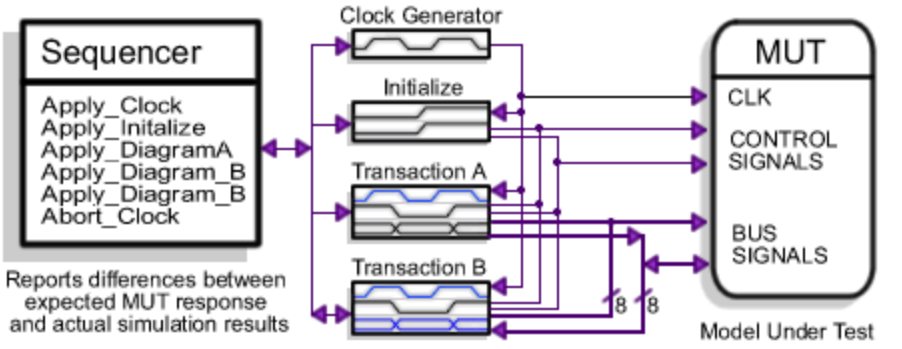
TestBencher Pro and Reactive Test Bench Help

Copyright © 2014, SynaptiCAD Sales, Inc.

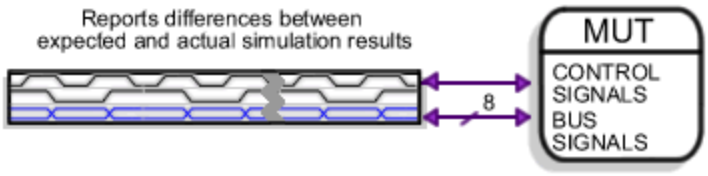
TestBencher Pro

Generates a Bus Functional Model from multiple diagrams



Reactive Test Bench Option

Generates a test bench from a single diagram



TestBencher Pro and Reactive Test Bench Help

Copyright Copyright © 2014, SynaptiCAD Sales, Inc., version 14

All rights reserved. No parts of this work may be reproduced in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems - without the written permission of the publisher.

Products that are referred to in this document may be either trademarks and/or registered trademarks of the respective owners. The publisher and the author make no claim to these trademarks.

While every precaution has been taken in the preparation of this document, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of information contained in this document or from the use of programs and source code that may accompany it. In no event shall the publisher and the author be liable for any loss of profit or any other commercial damage caused or alleged to have been caused directly or indirectly by this document.

Printed: September 2014 in (whereever you are located)

TestBench Generation

TestBencher Pro, Reactive TestBench Option for WaveFormer Pro

This is the manual for TestBencher Pro and the Reactive Test Bench Option. These products create self-testing test benches which generate error reports and react to the model under test during simulation. The test benches can be either cycle-based or time-based. TestBencher Pro creates bus functional models using multiple timing diagrams and a sequencer process to apply the diagrams, whereas the Reactive Test Bench option uses only a single timing diagram.

Table of Contents

Foreword 0

Chapter 1A: TestBencher Pro Design Flow 8

Step 1: Create a New Project ... 8
Step 2: Add the MUT to the Project ... 12
Step 3: Extract Ports into Template Diagram ... 14
Step 4: Create the Timing Transactor Diagrams ... 16
Step 5: Define Sequencer Process ... 17
Step 6: Generate the Test Bench ... 19
Step 7: Setup External Simulators ... 21
Step 8: Simulate the Test Bench ... 24
Step 9: Learning more about TestBencher Pro ... 24

Chapter 1B: Reactive Test Bench Option Design Flow 25

Step 1: Create a Project to hold the MUT ... 25
Step 2: Create the Reactive Diagram ... 28
Step 3: Generate the Reactive Test Bench ... 30

Chapter 2: Timing Transactor Basics 32

2.1 TestBencher Transactor Waveform Initilalization ... 32
2.2 Drawing Waveforms ... 33
2.3 Waveform Colors and Bi-Directional signals ... 35
2.4 Waveform States with Variables and Conditions ... 36
2.5 Simulated Signals ... 39
2.6 Sensitive Edges - a Blocking Construct ... 40
2.7 Controlling the Triggering Order of Parameters ... 40
2.8 Adding Signals Manually ... 42
2.9 Transactor Clock Domains and Waveform Code Generation ... 44
2.10 Debugging Transactor Clock Domains ... 46

Chapter 3: Transaction Samples 49

3.1 Adding a Sample ... 49
3.2 Sample Checking and Actions ... 51
3.3 Samples that Block Clock Domains ... 54
3.4 Samples Triggering a Delayed Transition or Another Sample ... 56
3.5 Using Sample Variables ... 58
3.6 Storing Sample Values in User Defined Variables ... 59

Copyright © 2014, SynaptiCAD Sales, Inc.

Chapter 4: Transaction Delays, Setups, and Holds 61

4.1 Delays 61
4.2 Setups and Holds 65
4.3 Creating Continuous Setups and Holds 67

Chapter 5: Transaction Markers 70

5.1 Adding a Marker to a Diagram 70
5.2 End Diagram Markers 73
5.3 Pause Simulation Marker (Verilog Only) 74
5.4 Wait Until Marker 74
5.5 Loop Markers 75
5.6 HDL Code Markers 77
5.7 Semaphore Markers (TestBencher Pro only) 78
5.8 Pipeline Boundary Markers (TestBencher Pro only) 78
5.9 Documentation and Time Break Markers 80

Chapter 6: Classes, Variables, and Files 82

6.1 Variables 82
6.2 Classes 86
6.3 Class Methods 87
6.4 Class Libraries 91
6.5 Random Number Generation and Constraints 92
6.6 Data Packing and Bit Streams 94
6.7 SynaptiCAD Test Vector File Format 95
6.8 Output Data to a File (VHDL & Verilog) 97
6.9 Input Data Dynamically from a File (VHDL) 100
6.10 (Verilog) Initialize Array from File 102
6.11 (VHDL) Initialize Array from File 104
6.12 Variables and Class Field Properties 107
6.13 Language Independent Types 109

Chapter 7: Transaction-Level Model and Transaction Sequencer 112

7.1 Template, Sequencer, and Transaction Calls 112
7.2 Master and Slave Transactions 115
7.3 Transaction Manager Overview 115
7.4 VHDL: TM fill with Post calls 117
7.5 VHDL: TM fill from a File 120
7.6 VHDL: TM fill from Random Generator 121
7.7 Randomizing Transaction Inputs 124
7.8 Changing a Project Template File 126

Copyright © 2014, SynaptiCAD Sales, Inc.

