_syncad\_classes.cpp/h** – contains common operators used by user defined classes and variables throughout the project hierarchy.

**projectName\_emulator.cpp/h** – If using a reference model (see *Section 2.7: Golden Reference Models*) then a file named *projectName\_emulator\_skeleton.cpp* is generated and placed in the Associated Files folder for the Component Model. This file should be copied to another file named *projectName\_emulator.cpp*, placed into the User HDL Files folder and modified to represent the reference model.

**Transaction Parameters for Apply Calls**

With TestBuilder, the parameters for a transaction apply call are placed into an arguments class of type **TProjectName\_diagramName\_Args**. Each apply call has an instance of the arguments class named **DefaultArgs** that can be used to pass information to the apply call. You can also define other instances and pass them into the **run** method of the apply call. This is useful when using randomization routines.

For example, consider a project with a timing diagram called **write** that has parameters **data** and **addr**. To call the transaction and pass in a data value of 0xAE, and an address value of 0xF0 you would use the following calls:

```
Tvm.Apply_write.DefaultArgs.addr = 0xF0;
Tvm.Apply_write.DefaultArgs.data = 0xAE;
Tvm.Apply_write.run();
```

**Generating Random Values Using TestBuilder**

TestBuilder provides the capability of generating random values for variables, class instances, and apply call parameters. Generation of these values is performed in the *Sequencer Process* of the *Component Model* in the TestBuilder template file. The values that are generated when the randomize method is called will conform to the constraints that are defined for the data members being randomized. See *Section 8.8: Constrained Random Number Generation* for more information on constraints.

To randomize a variable or class instance:

- Double-click on the **Component Model** in the *Project* window to open the template file.
- Scroll down to the **Sequencer Process**.
- Add a call to randomize on the parameter that you wish to randomize. Note: this method can be called on the entire instance of any class or on specific fields of a class instance.

Example. Write random data to a random address.

```
Tvm.Apply_write.DefaultArgs.randomize();
```

```
Tvm.Apply_write.run();
```

### TestBuilder Methods for use with Transaction Manager

The following methods are available for the TestBuilder Transaction Manager. The post diagram calls insert the transaction into the Transaction Manager's queue. The apply file calls hand an entire file to the transaction manager.

**Diagram Method Calls:** For each diagram there are three transaction classes (once, looping, and abort). For each of the transaction classes there are two diagram methods are defined: *Post* and *Post\_nowait*. The post diagram calls insert the transaction into the Transaction Manager's queue. To call one of these methods from the template file use either the relative diagram name or the instance name of the subproject (ie.

**tvm.Apply\_diagramName\_looping.Post(...)** or **tvm.projectInstanceName.Apply\_diagramName\_once.Post(...)**.

```
Post (TransactionManager, <parameters> );
```

```
Post_nowait( TransactionManager, <parameters>);
```

**TransactionManager** - For each project with a transaction manager enabled, there will be a model called **TransactionManager** that is instantiated in the component model. If you are posting to the manager in the current project use **TransactionManager** for this parameter. If you are posting to a subprojects manager use **subprojectInstanceName.TransactionManager**.

**<parameters>** - there will be additional parameters to the apply calls based on the type of parameters that exist for your diagram. Note that the "nowait" apply call procedures will NOT contain any parameters that are outputs from the transaction since the apply call will return immediately to the calling process once the diagram starts.

For example, consider a project with a timing diagram called **write** that has parameters **data** and **addr**. To post a write transaction to run once you would use the following calls:

```
Tvm.Apply_write.DefaultArgs.addr = 0xF0;
```

```
Tvm.Apply_write.DefaultArgs.data = 0xAE;
```

```
Tvm.Apply_write.Post( TransactionManager );
```

### Transaction Manager Methods

```
ApplyFile ( const char* filename )
```

This method sets the file name for which the transaction manager will pull apply calls from. This method is blocking. This method will not return until the last apply call in the file has been placed on the transaction manager's queue.

```
ApplyFile_nowait ( const char* filename )
```

This method performs the same actions as **ApplyFile** except that it is spawned off in another process and returns control to the calling method immediately.

```
SetApplyCallMode ( TApplyCallMode mode )
```

Sets the mode of the transaction manager. By default it is set to looping so that transactions are executed when they are available in the queue. **TApplyCallMode** can be any of the following:

**TB\_ONCE** – If this is set then **ApplyNextTransaction()** controls when apply calls are taken off of the transaction manager's queue and applied.

**TB\_LOOPING** - run the next transaction from the queue whenever there are apply calls in the queue.

**TB\_SUSPEND** - finish running the current transaction and stop reading apply calls from the queue until the apply call mode is changed or **ApplyNextTransaction** is called.

```
ApplyNextTransaction()
```

If the apply call mode is set to **TB\_ONCE** then this will cause the next apply call on the queue to be pulled off and run.

```
Post_ApplyCall ( tvbTaskT* const taskP )
```

This method is not normally used directly by user written code. It is used within the diagram port methods to post apply calls to the transaction manager. This method could be used directly if you create the tvbTaskT object manually. Each transaction has three corresponding classes defined that derives from tvbTaskT and are named **TApply\_projectName\_diagramName**, **TApply\_projectName\_diagramName\_looping**, **TAbort\_projectName\_diagramName**.

### Transaction Generator

If the Transaction Manager is enabled then the following functions and weighting table will be generated for the project. These are covered in Section 9.5.

- The weightings table is a matrix of the master transactions. The order of the transactions is the order in which they appear in the Project window. Copy this out of the generation macro to the code section of the Sequencer process before editing. An example table might look like:

```

// to | |
// reset | write | read
int weightings[3][3] = { { 0, 5, 1 }, // from reset
                        { 0, 1, 1 }, // from write
                        { 0, 1, 1 } }; // from read

```

- The **SetTransactorWeightings** function registers the weightings matrix. You can change the weightings at any time during the simulation.

```
SetTransactorWeightings( weightings );
```

- The **RunRandomTransactor** function randomly creates a transaction call and hands it to the Transaction Manager to put on the transaction queue. Each time a transaction is created with this function the transaction arguments are randomly generated using the constraint settings for the variables. Usually this function is used inside of a loop for example:

```

for (int i=0; i < 10; i++)
{
RunRandomTransactor();
}

```



# Appendix A: Editor Commands

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

## Enabling XEmacs Integration

To enable VeriLogger's XEmacs Integration feature:

- Install XEmacs onto the computer that is running VeriLogger.
  - 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.

Information about using breakpoints with TestBench's Verilog simulator can be found in *Section 4.5* of the VeriLogger Pro manual.

## TestBench Editor Commands

Listed below are the keyboard and mouse commands supported by the editor window.

<u>Key</u>	<u>Purpose</u>
Left/right arrow	Moves the cursor one space left or right
Up/down arrow	Moves the cursor one line up or down
Page Up	Moves the cursor one page up
Page Down	Moves the cursor one page down
Home	Move to the beginning of the current line
End	Move to the end of the current line
Backspace	Deletes the character to the left of the cursor OR deletes the selected text
Delete	Deletes the character to the right of the cursor OR deletes the selected text

Shift+Leftt	Selects text one character at a time to the left
Shift+Right	Selects text one character at a time to the right
Shift+Down	Selects one line of text down
Shift+Up	Selects one line of text up
Shift+End	Selects text to the end of the line
Shift+Home	Selects text to the beginning of the line
Shift+Page Down	Selects text down one window 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 this help file
F4	Print from window
Shift+F4	Print options
Tab	Inserts Tab
Ctrl+F	Search and/or Replace Dialog
Ctrl+X	Cut
Ctrl+C	Copy
Ctrl+V	Paste
Ctrl+Z	Undo
Ctrl+Y	Redo
Ctrl+F	Search
Ctrl+G	Jump to line#

In addition to the *Editor* windows, TestBencher's *Report* window displays are full-featured editor windows.





## Appendix B: Supported Simulators

TestBencher supports a number of different simulators, depending on whether TestBencher is running under Windows or UNIX. For more information on this feature, see *Section 10.2 Simulator/Compiler Settings Dialog*.

The following simulators and languages are supported under Windows:

**Verilog-XL:** Verilog and TestBuilder/Verilog.

**VeriLogger command line:** Verilog and TestBuilder/Verilog.

**ModelSim command line:** Verilog, VHDL, and TestBuilder/Verilog. (TestBencher cannot display simulation E is used.)

**ModelSim GUI:** Verilog, VHDL, and TestBuilder/Verilog.

**NC Verilog:** Verilog and TestBuilder/Verilog.

**NC VHDL:** VHDL

**Active-HDL:** Verilog, VHDL, TestBuilder/Verilog and TestBuilder/VHDL.

The following simulators and languages are supported under UNIX:

**Verilog-XL:** Verilog and TestBuilder/Verilog.

**VeriLogger command line:** Verilog and TestBuilder/Verilog.

**ModelSim command line:** Verilog, VHDL, and TestBuilder/Verilog.

**ModelSim GUI:** Verilog, VHDL and TestBuilder/Verilog.

**VCS:** Verilog and TestBuilder/Verilog.

**NC Verilog:** Verilog and TestBuilder/Verilog.

**NC VHDL:** VHDL.

If TestBencher is running under Windows, VCS must be run on a different computer (running UNIX); Verilog-XL, NC Verilog, and NC VHDL can be run on the same computer as TestBencher or on a different computer (running UNIX). ModelSim (both versions) and the command-line version of VeriLogger can only be run on the computer running TestBencher.

If TestBencher is running under UNIX, all integrated simulators must be run on the computer that is running TestBencher.



## Appendix C: Language Independent Operators

SynaptiCAD uses a standard Boolean equation format. These equations can be use in a Signal's Boolean Equation edit box and a waveform state's Virtual state value.

The table below lists the operators that are supported by TestBencher. The operators are grouped and separated by gray rows. Each group contains the operators that are of equal precedence. The groups are listed in order of precedence from highest to lowest. The Syncad column depicts language independent operator representations that can be used for any generation language for which TestBencher can generate an appropriate expression. Where the Syncad column displays **(any)**, any operator that is valid for a generation language can be used.