Chapter 8: Simulating a Test Bench 128

8.1 Simulate the Test Bench and Design ... 128
8.2 Project Simulation Properties Dialog ... 129
8.3 TestBencher Simulation Modes ... 134
8.4 Generate Command Files for Third Party Simulators ... 135
8.5 Parsing Package Files into Libraries ... 135

Chapter 9: Advanced Project Features and Properties 137

9.1 Sub-Projects ... 137
9.2 Component Instances of Sub-Projects ... 139
9.3 Golden Reference Models ... 140
9.4 Libraries and Use Clauses dialog (VHDL) ... 141
9.5 Signals and Ports for Components dialog ... 142
9.6 Project Generation Properties dialog ... 144
9.7 Diagram Properties dialog ... 146
9.8 Diagram Settings dialog (TestBencher only) ... 148
9.9 Project Menu and New Project Wizard dialog ... 151

Chapter 10: Test Bench Techniques and Examples 154

10.1 Pipeline Transactions ... 154
10.2 Waiting for Signal Transitions or Conditions ... 157
10.3 Burst Mode Transactions ... 159
10.4 Conditionally Moving Signal Edges (Sweep Tests) ... 160
10.5 Reading and Writing Serial Data ... 161
10.6 Testing a Counter Model ... 165
10.7 Different ways to bring in External Models ... 166
10.8 Different ways to Loop ... 167
10.9 Scoreboarding ... 168

Chapter 11: Language Specific Details and Test Bench Architecture 170

11.1 Verilog ... 170
11.2 VHDL ... 171
11.3 TestBuilder ... 175

Appendix A: Editor Commands 179

Appendix B: Supported Simulators 181

Appendix C: Language Independent Operators 182

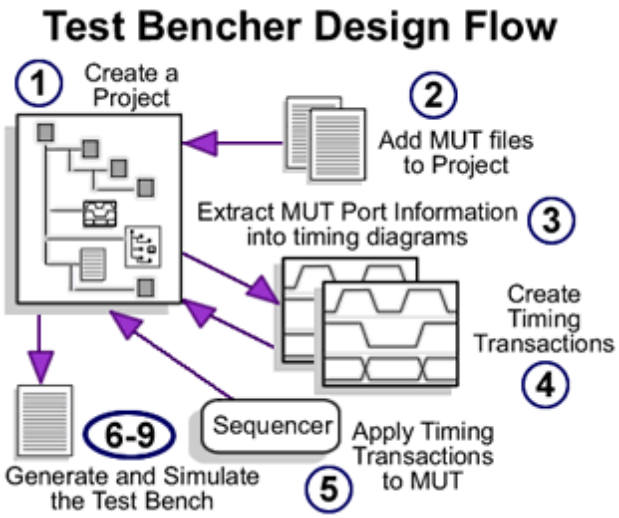
Copyright © 2014, SynaptiCAD Sales, Inc.

Index **191**

Copyright © 2014, SynaptiCAD Sales, Inc.

Chapter 1A: TestBencher Pro Design Flow

This chapter will cover the basic design flow for generating a transaction-level model (TLM) using TestBencher 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 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 8
Step 2: Add the MUT to the Project 12
Step 3: Extract Ports into Template Diagram 14
Step 4: Create the Timing Transactions 16
Step 5: Define Sequencer Process 17
Step 6: Generate the Test Bench 19
Step 7: Setup External Simulators 21
Step 8: Simulate the Test Bench 24

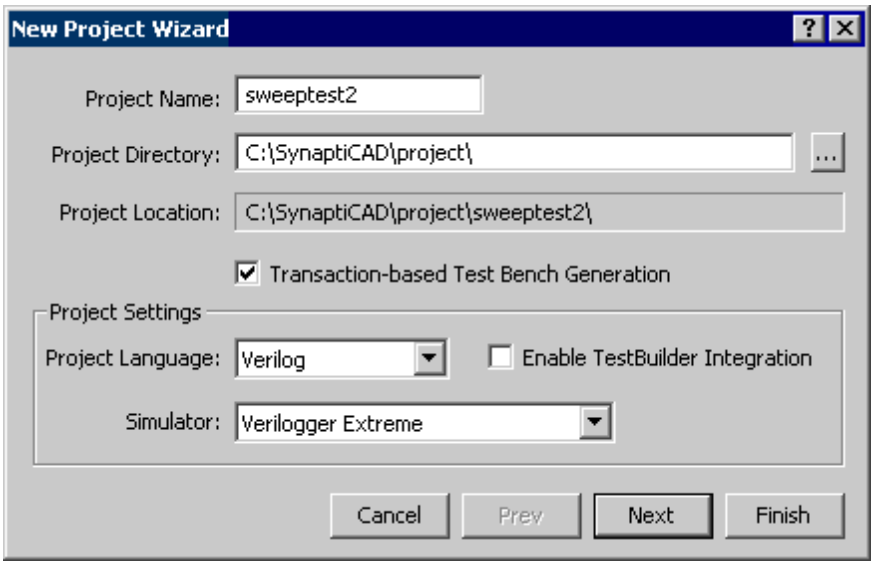
Step 1: Create a New Project

Projects hold all the information needed to generate the TestBencher Pro transaction-level model (TLM) including the transaction diagrams, top-level file, and the code generation settings. Chapter 9: TestBencher Advanced Project Features 137 describes projects, sub-projects 137 and creating instances of projects 139. Here are the steps for creating a basic single-level project.

Copyright © 2014, SynaptiCAD Sales, Inc.

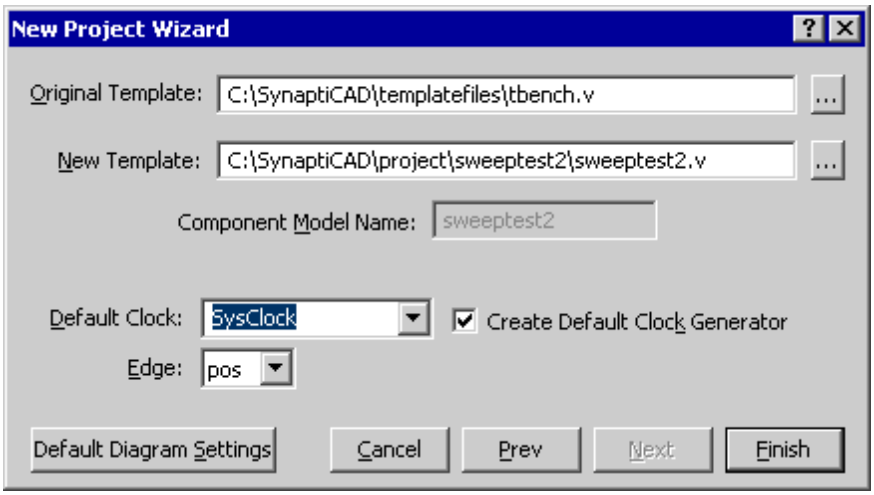
Create a project:

- 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.



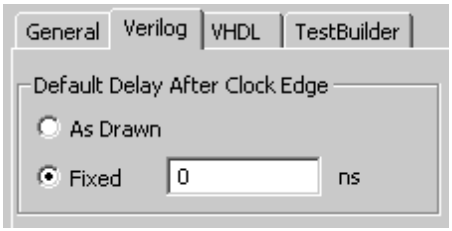
- Check the **Transaction-based Test Bench Generation** to make TestBencher generate transaction-level models instead of single diagram test benches.
- The **Language** drop down list defines the generated language for the test bench model. Certain features, such as valid signal types, are dependent on the generated language so this option affects the operation of TestBencher.
- If you are using Verilog and want to enable constrained random number generation, check the **Enable TestBuilder Integration** checkbox.
- Press the **Next** button to move to the next page of the dialog.

- If your project is clocked, then type in the name of the clocking signal, so that the project will generate a special transactor for the clock.
- Press the **Default Diagram Settings** button to edit the default diagram settings for new diagrams.



Copyright © 2014, SynaptiCAD Sales, Inc.

- By default, the **Default Delay After Clock Edge** setting is fixed to a delay time of 0 for your design. See Section 9.8: Diagram Setting Dialog (TestBencher Only) 148 for information about how this setting is used.