Operator Name	Equations	States	Autogen	Syncad	Example	Definition
Parentheses	yes	yes	yes	()	(a + b)	Groups expressions
Bit Concatenation	yes	yes	yes	{ }	{a,b,...,z}	Concatenates operands sequentially
Bit Replication	yes	yes	yes	{{}}	{n{a}}	Concatenates a n times
Selection	yes	yes	yes	[ ]	a[i]	Selects a piece of the operand
Hierarchy	yes	yes	yes	.	a.b	Hierarchially accesses scope
List Slice	no	yes	no	..	a[0..5]	See e Language Reference manual
List Concatenation	no	yes	no	{ ; }	{a;b;c}	See e Language Reference manual
Bit Slice	yes	yes	yes	: to downto	a[3:0]	Used with selection operator to select a rang of bits
Class	no	yes	no	@	@class1	Specifies that the name following @ is an instance of a user defined class <sup>a</sup>
State Variable	no	yes	no	\$\$	\$\$addr	Specified that the name following \$\$ is a State Variable <sup>b</sup>
Macro Definition	no	yes	no	'	'base_address	Refers to a defined macro
Logical Negation	yes	yes	yes	not	!a	Converts a nonzero operand into 0 and a zero operand into 1
Bitwise Negation	yes	yes	yes	not	~a	Performs bit-wise negation of operand

Operator Name	Equations	States	Autogen	Syncad	Example	Definition
Absolute Value	yes	yes	yes	abs( )	abs(a)	Negates operand if it is negative, otherwise operand is not changed
Exponentiation	yes	yes	yes	**	a**2	See VHDL Language Reference manual
Identity	yes	yes	yes	+	+a	Leaves operand unchanged
Negation	yes	yes	yes	-	-a	Negates operand
Multiply	yes	yes	yes	*	a * b	Multiplies left and right operands
Divide	yes	yes	yes	/	a / b	Divides left and right operands
Modulo	yes	yes	yes	mod	a mod b	See VHDL Language Reference manual
Remainder	yes	yes	yes	rem	a % b	Performs division operation and returns remainder
Addition	yes	yes	yes	+	a + b	Adds left and right operands
Subtraction	yes	yes	yes	-	a - b	Subtracts right operand from left operand
Shift Left Logical	yes	yes	yes	<< (any)	a << 5	Performs left shift of left operand by the number of bit positions specified by right operand
Shift Right Logical	yes	yes	yes	>> (any)	a >> 5	Performs right shift of left operand by the number of bit positions specified by right operand
Shift Left Arithmetic	yes	yes	yes	sla	a sla 5	See VHDL Language Reference manual
Shift Right Arithmetic	yes	yes	yes	sra	a sra 5	See VHDL Language Reference manual
Rotate Left	yes	yes	yes	rol	a rol 5	See VHDL Language Reference manual
Rotate Right	yes	yes	yes	ror	a ror 5	See VHDL Language Reference manual

Operator Name	Equations	States	Autogen	Syncad	Example	Definition
Less Than	yes	yes	yes	<	$a < b$	Returns TRUE if the first expression is smaller than the second expression
Less Than or Equal	yes	yes	yes	<=	$a <= b$	Returns TRUE if the second expression is not smaller than the first expression
Greater Than	yes	yes	yes	>	$a > b$	Returns TRUE if the second expression is greater than the first expression
Greater Than or Equal	yes	yes	yes	>=	$a >= b$	Returns TRUE is the first expression is not smaller than the second expression
Positive Subtype Identification	no	yes	no	is a	$a \text{ is } a$	See e Language Reference manual
Negative Subtype Identification	no	yes	no	is not a	$a \text{ is not } a$	See e Language Reference manual
Equality	yes	yes	yes	==	$a == b$	Results 1 if operands are the same, otherwise 0 <sup>c</sup>
Inequality	yes	yes	yes	!= (any)	$a != b$	Results in 1 if operands are different, otherwise 0 <sup>c</sup>
Case Equality	yes	yes	yes	=== (any)	$a === b$	Results in 1 if operands are same, otherwise 0 <sup>c</sup>
Case Inequality	yes	yes	yes	!== (any)	$a !== b$	Results in 1 if operands are different, otherwise 0 <sup>c</sup>
Wild Equality	no	yes	yes	=?=	$a =?= b$	Results in 1 if operands are same, otherwise 0 (x and y are wildcards) <sup>c</sup>
Wild Inequality	no	yes	yes	!?=	$a !?= b$	Results in 1 if operands are different, otherwise 0 (x and y are wildcards) <sup>c</sup>
In Range List	no	yes	no	in	$a \text{ in } b$	See e Language Reference manual
Positive String Matching	no	yes	no	~	$a \sim b$	See e Language Reference manual
Negative String Matching	no	yes	no	!~	$a !\sim b$	See e Language Reference manual

Operator Name	Equations	States	Autogen	SynCAD	Example	Definition
Bitwise AND	yes	yes	yes	&	a & b	Performs bit-wise AND of two operands
Reduction AND	yes	yes	yes	&	&a	Performs bit-wise AND of one operand and results in a single bit
Reduction NAND	yes	yes	yes	~&	~&a	Performs bit-wise NAND of one operand and results in a single bit
Bitwise XOR	yes	yes	yes	^	a ^ b	Performs XOR of two operands
Reduction XOR	yes	yes	yes	^	^a	Performs bit-wise XOR of one operand and results in a single bit
Bitwise XNOR	yes	yes	yes	^~or~^	a ^~ b	Performs bit-wise XNOR of two operands
Reduction XNOR	yes	yes	yes	^~or~^	^~a	Performs bit-wise XNOR of one operand and results in a single bit
Bitwise OR	yes	yes	yes		a   b	Performs bit-wise OR of two operands
Reduction OR	yes	yes	yes		a	Performs bit-wise OR of one operand and results in a single bit
Bitwise NOR	yes	yes	yes	nor (any)	a  ~b	Performs bit-wise NOR of two operands
Reduction NOR	yes	yes	yes	~	~ a	Performs bit-wise NOR of one operand and results in a single bit
Bitwise NAND	yes	yes	yes	nand	a nand b	Performs bit-wise NAND of two operands
VHDL Bitwise XNOR	yes	yes	yes	xnor	a xnor b	Performs bit-wise XNOR of two operands <sup>d</sup>
VHDL Bitwise XOR	yes	yes	yes	xor	a xor b	Performs bit-wise XOR of two operands <sup>d</sup>
VHDL Bitwise OR	yes	yes	yes	or	a or b	Performs bit-wise OR of two operands <sup>d</sup>

Operator Name	Equations	States	Autogen	Syncad	Example	Definition
VHDL Bitwise AND	yes	yes	yes	and	a and b	Performs bit-wise AND of two operands <sup>d</sup>
Logical AND	yes	yes	yes	&&	a && b	If both operands are non-zero then result is 1, otherwise result is 0
Logical OR	yes	yes	yes		a    b	If either operand is non-zero then result is 1, otherwise result is 0
Boolean Implication	yes	yes	yes	=>	a => b	If both operands are true or first operand is false, then result is 1, otherwise result is 0
Boolean Event Check	no	yes	no	now @	now @sys.a	See e Language Reference manual.
Ternary	yes	yes	yes	any	(condition)?a:b	If condition is TRUE then result is a, otherwise result is b
Delay	yes	no	no	delay	SIG0 delay 5	Delays left operand by time specified by right operand

a. See *Chapter 8: Classes and Variables* for more information on User Defined Classes.

b. See *Section 3.3: Transaction Variables* for more information on State Variables.

c. Treatment of a and y values depend on language. Refer to the appropriate language reference manual for details.

d. These operators are to be used when the VHDL precedence is needed.

### Language Specific Operators

The operators used in building these equations can use either be the standard operators that are provided in the generation language, or they can use the language independent operators. In some cases, languages do not have an operator to represent the operation described. TestBencher generates an expression that will replicate operations not represented by the generation language when possible. The table below lists the operator representations for each language, as well as the language independent operators (shown in the Syncad column). In the language specific columns in the table below, cells that have **generates:** represent occurrences of TestBencher generating an appropriate expression for the given operator.

Operator Name	Syncad	Verilog	VHDL	e	OpenVera	TestBuilder
Parentheses	()	()	()	()	()	()
Bit Concatenation	{ }	{ }	&	%{ }	{ }	n/a

Operator Name	Syncad	Verilog	VHDL	e	OpenVera	TestBuilder
Bit Replication	{{}}	{{}}	n/a	n/a	n/a	n/a
Selection	[]	[]	[]	[]	[]	()
Hierarchy	.	.	n/a	. ~//	.	.
List Slice	..	n/a	n/a	..	n/a	n/a
List Concatenation	{;}	n/a	n/a	{;}	n/a	n/a
Bit Slice	: to downto	:	to downto	:	:	,
Class	@	n/a	n/a	n/a	n/a	n/a
State Variable	\$\$	n/a	n/a	n/a	n/a	n/a
Macro Definition	‘	‘	n/a	‘	n/a	n/a
Logical Negation	! (any)	!	not	!	!	!
Bitwise Negation	~ (any)	~	not	~	~	~
Absolute Value	abs()	n/a	abs()	n/a	n/a	abs()
Exponentiation	**	n/a	**	n/a	n/a	n/a
Identity	+	+	+	+	+	+
Negation	-	-	-	-	-	-
Multiply	*	*	*	*	*	*
Divide	/	/	/	/	/	/
Modulo	mod	n/a	mod	n/a	n/a	n/a
Remainder	% (any)	%	rem	%	%	%
Addition	+	+	+	+	+	+
Subtraction	-	-	-	-	-	-



Operator Name	Syncad	Verilog	VHDL	e	OpenVera	TestBuilder
Shift Left Logical	<< (any)	<<	sll	<<	<<	<<
Shift Right Logical	>> (any)	>>	srl	>>	>>	>>
Shift Left Arithmetic	sla	n/a	sla	n/a	n/a	n/a
Shift Right Arithmetic	sra	n/a	sra	n/a	n/a	n/a
Rotate Left	rol	n/a	rol	n/a	n/a	n/a
Rotate Right	ror	n/a	ror	n/a	n/a	n/a
Less Than	<	<	<	<	<	<
Less Than or Equal	<=	<=	<=	<=	<=	<=
Greater Than	>	>	>	>	>	>
Greater Than or Equal	>=	>=	>=	>=	>=	>=
Positive Subtype Identification	is a	n/a	n/a	is a	n/a	n/a
Negative Subtype Identification	is not a	n/a	n/a	is not a	n/a	n/a
Equality	==	==	=	==	==	==
Inequality	!= (any)	!=	/=	!=	!=	!=
Case Equality	===	===	n/a	===	===	n/a
Case Inequality	!==	!==	n/a	!==	!==	n/a
Wild Equality	=?=	n/a	n/a	n/a	=?=	n/a
Wild Inequality	!?=	n/a	n/a	n/a	!?=	n/a
In Range List	in	n/a	n/a	in	n/a	n/a
Positive String Matching	~	n/a	n/a	~	n/a	n/a
Negative String Matching	!~	n/a	n/a	!~	n/a	n/a
Bitwise AND	&	&	n/a	&	&	&
Reduction AND	&	&	n/a	n/a	&	.reductionAnd()

Operator Name	Syncad	Verilog	VHDL	e	OpenVera	TestBuilder
Reduction NAND	$\sim\&$	$\sim\&$	n/a	n/a	$\sim\&$	.reductionNand()
Bitwise XOR	$\wedge$	$\wedge$	n/a	$\wedge$	$\wedge$	$\wedge$
Reduction XOR	$\wedge$	$\wedge$	n/a	n/a	$\wedge$	.reductionXor()
Bitwise XNOR	$\wedge\sim\wedge$	$\wedge\sim\wedge$	n/a	generates: $\sim(x\wedge y)$	$\wedge\sim\wedge$	generates: $\sim(x\wedge y)$
Reduction XNOR	$\wedge\sim\wedge$	$\wedge\sim\wedge$	n/a	n/a	$\wedge\sim\wedge$	.reductionXnor()
Bitwise OR	$ $	$ $	n/a	$ $	$ $	$ $
Reduction OR	$ $	$ $	n/a	n/a	$ $	.reductionOr()
Bitwise NOR	nor (any)	generates: $\sim(x y)$	nor	generates: $\sim(x y)$	$ $ $\sim$	generates: $\sim(x y)$
Reduction NOR	$\sim $	$\sim $	n/a	n/a	$\sim $	.reductionNor()
Bitwise NAND	nand	generates: $\sim(x\&y)$	nand	generates: $\sim(x\&y)$	$\&$ $\sim$	generates: $\sim(x\&y)$
VHDL Bitwise XNOR	xnor	generates: $x\sim y$	xnor	n/a	n/a	n/a
VHDL Bitwise XOR	xor	generates: $x\wedge y$	xor	n/a	n/a	n/a
VHDL Bitwise OR	or	generates: $x y$	or	n/a	n/a	n/a
VHDL Bitwise AND	and	generates: $x\&y$	and	n/a	n/a	n/a
Logical AND	$\&\&$	$\&\&$	n/a	$\&\&$	$\&\&$	$\&\&$
Logical OR	$\ \ \ $	$\ \ \ $	n/a	$\ \ \ $	$\ \ \ $	$\ \ \ $
Boolean Implication	$\Rightarrow$	generates: $(!x)\ \ y$	generates: (not x) or y	$\Rightarrow$	generates: $(!x)\ \ y$	generates: $(!x)\ \ y$
Boolean Event Check	now @	n/a	n/a	now@	n/a	n/a
Ternary	? : (any)	? :	when else	? :	? :	? :

<b>Operator Name</b>	<b>Syncad</b>	<b>Verilog</b>	<b>VHDL</b>	<b>e</b>	<b>OpenVera</b>	<b>TestBuilder</b>
Delay	delay	#	transport after	n/a	n/a	n/a



# Appendix D: License Agreement

SynaptiCAD

TestBench Pro - DataSheet Pro - WaveFormer Pro - WaveFormer Lite -  
Timing Diagrammer Pro - VeriLogger Pro - BugHunter Pro - GigaWave Viewer  
Software License Agreement

- Read Before Use -

Please read and understand this license.

Note: Throughout this agreement, the word Software refers to the software product that you have licensed from SynaptiCAD.

You have purchased a license to use one of the following products: **TestBench Pro, DataSheet Pro, WaveFormer Pro, WaveFormer Lite, VeriLogger Pro, BugHunter Pro, GigaWave Viewer, or Timing Diagrammer Pro** software. The software is owned and remains the property of SynaptiCAD, is protected by international copyrights, and is transferred to the original purchaser and any subsequent owner of the Software media for their use only on the license terms set forth below. Opening the packaging for **TestBench Pro, DataSheet Pro, WaveFormer Pro, WaveFormer Lite, VeriLogger Pro, BugHunter Pro, GigaWave Viewer** or **Timing Diagrammer Pro** and/or using either **TestBench Pro, DataSheet Pro, WaveFormer Pro, WaveFormer Lite, VeriLogger Pro, BugHunter Pro, GigaWave Viewer** or **Timing Diagrammer Pro** indicates your acceptance of these terms. If you do not agree to all of the terms and conditions, return the unopened Software and manuals immediately for a full refund.

## Use of the Software

- **SynaptiCAD** grants the original purchaser ("Licensee") the limited rights to possess and use the Software and User's Manual ("Software") for its intended purpose. Licensee agrees that the Software will be used solely for Licensee's internal purposes and that the Software will be installed on a single computer only. If the Software is installed on a networked system, or on a computer connected to a file server or other system that physically allows shared access to the Software, Licensee agrees to provide technical or procedural methods to prevent use of the Software by more than one user.
- One machine-readable copy of the Software may be made for BACKUP PURPOSES ONLY, and the copy shall display all proprietary notices, and be labeled externally to show that the backup copy is the property of SynaptiCAD, and that use is subject to this License.
- Use of the Software by any department, agency or other entity of the U.S. Federal Government is limited by the terms of the below "Rider for Governmental Entity Users."
- Licensee may transfer its rights under this License, provided that the party to whom such rights are transferred agrees to the terms and conditions of this License, and written notice is provided to SynaptiCAD. Upon such transfer, Licensee must transfer or destroy all copies of the Software.
- Except as expressly provided in this License, Licensee may not modify, reverse engineer, decompile, disassemble, distribute, sub-license, sell, rent, lease, give or in any other way transfer, by any means or in any medium, including telecommunications, the Software. Licensee will use its best efforts and take all reasonable steps to protect the Software from unauthorized use, copying or dissemination, and will maintain all proprietary notices intact.

**LIMITED WARRANTY** SynaptiCAD warrants the Software media to be free of defects in workmanship for a period of ninety days from the purchase. During this period, SynaptiCAD will replace at no cost any such media returned to SynaptiCAD, postage prepaid. This service is SynaptiCAD's sole liability under this warranty.

**DISCLAIMER** LICENSE FEES FOR THE SOFTWARE DO NOT INCLUDE ANY CONSIDERATION FOR ASSUMPTION OF RISK BY SYNAPTICAD, AND SYNAPTICAD DISCLAIMS ANY AND ALL LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR OP-

ERATION OR INABILITY TO USE THE SOFTWARE, EVEN IF ANY OF THESE PARTIES HAVE BEEN ADVISED OF THE POSSIBILITIES OF SUCH DAMAGES. FURTHERMORE, LICENSEE INDEMNIFIES AND AGREES TO HOLD SYNAPTICAD HARMLESS FROM SUCH CLAIMS. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY THE LICENSEE. THE WARRANTIES EXPRESSED IN THIS LICENSE ARE THE ONLY WARRANTIES MADE BY SYNAPTICAD AND ARE IN LIEU OF ALL OTHER WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND OF FITNESS FOR A PARTICULAR PURPOSE. THIS WARRANTY GIVES YOU SPECIFIED LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM JURISDICTION TO JURISDICTION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF WARRANTIES, SO THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

#### Term

- This license is effective as of the time Licensee receives the Software, and shall continue in effect until Licensee ceases all use of the Software and returns or destroys all copies thereof, or until automatically terminated upon failure of Licensee to comply with any of the terms of this License.

#### General

- This License is the complete and exclusive statement of the parties' agreement. Should any provision of this license be held to be invalid by any court of competent jurisdiction, that provision will be enforced to the extent permissible, and the remainder of the License shall nonetheless remain in full force and effect. This License shall be controlled by the laws of the State of Virginia, and the United States of America.

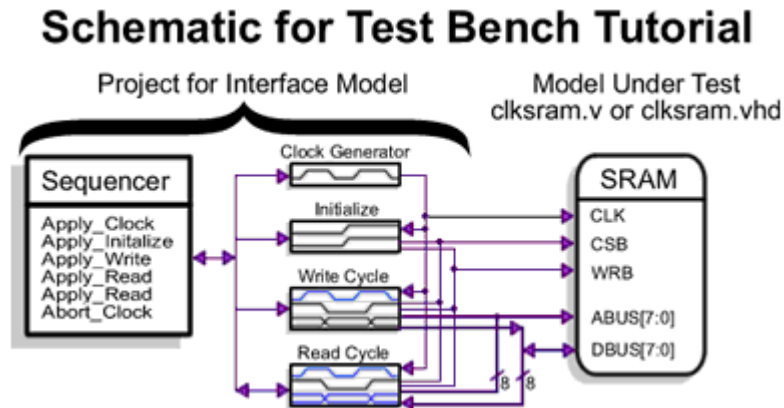
### **Rider For U.S. Governmental Entity Users**

This is a rider to **TestBench Pro/ DataSheet Pro/VeriLogger Pro/ WaveFormer Pro/ WaveFormer Lite/ BugHunter Pro/ GigaWave Viewer/ Timing Diagrammer Pro** SOFTWARE LICENSE AGREEMENT ("License") and shall take precedence over the License where a conflict occurs.