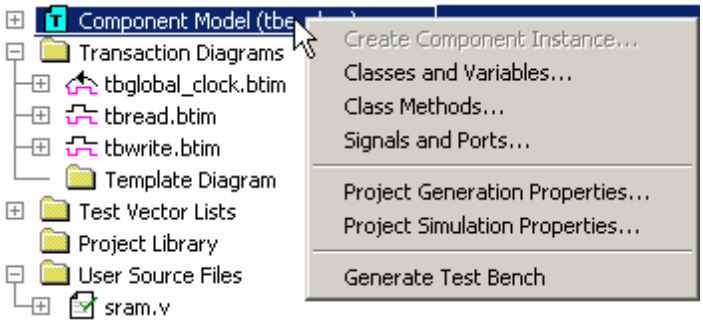


- Press the **Finish** button to close the *New Project Wizard* and create the new project.

Notice that TestBencher automatically opens the Project window and populates it with the top-level template file 112 and the template diagram 141. 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 transaction-level model.

Basic controls of the Project window:

- **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.



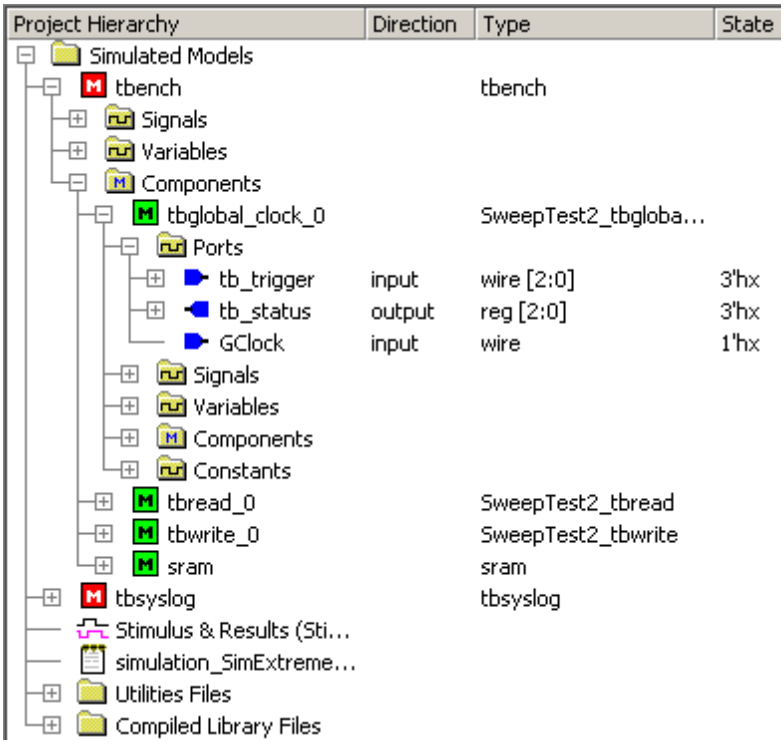
- **Double-left-click** on any node to perform the default action for that node (usually open that file or object in an appropriate editor).
- To expand or hide branches of a tree, click ⊞ or ⊟.
- Drag and drop the column headings to resize columns.

Basic Folders of the Project window:

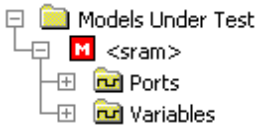
Several folders are created in each project and are used to organize files and objects at the different levels of the test bench. Each of these folders will be discussed at different points in the manual. As an overview:

Copyright © 2014, SynaptiCAD Sales, Inc.

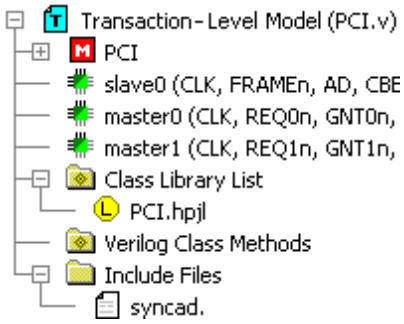
- **Simulated Models** folder contains the complete compiled design, the *Stimulus & Results* diagram, and the log file from the simulator. See Chapter 8: Generation and Simulation 128 for more information. This area is used for analyzing and debugging a compiled simulation.



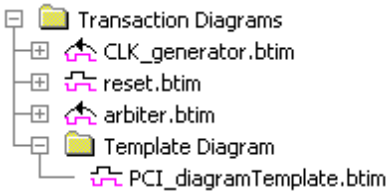
- **Models Under Test** contains the models with the ports that will hook up to the test bench. By default, the first model that is not instantiated within another model is selected as the model under test when the **Extract MUT Ports** button is pressed. After the first build, you can optionally right click on other modules and choose **Set as Model Under Test** to control which models are instantiated in the test bench. See Step 3: Extract Ports into Template Diagram 14 for more information.



- **Transaction-Level 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 7: Transaction-Level Model Transaction Sequencer 112 for more information.

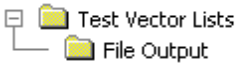


- **Transaction Diagrams** folder contains the template timing diagram, the timing diagrams that have been added to the project, and the associated generated source code files (in the level beneath the timing diagram). Master diagrams are shown with the waveform icon, and Slave diagrams are shown with the loop back icon. See Chapter 2: Timing Transactor Basics 32 for more information.

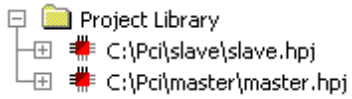


Copyright © 2014, SynaptiCAD Sales, Inc.

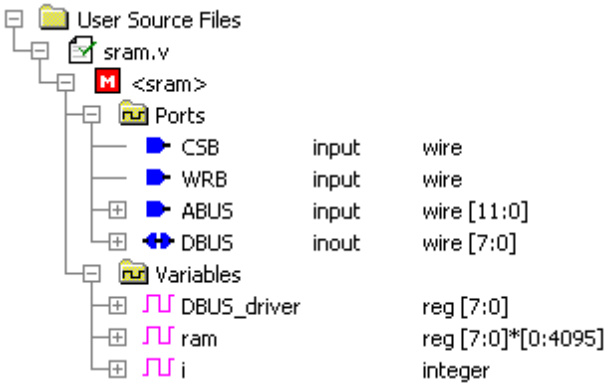
- **Test Vector Files** folder contains input and output test vector files. Chapter 8: Classes and Variables 82 for more information.



- **Project Library** folder holds any sub-projects that may be instantiated within the current project. Section 9.1: Sub-Projects 137 discusses this folder.



- **User Source Files** folder contains source files for use in the test bench. Files with a green check-mark icon have been compiled into the test bench; files with a red X icon have not yet been compiled Step 3: Extract MUT Ports into Template Diagram 14 has more information.

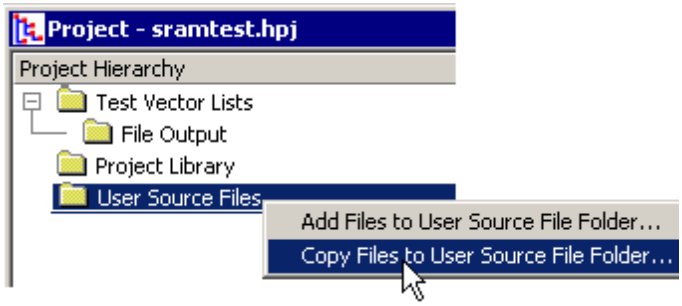


Step 2: Add the MUT to the Project

The Model Under Test (MUT) files of the design should be added to the project. TestBencher will use these files to instantiate MUTs into your testbench and extract the signal and port information for use in the transaction diagrams that stimulate the MUTs.

Copy or Add the MUT source files to the project:

- Right click on the ***User Source Files*** folder and either add or copy your design files to the project.
- The **Copy** menu copies the source file to the project folder and adds the copy to the project list. The **Add** function adds the file and its path without making a copy. Files can also be added by choosing the **Project > Add User Source File(s)** menu from the main bar.