1. The Software was: developed at private expense (no portion was developed with government funds) and is a trade secret of SynaptiCAD and its licensor for all purposes of the Freedom of Information Act; is "commercial computer software" subject to limited utilization as provided in any contract between the vendor and the government entity; and in all respects is proprietary data belonging solely to SynaptiCAD and its licensor.
2. For units of the DoD, the Software is sold only with "Restricted Rights" as that term is defined in the DoD Supplement to DFAR 252.227-7013 (b)(3)(ii), and use, duplication or disclosure is subject to restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013. Manufacturer: SynaptiCAD, PO Box 10608, Blacksburg, Va 24062-0608 USA.
3. If the Software was acquired under a GSA Schedule, the Government has agreed to refrain from changing or removing any insignia or lettering from the Software or Documentation or from producing copies of manuals or disks (except for backup purposes) and: (1) Title to and ownership of the Software and Documentation and any reproductions thereof shall remain with SynaptiCAD and its licensor; (2) use of the Software shall be limited to the facility for which it is acquired; and (3) if the use of the Software is discontinued at the original installation location and the Government wishes to use it at another location, it may do so by giving prior written notice to SynaptiCAD, specifying the new location site and class of computer.
4. Government personnel using the Software, other than under the DoD contract or GSA Schedule, are hereby on notice that use of the Software is subject to restrictions that are the same or similar to those specified above.

# TestBencher Pro: Basic Tutorial

In less than 30 minutes you will create a reusable test bench that can apply different stimulus and verify the results of a clocked SRAM. Below is a schematic of the different components that you will construct. First you will create the Project file that controls the generation of the interface model (test bench). Next you will draw the different transaction diagrams that are needed to communicate with the SRAM. And then you will edit the sequencer process to apply the transactions to the model under test. Finally you will simulate the design and verify the operation of the SRAM model.



## Preparation

Before we begin there are few things to setup and understand:

1. This tutorial requires a full version license or an evaluation license. If you are evaluating then you can obtain a license by completing the form under **Help > Request License** menu item and contacting our sales department. To check that you have a good license, verify that you can save a timing diagram.
2. This tutorial assumes that you are familiar with the SynaptiCAD timing diagram editing environment. If you would like more information on the drawing environment then work through the short **Help > Tutorial > Basic Drawing and Timing Analysis** tutorial.
3. This tutorial can be used to generate VHDL, Verilog, TestBuilder, OpenVera and e code. Sometimes a file name will be written as *filename.<language extension>*. This means that the file extension will be different depending on the language used: Verilog \*.v, VHDL \*.vhd, TestBuilder \*.cpp, OpenVera \*.vr, and e code \*.e.

## 1) Create a Project

TestBencher Pro uses a project file to represent and to control the generation of a bus-functional model (BFM) component. The information in the project file is displayed in the Project window and context sensitive menus provide a list of actions that can be performed for the elements in the project tree. In this section the project will be created, the Model Under Test (MUT) file will be added to the project, and the template diagram will be constructed.

### 1.1 Use the New Project Wizard to Create a Project

Projects are created using New Project Wizard dialog. This dialog helps setup the project directory, the generated language, and the clocking signal for the project.

To create a new project:

1. Select the **Project > New Project** menu option to launch the *New Project Wizard* dialog.
2. Enter **sramtest** in the *Project Name* edit box. The actual project directory will be a subdirectory below the displayed path in the *Project Directory* edit box. This subdirectory will have the same name as the project.

**Unix users:** Make sure that you have read/write access to the directory specified in the *Project Directory* edit box.

3. From the *Project Language* dropdown, select the code generation language.
4. Check the **Transaction-based Test Bench Generation** checkbox.
5. Click the **Next** button to move to the second page of the *New Project Wizard*.
6. Note that the name of the *New Template* is **sramtest** (the name of the project). TestBench will use this file to generate the top-level module of the test bench. The *Original Template*, named **tbench**, is copied into the *New Template* file.
7. Type **CLK** into the **Default Clock** dropdown, and choose **neg** from the **Edge** dropdown box. Selecting a default clock causes the test bench to be cycle-based; if no clock is specified, the test bench will be event-based.
8. Check the **Create Default Clock Generator** box. This will cause TestBench to create a slave timing diagram called **Clk\_generator.btim** that will drive the *CLK* signal.
9. Click the **Finish** button to close the *New Project Wizard*, create the project, and populate the *Project* window.

## 1.2 Add the MUT to the Project

Next we will add the clocked SRAM model file to the project. TestBench uses the model under test files to extract the signal and port information for use in the transaction diagrams. TestBench also uses the MUT file information to instantiate it in the component model (template file).

**Note for Remote Simulators:** If your simulator or HVL tools are running on a different computer than TestBench Pro, then the external simulator integration feature requires that all files used for the project be in the project directory, or a subdirectory thereof. If you are working a remote simulator, copy the appropriate MUT file (**clksram.v** or **clksram.vhd**) from the **SynaptiCAD > Examples > TestBenchBasicTutorial** directory into the project directory prior to adding the MUT to the project.


To add the MUT to the project:

1. Right-click the *User Source Files* folder in the *Project* window and select the **Add File(s) to User Source File Folder...** from the context menu option. This will open the *Add Files...* dialog.
2. Select the file to use as the MUT from the **SynaptiCAD > Examples > TutorialFiles > TestBenchBasicTutorial** directory (or from the project directory if your simulator is on a remote machine):  
Verilog model file is **clksram.v**.  
VHDL model file is **clksram.vhd**.
3. Click the **Open** button to close the dialog and add the file to the *User Source Files* folder in the *Project* window.

## 1.3 Extract Port Information from the MUT into the Template Diagram

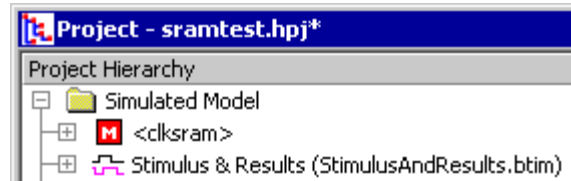
When TestBench created the project it also generated a template diagram. New transaction diagrams that are created for this project will contain the same signals, waveforms, parameters, and properties as the template diagram. Currently the CLK signal is the only signal in this diagram and we are going to add the port signals for the clocked SRAM.

To extract the ports from the SRAM into the template diagram:

1. In the *Project* window, under the *Template Diagram* folder, double click on **sramtest\_templateDiagram.btim** to open the template diagram window.
2. Click the **Extract Ports from MUT**  button. This will build the MUT and insert the signals for the MUT ports into the template diagram.



- Notice that `<clksram>` is now present in the *Project* window under the *Simulated Model* folder. The single angle brackets indicate that `clksram` is the Model Under Test. Expanding this tree will display signal, port, and component information of the MUT.




**Note:** If `<clksram>` was not generated as the MUT, then change the simulation preferences by choosing the **Options > Diagram Simulation Preferences** menu. Check the **Auto-create test bench and tree** check box. Press the **Extract Ports from MUT** button to rebuild the MUT.

### 1.4 Modify the Template Diagram

The transaction diagrams use an End Diagram Marker to indicate the exact time that the transaction ends. So we will add an end diagram marker to the template diagram, so all new transactions will get the marker.

To add an end diagram marker:

- Click on the **Marker** button  on the diagram button bar.
- Click on the fourth falling edge of the CLK signal (at 350 ns) to select it. Then right-click to draw a marker that is attached to the edge.
- Double-click on the marker to open the *Edit Time Marker* dialog.
- Select a *Marker Type* of **End Diagram** from the drop down list box. This end diagram marker will force the transaction to end at the fourth falling edge of the CLK signal.
 

Marker Type:
- Select **Type** from the *Display Label* list box. This will cause the marker to display its type rather than its name.
- Click **OK** to close the *Edit Time Marker* dialog.
- Use the **File > Save All Files** menu option to save the project and the template diagram.

The template diagram should look like the following:

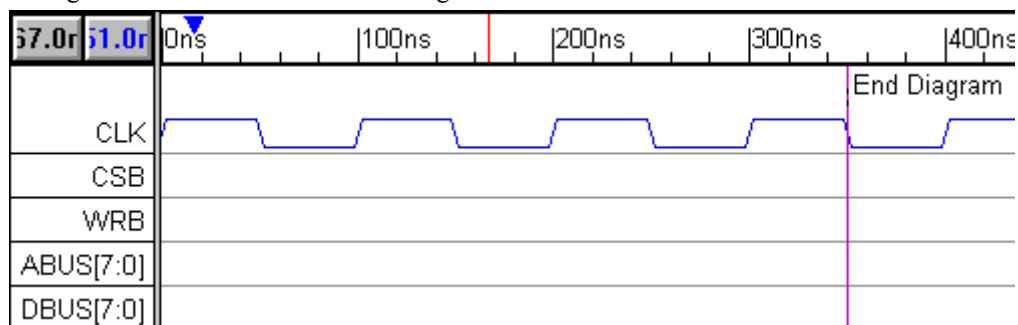


Figure 2: Completed Template Diagram

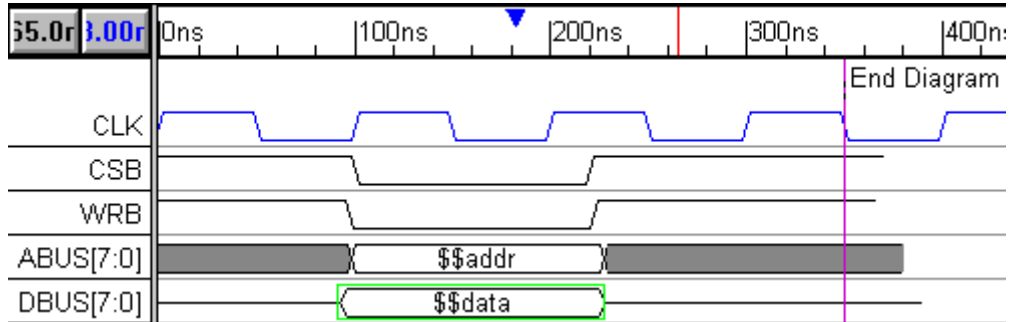
## 2) Create the Write Cycle Transaction Diagram

TestBench Pro uses timing diagrams that represent reusable bus transactions to generate the test bench. This tutorial will use two timing diagrams, `tbread.btim` and `tbwrite.btim`, to represent the read and write cycles used in testing the memory module. First, the write cycle diagram will be created. Then this diagram will be used as a basis for creating the read cycle diagram. Variables will be used in the diagrams so that values can be passed into the address and data buses.

### 2.1 Draw the Timing Diagram for the Write Cycle

This section explains how to create the timing diagram that represents the write cycle transaction.

1. In the *Project* window, right click the *Transaction Diagrams* folder and select **Create a new Master Transaction** from the context menu. This will cause the *Save As* dialog to open.
2. Name the file **tbwrite** and press the **Save** button. This will copy the template diagram to the new file, list the file in the *Transaction Diagram* folder, and open the new diagram.
3. Draw the following waveforms (the state values will be added in the next section):




**Figure 3: Completed Write Cycle Diagram**

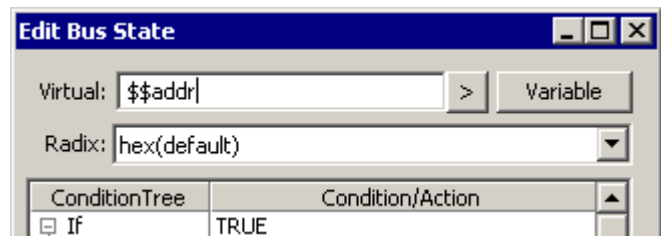
Note: If you have trouble drawing the waveforms, then refer to the **Basic Drawing and Timing Analysis** tutorial.

## 2.2 Add Parameterized State Values for Write Cycle

The next step is to add state variables to the timing diagram so that values for the address and data buses may be passed into the test bench transaction. Parameterized state values, called **state variables**, are passed into the transaction call in the top-level template file, and are used to provide state or comparison values during transaction execution. The write cycle diagram will have a state variable for the value on the address bus, and a state variable for the value on the data bus. When the top-level template file is modified, values will be passed into the state variables.

To add the address and data state variables to the diagram:

1. Double click on the valid segment in the center of *ABUS* to open the *Edit Bus State* dialog.
2. Type **\$\$addr** into the **Virtual** edit box. The "\$\$" in front of the variable name indicates that this is a state variable. If the "\$\$" is missing, TestBench Pro will assume that this is the value of the address rather than a variable that will accept a value at a later time.
3. Click on the valid segment in the center of *DBUS* to move the focus of the *Edit Bus State* dialog to the new segment.
4. Type **\$\$data** in the **Virtual** edit box.
5. Click **OK** to close the *Edit Bus State* dialog. The two edited segments will display the state variables.
6. Click the diskette icon  on the main toolbar to save the timing diagram.



## 3) Create the Read Cycle Transaction Diagram

The read cycle will initiate a read with the clocked SRAM and monitor the data bus to verify the result of the read. For the read cycle, the data bus will be an input signal (not driven like the write cycle), and the **\$\$data** variable will be used for comparison with the actual value driven by the SRAM.

### 3.1 Create Read Cycle Diagram and Add it to the Project

Since the signals for the read diagram are so similar to the write diagram, a modified copy of **tbwrite.btim** can be used to create the read diagram.

Create the read cycle timing diagram and add it to the project:

1. In the **tbwrite** diagram window, right-click in the Label window and select the **Save As...** menu option to open the *Save As* dialog.
2. Name the file **tbread**, and press the **Save** button. This will create a new file, but you still will need to add the file to the project.
3. Right-click in **tbread**'s Label window, and select **Add Master Diagram to Project** from the context menu. This will add **tbread** to the *Transaction Diagrams* folder in the *Project* window.

### 3.2 Edit the Waveforms for Read Cycle

The WRB and DBUS signals need to be changed for the Read cycle. The write control signal, WRB, should stay high (inactive) for the duration of the read. And during the read the DBUS signal will be driven by the SRAM, so the data segment of the signal needs to be set to input. Also since our SRAM is clocked the data comes out on the clock cycle after the chip select signal, CSB, goes active.

To edit the waveforms:

1. Make the WRB signal high for the entire read cycle. Select the center segment and press the delete key to remove the low signal segment.
2. Shift the start of the DBUS data segment to 200ns. Hold down the <2> key (the number 2 key) on the keyboard, while dragging the starting transition to 200ns. The <2> key causes transitions to the right of the selected edge to move with the dragged edge.
3. Set the DBUS data segment to be a blue input segment. Double click on the data segment to open the *Edit Bus State* dialog, uncheck **Driven (Export to source code)** checkbox.

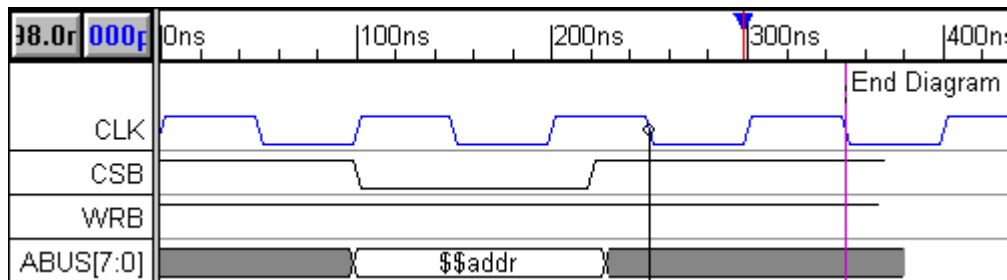



Figure 4: Completed Read Cycle Diagram


### 3.3 Add a Sample to Verify Data

Next a Sample will be added to the timing diagram. Samples compare the actual state value of an input signal to the expected state value, and conditionally react to the results of the comparison.

To add a Sample:

1. Click on the **Sample** button  on the button bar.
2. Click on the **third falling edge** (250ns) of **CLK** to select the edge.
3. Right-click near the end of the **blue valid segment** on **DBUS**. This adds a Sample parameter named **SAMPLE0** that lines up with the third neg edge of the **CLK** signal. Refer the image in the previous section.

The default behavior of the sample compares the run time value with the drawn value (\$\$data) and throws an *Error* if they are different. This is the behavior that we need for the tutorial. The next few steps show you the HDL code generation dialog and how to control the generated code. You do not need to make any changes to the dialog defaults.

1. Double-click on the sample name **SAMPLE0** in the drawing window to open the *Sample Properties* dialog.
2. Press the **HDL Code** button in the dialog to open the *Code Generation Options* dialog. 
3. In the *If Condition* dropdown, select **Sample state matches**. This means that during simulation, the test bench will compare the actual value on the data bus with the value passed into the timing diagram (\$\$data).
4. In the *Then Action* dropdown, select **Do nothing**. If the value on the data bus matches the value of \$\$data, then the circuit is working properly and no action should be taken.
5. In the *Else Action* dropdown, select **Display Message**. This means that if the values don't match, a message will be displayed during the simulation.
6. Below the *Else Action* dropdown, choose the **Error** radio button. These radio buttons allow a severity level to be defined for the message that is displayed.
7. Click **OK** to close the *Code Generation Options* dialog.
8. Click **OK** to close the *Sample Properties* dialog.
9. Save the timing diagram by selecting **File > Save Timing Diagram** from the main TestBench menu.

## 4) Create the Initialize Transaction Diagram

When drawing the waveforms for a transaction diagram it is important to remember that transactions do not automatically include an event at time zero and that only the drawn events are driven. This is a feature that allows transactions to be reused any time during simulation without implying any initialization information. In our example the clocked SRAM control signals, CSB and WRB, need to be initialized before the read and write cycles are applied to the model. We will draw a simple initialization diagram that will drive the control signals to high (inactive).

### 4.1 Draw the Initialization Waveforms

Create the Initialization diagram by first copying the template diagram, removing the extra signals, and drawing the waveforms.

1. In the *Project* window, right click the *Transaction Diagrams* folder and select the **Create a new Master Transaction** from the context menu. This will cause the *Save As* dialog to open.

2. Name the file **tbinitialize** and press the **Save** button. This will save the diagram, add it to the *Transaction Diagram* folder, and open the new diagram.
3. Remove the **ABUS** and **DBUS** signals, because the tri-state bus signals do not need to be initialized. Select the **ABUS** and **DBUS** signals by clicking on them, and then press the **<delete>** key to delete the selected signals.
4. Draw the following waveforms:

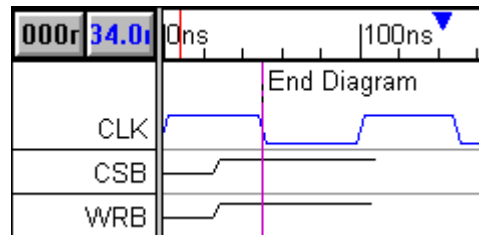



Figure 5: Completed Initialization Diagram

#### 4.2 Move the End Diagram Marker for Initialization Diagram

The initialization timing diagram will only need one clock cycle to initialize the control signals. Therefore, the **End Diagram** marker can be moved to the 1st negative clock edge.

To move the **End Diagram** marker:

1. Double-click on the marker to open the *Edit Time Marker* dialog.
2. Select **Attach to Edge** from the radio buttons.
3. Click **OK** to close the *Edit Time Marker* dialog. This will put TestBencher into a special select mode.
4. Click on the first negative clock edge (at 50ns) to attach the marker to that edge.
5. Click the diskette icon  on the main toolbar to save the timing diagram.