- If your MUT requires definitions from a package compiled into a different library, follow the steps in Section 8.5: Parsing Package Files into Libraries 135 to ensure these packages are properly parsed. If your MUT parse does require files to be compiled into multiple libraries, you will probably need to directly set your desired model under test after the first parse, then parse again (see Step 3: Extract Ports into Template 14).

(Optional) Instantiate multiple MUTs in the Project

If your project only hooks up to one Model Under Test, MUT, that is the top level model in the source files then you can skip this and go on to Step 3. However, this product supports multiple MUTs and

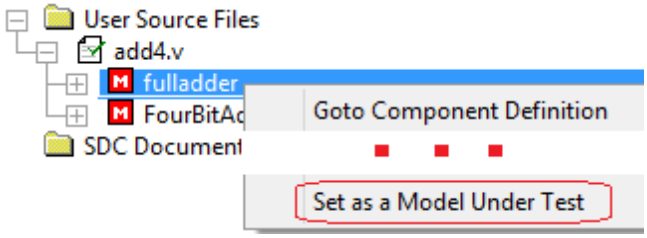
Copyright © 2014, SynaptiCAD Sales, Inc.

multiple instantiations of the same MUT. First build the project, then select your MUT and edit the port mappings of the new MUTs as desired. This should be done before creating the transactions so that you have all signals available during the transaction build step.

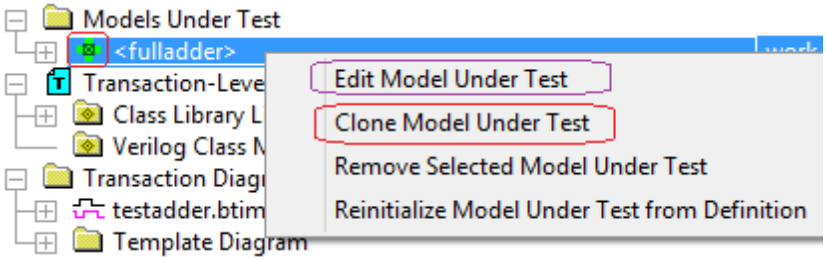
- Press the **Build Button** to parse the source files and populate the project window tree with module information.



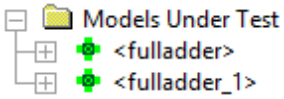
- Drill down into the User Source files node and right click on a MUT to open the context menu.
- Choose **Set as a Model Under Test.**



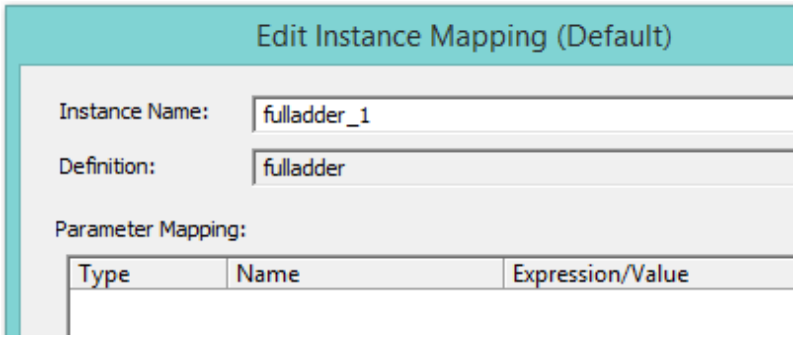
- Notice that the module has been added to the **Models Under Test** tree node. The green color indicates the port mappings are using defaults that match the port names of the MUT.



- Choosing **Clone Model Under Test** from the right click context menu (shown above), creates another instance of the MUT.

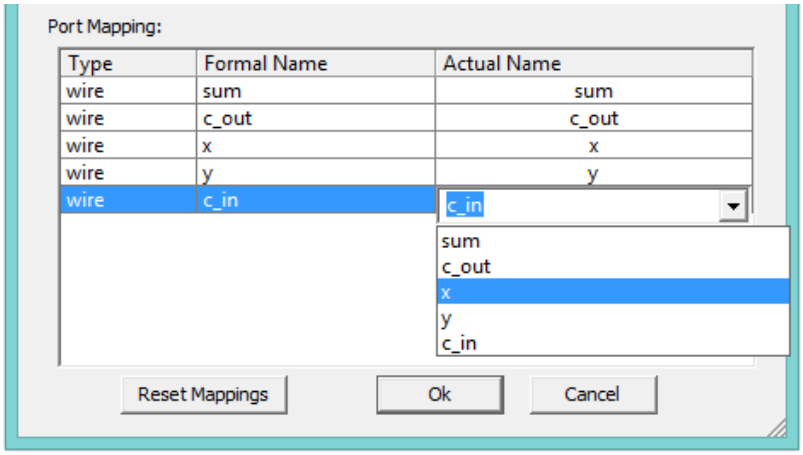


- The program guesses at the port mappings by matching same name signals. However, choosing **Edit Model Under Test** from the right click context menu (shown above), opens a dialog that allows the model instance name to be changed and the port and parameter mappings to be edited.



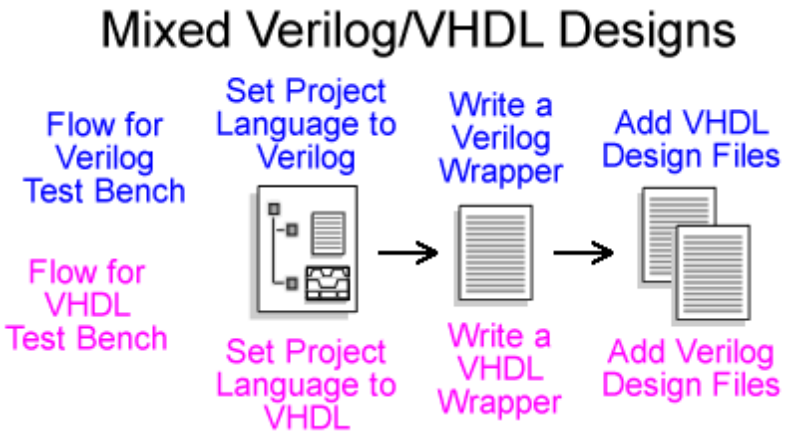
Copyright © 2014, SynaptiCAD Sales, Inc.

- Double clicking on the port name opens a drop-down that lets you choose a different port or type in a different name.
- After you edit the instance mappings the icon in the tree will turn RED, to indicate that the program will not overwrite the ports if other things change in the project.



Mixed VHDL and Verilog Designs:

If you have a multiple-language design or if you are more comfortable working in a particular HDL language, then the generated test bench and the design model do not have to be in the same language. You will need to write a wrapper module in the test bench generation language that instantiates the models in the other language. You will also need a multiple-language simulator to compile and debug the design.



- In Step 1, when you create the project, set the language to be the language that the test bench will be generated in. Also set the simulator to the name of your multiple-language simulator.
- Create a new file containg a wrapper module in the test bench language that instantiates the design model and maps signal types between the two different languages. The syntax for the instantiation of the design module will specified by your simulator. The ports for the wrapper will be the signals that the testbench drives and observes.
- Add the design files to **User Source Files** folder as described above. TestBencher can parse both VHDL and Verilog files and also send the files to the multiple-language simulator.