### 5) Modify the Sequencer Process

Inside the primary template file for the project is a *Sequencer Process*. The Sequencer Process is the place in the top-level test bench that defines the order in which the timing transactions are applied to the model under test.

The *Insert Diagram Subroutine Calls* dialog generates diagram apply calls so you do not need to memorize the function syntax. Each timing diagram generates three task calls: Apply, Apply-nowait, and Abort. Apply runs the transaction in a blocking mode, and Apply-nowait runs the transaction concurrently with other transactions. The *Master/Slave Diagram Setting* determines how many times a transaction executes. Master Transactors, like the Read, Write, and Initialize diagrams run once and stop. Slave Transactors like the Global Clock Generator run in a looping mode until an Abort call is received.

In addition to these task calls, you can also place HDL code in the sequencer. One example where this would be useful is if you wish to place conditions on whether or not a timing transaction is called, or on the parameter values that you wish to have applied.

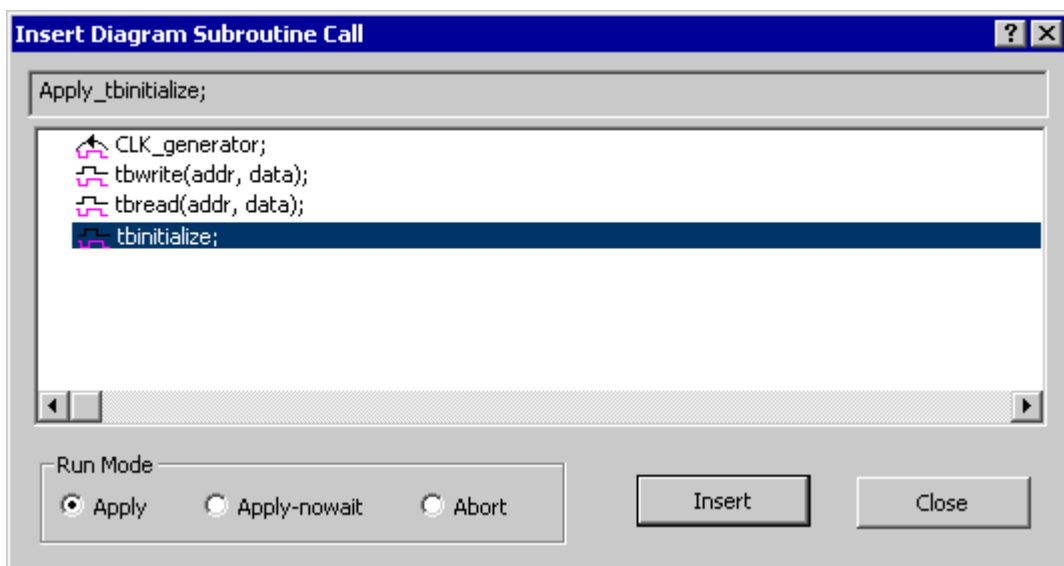
An alternative method to placing transaction calls in the sequencer process is to create a file external to the bus-functional model with transaction calls and during simulation read the transaction calls from a file (see *Section 9.4: Transaction Manager and Test Reader* in the online TestBencher Manual).

#### 5.1 Adding Apply Calls to the Sequencer Process

Use the *Insert Diagram Subroutine Calls* dialog to add apply statements to the Sequencer. We will first start the clock, initialize the control signals, write to the SRAM, the read from the SRAM twice, and then abort the clock.

To edit the sequencer process:

1. In the *Project* window, double click on the *Component Model* folder to open an editor window with the **sramtest** template file.
2. Scroll down in the **sramtest** editor window near the end of the file until you find the comment block that has this line:  
  
Transaction Sequencer - After this comment, define how to apply transactions to the model under test using:
3. Click in the **sramtest** editor window below this comment so that the blinking cursor is in the place where the apply statement should be added.
4. Right-click in the editor window and select **Insert Diagram Calls** to open the *Insert Diagram Subroutine Call* dialog.



5. Arrange the windows so you can see the editor and the dialog at the same time.

Use the *Insert Diagram Subroutine Calls* dialog to add the apply calls. When you select a slave diagram, the dialog will automatically default to **Apply-nowait**, because most of the time slave diagrams will run concurrently with other diagrams. When you select a master diagram, the dialog will automatically default to **Apply**, because most of the time master diagrams run in a blocking mode:

1. Double click on the **CLK\_generator** entry in the *Insert Diagram Subroutine Calls* dialog. This adds an apply call to the editor window.
2. Double click on the **tbinitialize** entry.
3. Double click on the **tbwrite** entry.
4. Double click on the **tbread** entry TWO times to insert the code to add two read calls.
5. Select **CLK\_generator** entry, choose **Abort** radio button, and then press the **Insert** button to insert the code. This will add the abort call to stop the clock generator.
6. The apply call should look similar to the following code block. Different languages may have extra parameters.

```
//*****
// Transaction Sequencer - After this comment, define how to
// apply transactions to the model under test using:
```

```

//
// - Transaction calls (Insert Diagram Calls in right-click menu)
// - Source code in Verilog
//*****
Apply_CLK_generator_looping_nowait;
Apply_tbinitalize;
// Apply_tfwrite(addr, data);
Apply_tfwrite(addr, data);
// Apply_tbread(addr, data);
Apply_tbread(addr, data);
// Apply_tbread(addr, data);
Apply_tbread(addr, data);
Abort_CLK_generator;

```

## 5.2 Providing Values for Variables in Timing Transactions

The **tfwrite** and **tbread** transactions have parameterized state values. These values are passed to the transaction in the Apply statements.

To set the values of the state variables in the transaction apply calls:

1. Edit the write and read Apply code lines and replace the state variable names with actual variables that will be passed into the timing diagrams. The comment lines are there to document the parameter variable names.  
**Note:** The code to be entered is **bold**.

For Verilog type:

```

Apply_tfwrite('hF0, 'hAE);
Apply_tbread('hF0, 'hAE);
Apply_tbread('hF0, 'hEE);

```

For VHDL type:

```

Apply_tfwrite(tb_Control, tb_InstancePath, x"F0", x"AE");
Apply_tbread(tb_Control, tb_InstancePath, x"F0", x"AE");
Apply_tbread(tb_Control, tb_InstancePath, x"F0", x"EE");

```

For OpenVera type:

```

tb_tfwrite.ExecuteOnce('hF0, 'hAE);
tb_tbread.ExecuteOnce('hF0, 'hAE);
tb_tbread.ExecuteOnce('hF0, 'hEE);

```

2. Save the top-level template file by right-clicking in the editor window and selecting **Save**.

Notice that the **tfwrite** apply statement writes the hex value AE to memory cell F0. The **tbread** diagram calls will then read the value from the same memory cell. The data values provided in the **tbread** diagram calls will be used to compare with the actual value. The first call to **tbread** will expect to find a value of hex AE in the address F0. The second call to **tbread** will expect to find the hex value EE instead. This will cause the sample to report an error during the second execution of **tbread**.

## 6) Generate Test Bench and Simulate

At this point all the timing diagrams have been created and you have edited the Sequencer process. Next we will generate the test bench and simulate the entire design.

### 6.1 Setup the Simulator

TestBench can control external simulators and compilers or use its built-in Verilog to compile and simulate the design. If you are using the built-in simulator, skip ahead to next section. *Section 10.3: External Program Integration* in the online manual has a complete list of instructions for working with remote simulators and for setting up a compiler for TestBuilder.

To configure a third-party simulator:

1. Choose the **Options > Simulator and Compiler Settings** menu option. This will open the *Simulator and Compiler Settings* dialog.
2. From the **Simulator and Compiler tools** dropdown select the appropriate simulator.
3. Enter the directory that contains the simulator executable in the **Simulator Path** edit box.
4. Click **OK** to close the *Simulator and Compiler Settings* dialog.


Select the third-party simulator:

1. Select the **Project > Project Settings** menu option. This will open the *Project Settings* dialog.
2. Select the tab for the language you are working with.
3. Select the desired simulator from the **Simulator Type** dropdown.
4. Click **OK** to close the *Project Settings* dialog.



### 6.2 Generate the Test Bench and Simulate

Once the simulator is setup you are ready to generate the test bench and simulate the design.

To generate the test bench:

1. Click on the **Make TB** button  on the main TestBench toolbar. This will expand the macros in the template file and pop up a dialog that says "*Finished generating test bench. Please check waveperl.log for errors.*" Close this dialog by clicking the **OK** button.
2. In the *Report* window, check the **waveperl.log** tab to see if TestBench encountered any errors during the test bench generation. If it did, fix the error and regenerate the test bench. (If you can not see the *Report* window, choose the **Window > Report** menu to bring it to the front.)

To simulate the design:

1. Click the yellow **Compile Model and Test Bench**  button. This builds (parses) the project using the tools specified in the *Project Settings* and *Simulator and Compiler Settings* dialogs.  
 In the bottom right corner, a yellow **Simulation Built** status message indicates the build was successful and that you are ready to simulate.  
 If the status indicates an error, the *Report* window *Errors* tab displays the compile errors. If there are errors then fix them, regenerate the test bench, and recompile.
2. Click the green run button  on the simulation button bar. This will simulate the design and display the results in the *StimulusAndResults* diagram and the *Report* window *simulation.log* tab.

In the bottom right corner, a **Simulation Good** status message indicates that the simulation has reached a successful end.



### 6.3 Examine Report Window Results

The *Report* window **simulation.log** tab displays the default log file for the simulator. TestBencher automatically writes a message to the log file each time a transaction starts and stops. The clocked SRAM contains code to display a message each time it performs a read or write. We also added a sample parameter to the Read Cycle, and set it to generate an error message when the data from the SRAM does not match the expected value.