Step 3: Extract Ports into Template Diagram

When TestBencher created the project, it also generated a template diagram. New transaction diagrams will contain the same signals, waveforms, parameters, and properties as the template diagram. It is a good idea to setup the template diagram before creating the timing transactions so

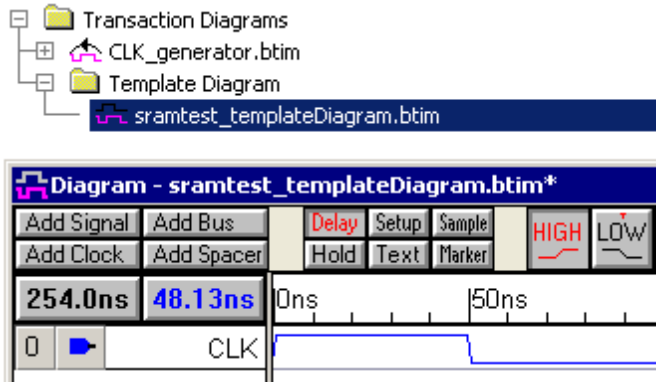
Copyright © 2014, SynaptiCAD Sales, Inc.

that all of the timing transactions will have the same interface signals.

Open the Template Diagram:

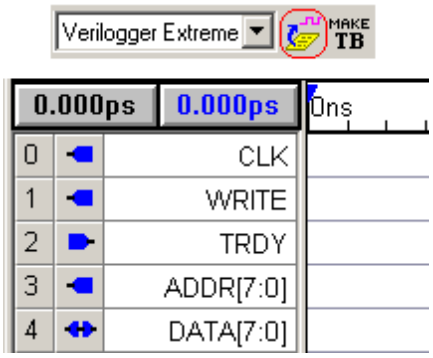
- In the *Project* window, under the *Template Diagram* folder, double click on diagram named ***projectname*_templateDiagram.btim** to open the template diagram window.

- If a default clock was created during the Project creation step, then the template diagram will contain the clock as an input signal, and there will also be a clock generator diagram that contains the clock as an output. The clock should only be driven by one timing transaction so that it can be properly aborted at the end of simulation.



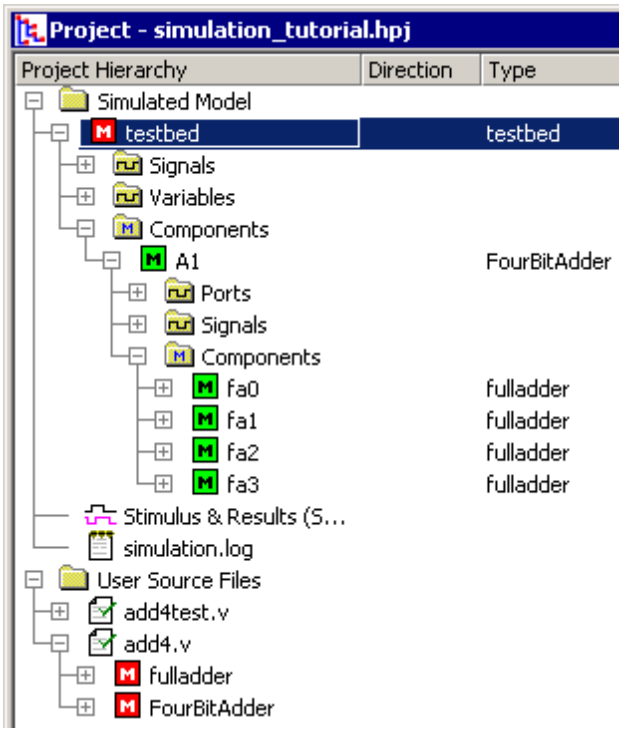
Extract the ports of the MUT into the template diagram:

- Click the **Extract Ports from MUT** button to build the MUT and insert the signals for the MUT ports into the template diagram.
- The blue icons indicate the direction of the signals. Use the mouse to hover over the direction icons and a tool-tip will tell you what each image means.



Copyright © 2014, SynaptiCAD Sales, Inc.

- By default, TestBencher extracted ports from the top-level models (those not instantiated within other models).
- After the first build, you can optionally right click on other modules and choose **Set as a Model Under Test** to control which models are instantiated in the test bench.
- Notice that the User Source files have green check marks to indicated that they are built.
- Double clicking on any component will open a editor scrolled to that place in the code.



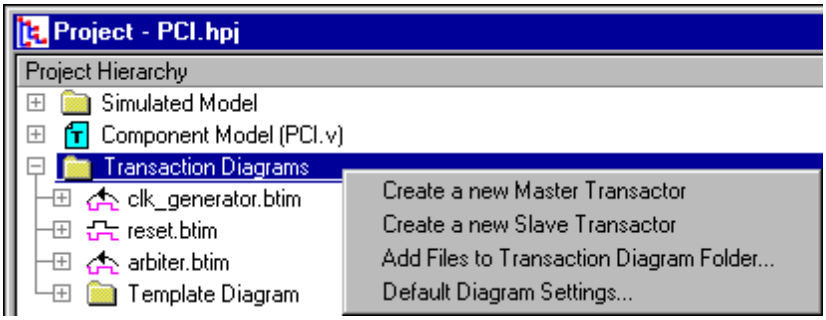
- Add other graphical objects, end diagram markers, and variables that you want to appear in all the test bench transactions.
- Choose the **File > Save Timing Diagram** to save the template diagram.

Step 4: Create the Timing Transactor Diagrams

TestBencher generates a transactor for each timing diagram in the project that represents a reusable interface specification of the bus-functional model that you are creating (e.g., read cycle, write cycle, interrupt cycle). These transactors are *modules* for Verilog, *entity/architecture* pairs for VHDL and *classes* for TestBuilder. Regardless of the language, the transactors use the same general architecture. And in all languages, the transactors have a similar functional API that can be used to trigger them (diagram apply calls). The **tutorials** and ***Chapters 2-6*** describe how to draw the timing diagrams and control the generated code. When a new timing diagram is created, it will contain all of the signals and properties of the *Template Diagram*. Do the following steps for each timing transactor that you want to call in the test bench:

Create a new timing transactor:

- In the *Project* window, right-click on the *Transaction Diagrams* folder and choose one of the **Create a new ... Transactor** menus. This opens a file dialog to create and save the diagram.



Copyright © 2014, SynaptiCAD Sales, Inc.

- After saving the diagram, it will be added to the *Transaction Diagrams* folder. The diagram will contain the template signals and properties.

Master Transactors are transactions that run once and then stop and are represented by the waveform icon.

Slave Transactors are transactions that will loop continuously until they receive an abort call and are shown by icon with a waveform and a loop-back line. Slave transactions are usually started at the beginning of a test bench and run until the test bench completes.

- Create a timing diagram by sketching the waveforms using the *Timing Diagram Editor*. Optional objects 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. See chapters 2-6 for how to create the transactions.
- 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.

Change to and from Slave and Master:

The Slave and Master setting for a timing diagram affects the way code is generated for the timing transaction. After the timing diagram is created, you can change to the other setting by doing the following:

- Right click on the timing diagram name in the Project window and choose Diagram Settings 148 from the context menu to open the *Diagram Settings* dialog.
- Then check either the **Slave Transactor** or **Master Transactor** radio button.

View the generated HDL source code:

Since the source code for a timing diagram is regenerated each time it is saved, you should not make any manual changes to the code. However, it is sometimes useful to see how the low level code changes based on the constructs that are placed in the diagram.

- Click the **Source Code** button to open an editor and view the code.



Step 5: Define Sequencer Process

TestBencher generates all of the top-level code for the test bench into the Transaction-Level Model (TLM) file, including the code that instantiates the model under test and connects it to the test bench model. This is a special template file that will contain both generated code and user-defined code. Any code that appears between a beginning-comment an end-comment will be regenerated each time the test bench is made, so this code should not be edited by hand. The following is an example of such a code block for a Verilog test bench:

```
// $DataStructureInstantiationsForAllDiagrams
      ...
// End $DataStructureInstantiationsForAllDiagrams
```

The TLM file also contains the Sequencer process that controls the sequencing of transactions during the simulation of the TLM. The Sequencer process is the place where you typically add function calls

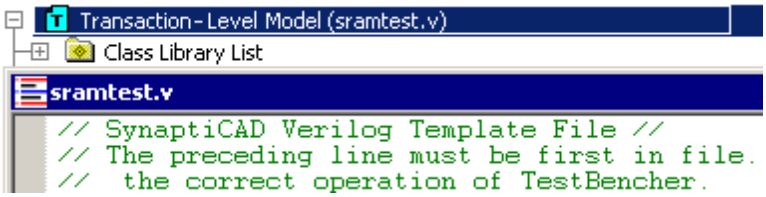
Copyright © 2014, SynaptiCAD Sales, Inc.

to apply transactions to the model under test.

If you need more complex transaction sequencing than is possible using simple Apply calls, you can enable generation of a Transaction Manager which allows transactions to be read from files, randomly generated, or placed in one or more queues where they can be scheduled for future execution. Chapter 7: Transaction-Level Model and Transaction Sequencer 112 describes the sequencer process, the transaction manager, and the TLM template file in detail.

Open the TLM file and find the Sequencer Process

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

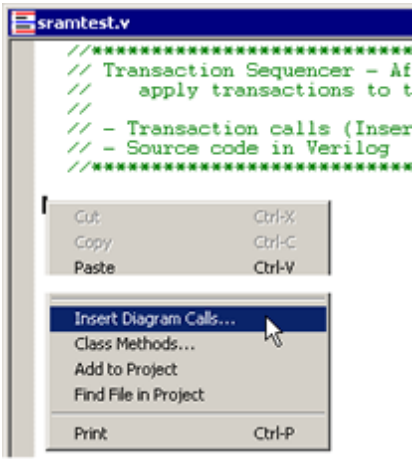


- Scroll down in the template file until you find the comment block that contains the Sequencer Process. The comment will contain the following text:

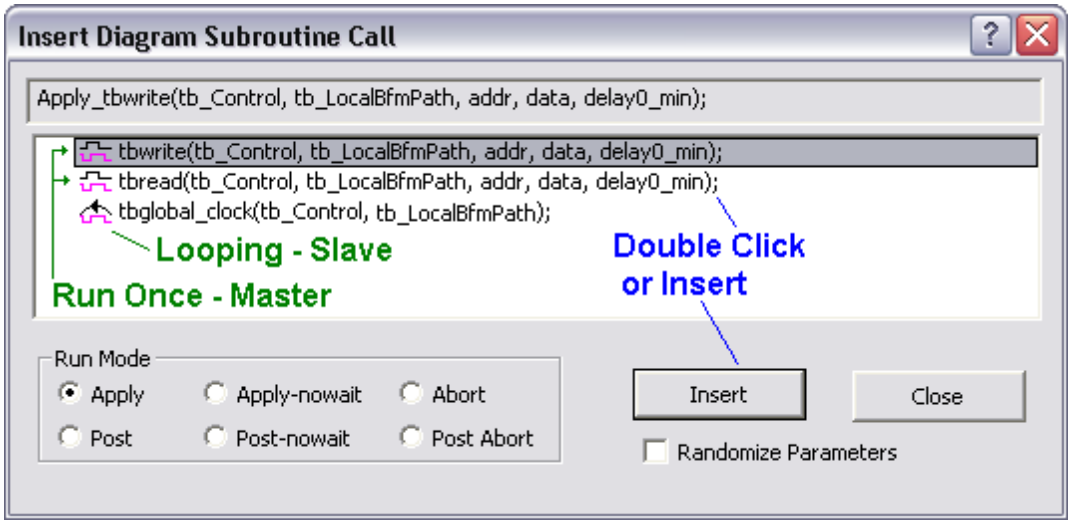
```
//*********************************************************
// Transaction Sequencer - After this comment, define how to
//    apply transactions to the model under test using:
```

Use the Insert Diagram Subroutine Call dialog to make Transaction Calls:

- Click in the *Editor* window just below this comment at the place that you want to add a transaction call.
- 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 transactors that have been added to the project.



Copyright © 2014, SynaptiCAD Sales, Inc.



- Arrange the windows so that you can see the dialog and sequencer editor at the same time.
- Select a ***timing diagram name*** and a **run mode**, then press the **Insert** button to insert the code into the editor.
    - 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. The **Abort** function will stop a transaction during mid-execution and is typically used to stop the execution of a slave transaction (which otherwise loops forever). For more information see Section 7.1: Template, Sequencer, and Transaction Calls 112.
    - A **Post** call schedules a transaction for future execution by placing it into a Transaction Manager queue (instead of immediately applying the transaction). The Transaction Manger must be enabled as described in Section 7.3 Transaction Manager Overview 115 for these calls to operate.
    - The **Randomize Parameters** box changes **Apply** and **Post** calls so that the data parameters of the transaction will be randomly generated instead of being specified in the procedure call. For details, see Section 7.7 Randomizing Transaction Inputs 124.
- Notice that the Apply statement was inserted at the same line as your cursor. The *Insert Diagram Call* dialog is a modeless dialog, so it can remain open while you perform other actions. If the applied transaction contains variables, then both a comment and a code line will be added. Edit the code line to apply the real parameters and leave the comment line 'as-is' to document the parameter names. Below is an edited example of an apply call in Verilog.

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

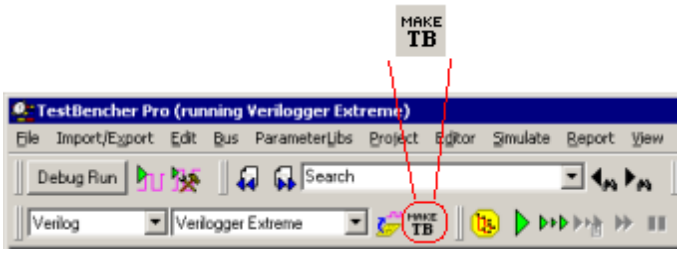
Step 6: Generate the Test Bench

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.

Copyright © 2014, SynaptiCAD Sales, Inc.

Generate the bus-functional model with the MAKE TB button:

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

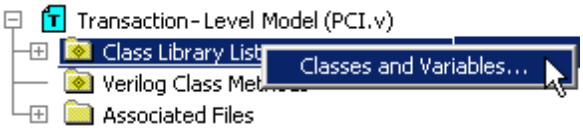


Viewing the Generated Code after MAKE TB:

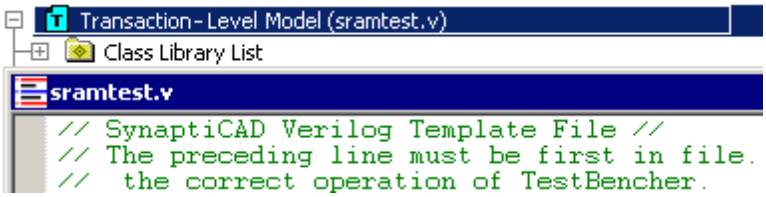
- TestBencher verifies that the source code files for each of the timing diagrams in the project is up-to-date. The diagram generates its transaction files each time it is saved. You can press the **Source Code** button to view the transaction code for each diagram



- All class libraries are regenerated. Double clicking on the **Classes and Variables** folder will show you the structures for the Project.