Examine the log file:

1. In the *Report* window, open the **simulation.log** tab and display the following results:

```
Running...
TB> Note: In "sramtest_CLK_generator" at 0.000ns: Executing LOOPING
TB> Note: In "sramtest_tbinitialize" at 0.000ns: Executing ONCE
TB> Note: In "sramtest_tbinitialize" at 50.000ns: Execution DONE
TB> Note: In "sramtest_tbwwrite" at 50.000ns: Executing ONCE
In clksram at 150.000ns: Writing ae to address f0
TB> Note: In "sramtest_tbwwrite" at 350.000ns: Execution DONE
TB> Note: In "sramtest_tbread" at 350.000ns: Executing ONCE
In clksram at 450.000ns: Reading ae to address f0
TB> Note: In "sramtest_tbread" at 650.000ns: Execution DONE
TB> Note: In "sramtest_tbread" at 650.000ns: Executing ONCE
In clksram at 750.000ns: Reading ae to address f0
TB> Error: In "sramtest_tbread" at 850.000ns: Sample SAMPLE0_process sampled
    signal: DBUS expected: ee ; detected: ae
TB> Note: In "sramtest_tbread" at 950.000ns: Execution DONE
TB> Note: In "sramtest_CLK_generator" at 950.000ns: Execution DONE
0 Errors, 0 Warnings
Compile time = 0.01000, Load time = 0.02000, Execution time = 0.05000
```

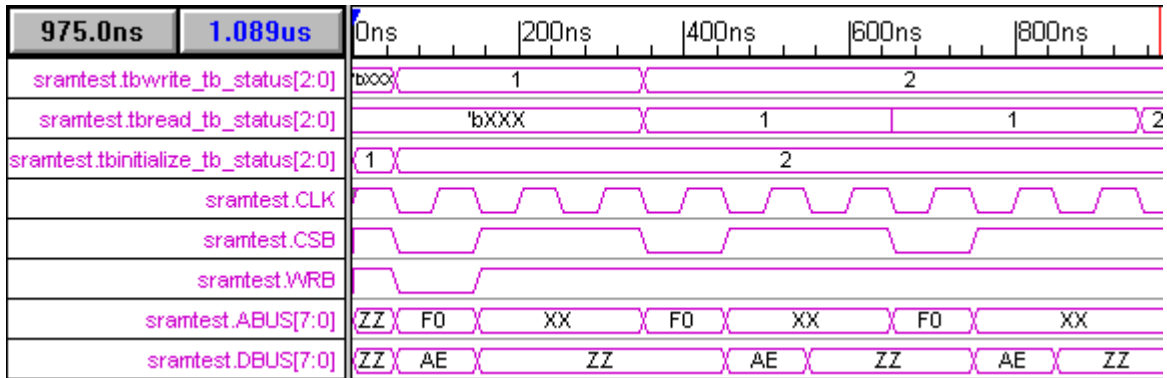
Normal exit

2. Notice that the clock generator starts at time zero and continues until the end of the simulation when the abort call is issued.
3. The initialization diagram also starts executing at time zero and blocks the next transaction until it is complete.
4. The write diagram starts next and writes a value to the SRAM. The SRAM acknowledges that is writing the value to the specified address.
5. The first read diagram executes successfully.
6. The second read diagram throws a warning because the expected value did not match the value from the MUT. We purposely passed in a bad expected data value so we could see how the sample throws the error.
7. Next the abort call to the clock stops the clock transaction and ends the simulation.

### 6.4 Examine the Stimulus and Results Diagram

After simulation the Stimulus and Results diagram will contain all of the top level signals of the project, the driver signals, and status and trigger signals for each transaction.

- Hide some of the signals in the Stimulus and Results diagram by selecting the signal names and choosing **View > Hide Selected Signals** until the diagram looks like this:



- A status signal of <1> indicates the transaction is running. You can see that the initialization diagram runs followed by the write cycle and two read cycles.
- During the write cycle, the data AE is written to address F0. When comparing the simulated write cycle to the drawn transaction, remember that this is a negative clock edge diagram.
- The read cycles read back the data from the memory.

## Index

### Symbols

\$\$ - see State Variables

@ - see Variables

### Numerics

2\_state 74

2\_state\_vector 74

4\_state 74

4\_state\_vector 74

### A

Absolute Samples 56

ActiveHDL 93

Add Timestamp to Each File 23

Advanced Register and Latch Controls 52

Applied 73

Apply Calls 43

    adding to template file 12

    constant data values 86

    in TestBuilder 113

    in VHDL 109

    parameters 86

    techniques 101

Arrays 73, 74

Associative Arrays 73, 74

Attach to Edge 63

Attach to Time 63

Auto Run 99

### B

Bi-directional Signals 42

Big Endian (Packing) 76

Bit Slice

    port mapping 24

Blocking Samples 58, 59

bool 74

Boolean Condition For Delay 51

Burst Mode Transactions 102

Buses - adding to diagram 44

byte 74

### C

C++ 93

Class Instances - see Variables

Class Libraries 69, 70

    creating 69, 70

    Default 69

Class Methods

    adding to class 72

    calling 30

    Component Model 22, 23

    creating 77, 78

    Diagram-Level 30

    editing 77, 78

    parameters 78

    source code 79

Classes 23, 69

    applied 73

    Class Methods 77, 78

    constraints 79, 80

        expressions in 80

    creating 71

    defining fields 71, 82

    editing 71

    field properties 73

    libraries

        see Class Libraries

    packing 73, 76

    static fields 73

    structure 74

Classes and Variables Dialog 29, 71, 72

Clocking Domain

    default 34

Clocks

    adding to diagram 44

    default 18

Code Generation

    controlling 11, 34, 56

    Enable Abort Code 38

    inserting HDL code 66, 67

    samples 56

Code Generation Options Dialog 56, 61

Command File - Generating 99

Compile the Active Project 15

Compilers 93

Compiling source code 93

Component Instance

    generation properties 21

    port mapping 24

    port mappings 21

    signals and ports 21

Component Instances 20

    editing name 21

- port mappings 23
- properties 21
- Signals and Ports 23
- Component Model
  - default port mapping 24
  - signals and ports 23
- Component Models 19, 21
  - Class Methods 22, 23, 77
  - Sequencer Process 85
- Components 19
  - Component Instances 20
  - Signals and Ports 20, 23
  - signals and ports 23
- Conditional State Transitions 103, 104
- Conditional State Values 43
- Constrained Random Number Generation 79, 80
  - also see Constraints
- Constraints
  - adding to class 72
- Context Sensitive Help 7
- Continuous Setups and Holds 52
- Count Clock Edges 51
- Create Default Clock Generator 18
- Critical Regions 82
  - defining 67
- Cycle Based Properties
  - default Clock 34
  - default Clock Edge 34
- Cycle Based Settings
  - delay after clock edge 38
- D**
- Data Packets - see Classes
- Data Packing - see Packing
- Data Structures - see Classes
- Data Targets - see Variables
- Debug Run 99
- Default Class Library 69
- Default Clock 18
- Default Diagram Settings 18, 34
- Default Port Mapping 24
- Default Sample Action 37
- Default Sample Condition 37
- Delay After Clock Edge 38
- Delay Settings - default for transaction 39
- Delays 50
  - also see Parameters
  - attaching to samples 60
  - conditional 51, 60
  - creating 50
  - cycle-based 51
  - Enable HDL Code Generation 36
  - properties 49, 51
  - resolving multiple delays 51
  - specifying the order of 45
  - verbosity 37
- Design Flow 9
- Diagram Calls 85
  - inserting 85
- Diagram Settings 34
  - default 18
  - general settings 36
  - language specific settings 38
- Diagram Settings Dialog 34
- Diagram-Level Variables 42
- Diagrams
  - adding items 44
  - adding parameters 49
  - adding to project 27
  - Class Methods 77, 78
  - creating 11, 27, 44
  - cycle based properties 34
  - default clocking domain 34
  - diagram-level variables 72
    - also see Variables
  - drawing waveforms 42
  - Execution 36
  - extracting MUT ports into 28
  - including library files 34
  - inserting HDL code 66, 67
  - properties 33
  - settings 34
  - Template Diagram 11, 19
  - Transaction 19, 27
    - view generated source code 12
- Display Applied Inputs 38
- Documentation Markers 68
- Driven Flags 42
- Driving Events 41
- E**
- Edge Properties
  - earliest transitions 51
  - latest transitions 51

- multiple delay resolution 51
- Edge Properties Dialog 46, 51
- Edit Bus State Dialog 42, 43
  - variables 43
- Elements 73, 74
- Enable Abort Code 38
- Enable Code Generation 36
- Enable HDL Code Generation 50
  - for Samples 56
- Enable Reference Model 22
- Enable TestBuilder Integration 9, 17
- Enable Transaction Manager 22
- End Diagram Markers 64
  - event 74
- Events 41
- Exit Loop When 66
- External Program Integration
  - settings 15
  - simulators 15
- Extract Ports from MUT 11, 28

**F**

- Falling Edge Sensitive 47

## Fields

- adding to a class 71
- defining from a file 82
- properties 73
- static 73
- structure types 73, 74