- The macro statements in the project's template files are expanded. Double-click on the **Transaction-Level Model** folder to open the template file.



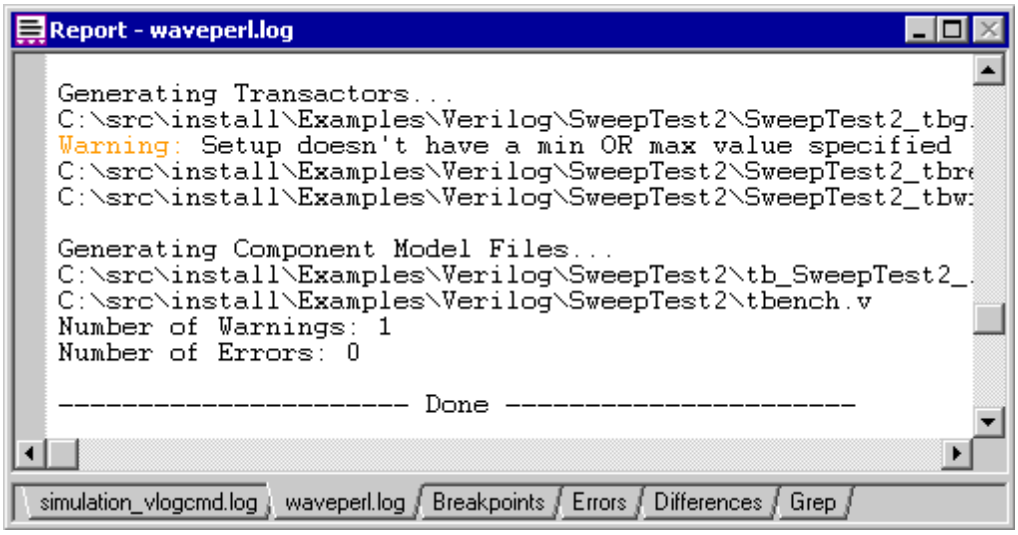
- During the Generate Test Bench process, several files are generated and they depend on the generation language (see Chapter 11: Language Specific Details 170 for more information). All of the generated files are displayed in the *Project* window.

Generation Code Errors and Warnings:

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. TestBencher 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.

- 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.

Copyright © 2014, SynaptiCAD Sales, Inc.



Step 7: Setup External Simulators

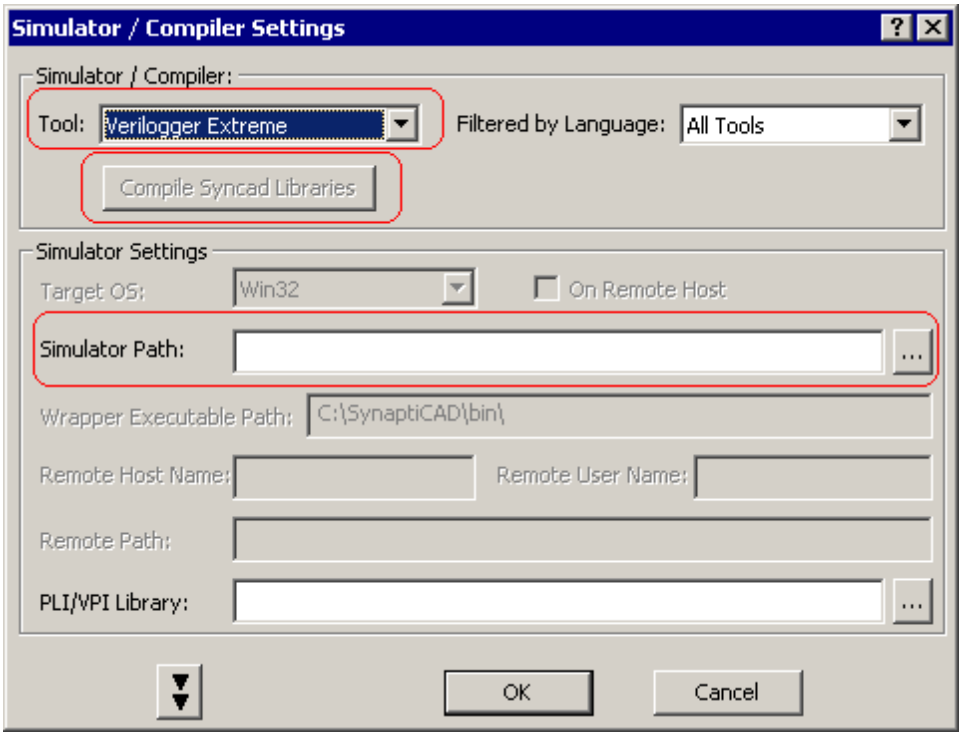
At this point, all of the test bench code has been generated and you can take the source code files from the project directory and simulate them using any VHDL or Verilog simulator. However, TestBencher Pro can act as a graphical debugger for your simulator. In the beginning, when you are debugging the test bench itself, it is recommended that you work within TestBencher so that you can quickly make changes to the graphical transactors and regenerate the testbench.

Use the Simulator and Compiler Settings dialog to set the paths for external tools:

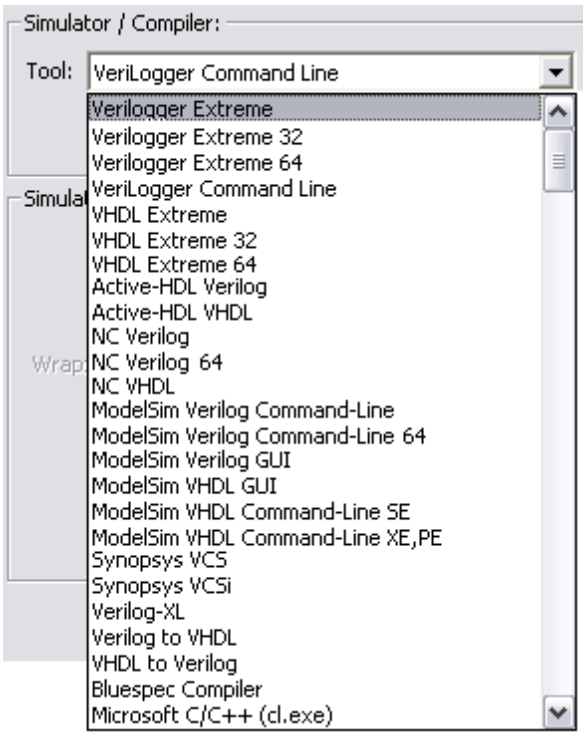
TestBencher Pro needs to know where your VHDL/Verilog simulator or C++ compiler is located and also needs to compile the Syncad library. If you are using one of SynaptiCAD's VeriLogger simulators you can skip this section, because the simulator was setup during installation. The external tool settings are saved in the file simulatorconfiguration.ini (located in the SynaptiCAD Local App Data directory) each time the program is closed (not in the project file), because the locations of the simulators are machine-specific. If you transfer your project to a new computer, you will need to repeat this step on the new machine. Also, if you upgrade simulators or upgrade TestBencher Pro, then you will need to recompile this library.

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

Copyright © 2014, SynaptiCAD Sales, Inc.



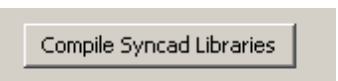
- In the **Tool** drop-down, choose your simulator or compiler. VeriLogger Extreme will run either in 32bit mode or 64bit mode depending on your OS version (e.g. 32bit or 64bit OS). To force 32bit operation on a 64bit system, select **VeriLogger Extreme 32**.