- File Input 74, 81

- File Output 74, 81

- File Structure Types 74

- fixed\_len\_string 74

- For Loops 65, 66

- Full Expect 58

- samples 58

**G**

- Generate Test Bench 13

- Generated Code - see Code Generation

- Generated Files

- adding keywords 23

- Generating Random Values

- in TestBuilder 113

- Generating the Test Bench 93

**H**

- HDL Code Markers 66, 67

- High Order (Packing) 76

## Holds

- also see Parameters

- continuous 52

- creating 52

- properties 49

- specifying the order of 45

**I**

- If-Then-Else statements 56

- creating 57

- Include Delay Time 38, 41

- Include Directories

- project simulation properties 96

- Including Library Files 34

- Initialization Diagram 41

- Initializing

- ports 24

- variables 73

- Inout Signals 42, 44

- Input Signals 44

- Insert Diagram Calls 12

- Insert Into Equation 61

- Instance Count 37

- Instance Settings 37

- int 74

- Internal Signals 44

- Is Apply Subroutine Input 50, 56, 104

**L**

- Language 17

- changing 22

- specific details per 107

- Libraries

- also see Class Libraries

- including in diagrams 34

- VHDL 25

- Library Directories 96

- Little Endian (Packing) 76

- Loop End 66

- Looping Markers 65

- Low Order (Packing) 76

**M**

- Make TB 13, 93

- Markers

- absolute 63

- attach to edge 63

- attach to time 63
- creating 63
- defined 63
- Documentation 68
- Enable HDL Code Generation 36
- End Diagram 64
- For Loop 66
- HDL Code 66, 67
- loops 65
- Pause Simulation 64
- relative 63
- Semaphore 67
- specifying the order of 45
- Time Break 68
- type 63
- verbosity 37
- Wait Until 65
- Master Transactions 101, 103
- Master Transactor 36
- Microsoft C++ 93
- Model Under Test - see MUT
- ModelSim 93
- Multiple Delay Resolution 51
  - earliest transitions 51
  - latest transition 51
- Multiplier 58
- MUT
  - adding to project 10, 28
  - extracting ports 11
  - extracting ports into diagrams 28
- N**
- NC Verilog 93
- NC VHDL 93
- Network Packing 76
- New Diagram Defaults 35
- New Project Wizard 9, 17
- Normal Order (Packing) 76
- O**
- Ordering Parameters 45
- Output Signals 44
- Outward Arrows 52
- P**
- Packing
  - big endian 76
  - bit normal 76
  - bit reverse 76
  - classes 73
  - high order 76
  - in Verilog 107
  - in VHDL 112
  - little endian 76
  - low order 76
  - network 76
  - normal order 76
  - reverse order 76
- Parameter Properties Dialog 49
- Parameter Variables
  - also see Variables
  - creating 30
  - using 30
- Parameters
  - adding 49
  - defining temporal expressions 49
  - Delays 49, 50
  - Enable HDL Code Generation 50
  - for Class Methods 78
  - Holds 49
  - Is Apply Subroutine Input 50
  - markers 63
  - properties 49
  - samples 55
  - Setups 49
  - specifying the order of 45
  - Temporal Expressions 45
- Pause Simulation (Verilog) 64
- Pipelining
  - instance count 37
- Port Mapping 21
  - bit slice 24
  - component instance 24
  - component model 24
  - initializing ports 24
- Post Semaphore Markers 67
- Prefix Generated Files With 23
- Project
  - adding diagrams 27
  - changing template files 91
  - Class Methods 77, 78
  - classes 23
  - Variables 23, 72
- Project Components
  - generation properties 21

- Project Generation Properties
  - add timestamp 23
  - prefix generated files with 23
  - Source Indent 23
  - transaction recording 23
- Project Language
  - changing 22
- Project Library 19, 20
- Project Simulation Properties 94
  - include directories 96
- Projects
  - adding files to 10
  - closing 17
  - Component Model 19
  - creating 17
  - opening 17
  - Project Components 19, 20
  - Project Window 18, 19, 20
  - saving 17
  - setting simulation run-time options 94
  - simulation properties 94
  - using the tree control 18
- Q**
- Queues 73, 74
- R**
- Random Transactions 89
- Randomization 73
  - generating constrained values 79, 80
- real 74
- Reference Model 22
- Relative Samples 56
- Repeat Loop 66
- Reset Defaults 35
- Restricted Expect
  - samples 58, 59
- Reverse Order (Packing) 76
- Rising Edge Sensitive 47
- Run Simulation 15
- RunRandomTransaction 89
- S**
- Sample Flags 60
- Samples
  - absolute 55, 56
  - Actions 57
  - actions 57, 59
  - blocking 58, 59
  - code generation 56
  - conditions 57
  - creating 56
  - data targets 61
  - default Action 37
  - default Condition 37
  - defined 55
  - Enable HDL Code Generation 36, 56
  - expects 58
  - multiplier 58
  - non-blocking 58, 59
  - point 55
  - referencing sample variables 60
  - relative 56
  - self-testing code 56
  - specifying the order of 45
  - storing sampled values 58
  - Temporal Expressions 45
  - triggering delays 60
  - triggering other samples 60
  - verbosity 37
  - window 55, 56
  - with delayed state transitions 59
- Save Defaults 36
- Select Variable Dialog 61
- Self-Testing Code
  - samples 56
- Semaphore Markers 67
- Semaphores 82
- Sensitive Edges 47
  - Enable HDL Code Generation 36
- Sequence Recognition
  - ordering events 46
  - verbosity 37
- Sequencer Process 12, 85
- Sequencing Transactions 87
- Serial Data 104, 105
- SetTransactorWeightings 89
- Setups
  - also see Parameters
  - continuous 52
  - creating 52
  - properties 49
  - specifying the order of 45
- Signal Button Bar 44
- Signal Direction

- default 36
  - Signal Direction - see Signals
  - Signal Properties 52
    - advanced register 53
    - sensitive edges 46
  - Signal Transitions
    - waiting for 101
  - Signals
    - adding to diagram 44
    - bi-directional 42
    - clocked 52
    - default Type 36
    - direction 42
    - driven 42
    - driving events on 41
    - inout 44
    - input 44
    - internal 44
    - output 44
  - Signals and Ports 23
    - Component 20
    - Component Model 19
    - of Component Instances 23
    - of Component Models 23
    - port mapping 21, 24
  - SignalScan
    - recording transactions for 23
  - signed\_logic 74
  - Simple Expect
    - samples 58, 59
  - Simulated Model 19
  - Simulating 97
    - with ModelSim 93
    - with third party simulators 93, 99
    - with VCS 93
  - Simulation 15
    - project properties 94
    - test bench 93
  - Simulation Button Bar 98
  - Simulation Files
    - for Verilog 107
    - needed for TestBuilder 112, 113
    - needed for VHDL 108
  - Simulation Mode 99
    - Auto Run 99
    - Debug Run 99
    - setting 99
  - Simulator and Compiler Settings Dialog 93
  - Slave Transactions 101
  - SlaveTransactor 36
  - Source Code
    - adding 79
    - generated for diagrams 12
  - Source Indent 23
  - State Transitions
    - conditional 103, 104
  - State Values
    - conditional 43
    - reading from file 81
    - simple variables 43
    - writing to file 81
  - State Variables 42
    - also see Variables
    - assigning values 86
    - creating 29
  - Static Fields 73
  - std\_logic 74
  - std\_logic\_vector 74
  - std\_ulogic 74
  - std\_ulogic\_vector 74
  - Store Sampled Value As Subroutine Output 58
  - Structure Types 73, 74
  - Sub-projects 19, 20
  - Sweep Tests 103, 104
- ## T
- Template Diagram 11, 19, 27
  - Template Files
    - adding Apply calls 12
    - adding diagram calls 85
    - changing for project 91
    - modifying 12
    - Sequencer Process 85
  - Temporal Expressions 45, 55
    - expressing with Parameters 49
  - Test Bench
    - compiling 98
    - generation 13, 93
    - looping 65
    - sequencing transactions 85
    - simulation 93, 97
  - Test Bench Techniques 101
  - Test Reader 87, 88
  - Test Vector Files 19, 81



- defining class fields from 82
  - format 82
- TestBencher Diagram Properties Dialog 33
- TestBencher Project
  - design flow 9
- TestBuilder 91, 112, 113, 114
  - enabling integration 9, 17, 22
  - Generating Random Values 113
  - simulating 93
- Third Party Simulators 93, 99
- time 74
- Time 0 (zero) 41
- Time Break Markers 68
- Timeout Settings 38
- Timing Diagrams - see Diagrams
- Top-level Template File 10
- Transaction Diagrams 19, 27
  - overview 27
- Transaction Generator 89
- Transaction Level Variables 29
- Transaction Logging
  - verbose 23
- Transaction Manager 22, 87, 88, 89
  - file format 88
  - in TestBuilder 114
  - in VHDL 109
  - run modes 89
  - VHDL procedures 109
- Transaction Recording 23
- Transactions
  - applying 43
  - calling 43
  - Execution 36
  - sequencing 12
- Triggered Delays 60
  - Delays
    - conditional 59
- Triggered Samples 60

**U**

- UART 104, 105
- UDT - see Classes
- unsigned\_int 74
- unsigned\_logic 74
- Update All Diagrams 35
- Update Existing 34
- Use Clauses (VHDL) 25

- User Defined Types - see Classes
- User Defined Variables 29
- User Source Code
  - adding class methods to diagrams 30
  - in Class Methods 79
- User Source File
  - adding to project 28
- User Source Files 19

**V**

- variable\_len\_string 74
- Variables 23, 69
  - applied 73
    - assigning values 86
  - constrained random generation 79
  - constraints 80
  - creating 72
  - diagram 29
  - diagram-level 42, 72
  - direction 29
  - editing 72
  - export signal states to a file 61
  - file input 81
  - file output 81
  - initial value 73
  - packing 76
  - project
    - accessing in Verilog 107
    - accessing in VHDL 112
  - Project Level 72
  - project-level 72
  - properties 73
  - random 73
  - referencing in a diagram 29
  - simple state 42
  - storing sampled state values 61
  - storing sampled value in 58
  - structure 74
  - structure type 73
    - arrays 74
    - associative arrays 74
    - element 74
    - File Input 74
    - File Output 74
    - File Structure Types 74
    - queues 74
  - transaction level 29

- usage 29
- user defined 29
- VCS 93
- Verbose
  - Delays 37
  - Markers 37
  - Samples 37
  - Sensitive Edges 37
  - Transaction Logging 23
- Verilog 107
  - enabling TestBuilder integration 9
- Verilog-XL 93
- VHDL 108, 109, 110, 112
- VHDL Libraries 25
- View Variables 29, 72

**W**

- Wait Semaphore Markers 67
- Wait Until Marker 65
- Waiting for Signal Transitions 101
- Waveforms 41
  - driving with variables 42
- waveperl.log 13
- Weightings Table 89
- While Loops 65, 66