- In the **Simulator Path** edit box, either type in the path name or use the **browse button** to search for the path.



- The **Compile Syncad Libraries** button compiles the Syncad library needed by external simulators and compilers for



Copyright © 2014, SynaptiCAD Sales, Inc.

SynaptiCAD projects. This ***needs to be done one time before using a new simulator***.

- The **PLI/VPI Library** setting is for the full path to the PLI or VPI library that the selected tool will use.

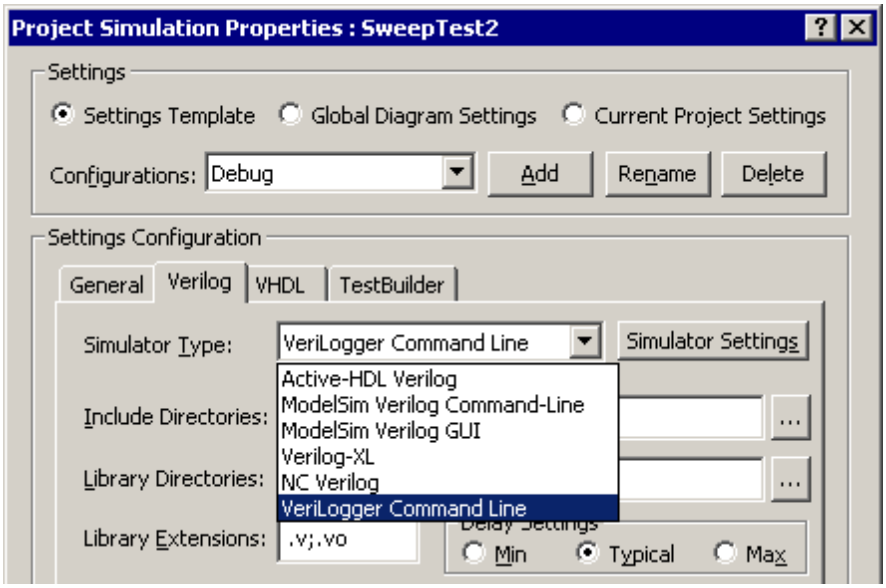


- Continue to setup the paths for each tool that you are interested in using. When you are done, click **OK** button to close the dialog.

Use the Project Simulation Properties dialog to set simulator specific commands:

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. To specify which tool to use and its' default settings:

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

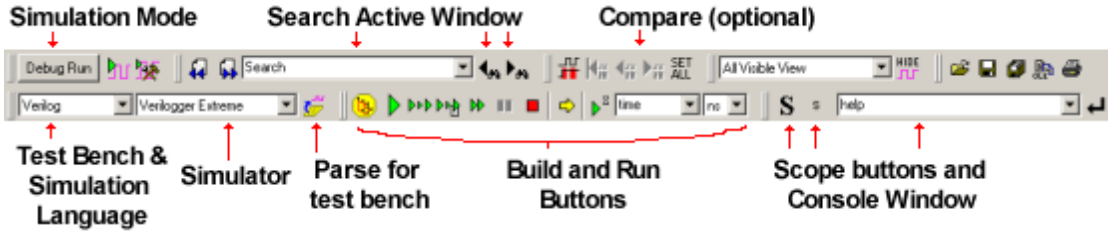


- The **Settings Template** radio button indicates the dialog will be editing the default project settings for all future projects. These settings are saved in the INI file. **Current Project Settings** effects only the project that is currently open.
  - Click on the language tab for the external tool that you are setting up.
  - From the **Simulator Type** drop-down, choose the external tool.
- The **Global Diagram Settings** radio button changes 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, you can set the simulator to your external simulator. This should only be necessary if you need to enter direct hdl code for simulated signals (not typically necessary).
- Section 8.2: Project Simulation Properties Dialog 129 has detailed information on the the controls in this dialog.
- Press the **OK** button to close the dialog.

Copyright © 2014, SynaptiCAD Sales, Inc.

Step 8: Simulate the Test Bench

TestBencher 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 and VeriLogger manual. Simulations are controlled by the buttons on the simulation button bar.



Build the Code:

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

- Check the *Report* window **Error** tab to see if there are any compile errors in the simulation results tab. If you cannot see the Report window, choose the **Window > Report** menu option to bring the window to the front.

Start a Simulation:

- Press one of the green Run buttons to start the simulator. The status of the simulator is reported in the lower right hand corner of the screen. or <F5> key

View Simulation Results:

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

Step 9: Learning more about TestBencher Pro

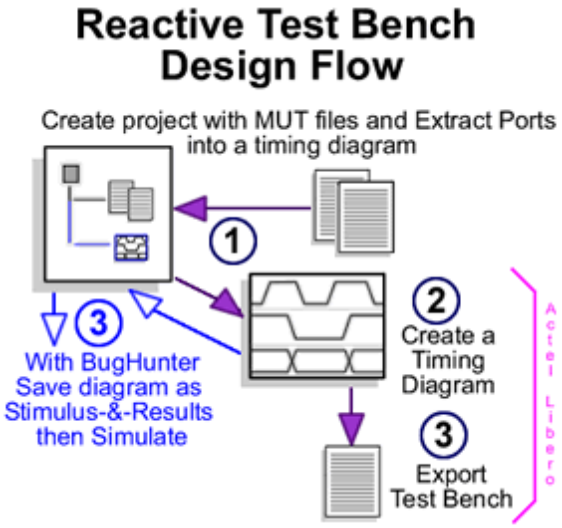
The quickest way to learn how to use TestBencher Pro is to work through the following:

- Perform the Test Bencher Pro Basic Tutorial which can be reached using the **Help > Tutorials** menu function. You will perform the above steps to create a transaction-level model for an SRAM model and simulate the results.
- Look through the **SynaptiCAD > Examples > TestBencher** directory to see several test bench examples ranging from simple test benches to PCI and ARM bus examples. These are also described in Chapter 10: Test Bench Techniques and Examples 154.
- Read through the rest of the manual (especially the chapter beginnings) so that you get an idea of some of the advanced features that are available to help you construct your models such as: constrained random number generation, loops in test benches, generating code from samples, and controlling execution.
- In addition, the Timing Diagram Editors Manual covers the drawing features of the editing environment, and the BugHunter and VeriLogger Manual covers setting up your simulator with TestBencher Pro and the simulation debugging features of the tool.

Copyright © 2014, SynaptiCAD Sales, Inc.

Chapter 1B: Reactive Test Bench Option Design Flow

The Reactive Test Bench Option is a sub-set of the TestBencher Pro product. It enables the creation of self-testing testbenches using a single timing diagram, rather than the multi-diagram bus-functional models created by TestBencher Pro. The Reactive test benches can respond to the model under test during simulation and also generate reports that describe the performance of the simulation. The Reactive Test Bench Generation Option can be added to WaveFormer Pro, WaveFormer Lite, DataSheet Pro, and BugHunter Pro.



Use the Stars to Find the Reactive Test Bench Features in the Manual:

Because everything has to be accomplished in a single diagram, the design flow for using the Reactive Test Bench Option is slightly different from using TestBencher Pro. Use this chapter to find information on the Reactive Test Bench **Project functions** and **Exporting the Test Bench**.

- The Reactive Test Bench Generation Option shares most of the TestBencher Pro transaction level features for defining a timing diagram and this information is in the chapters marked with the star on the book.

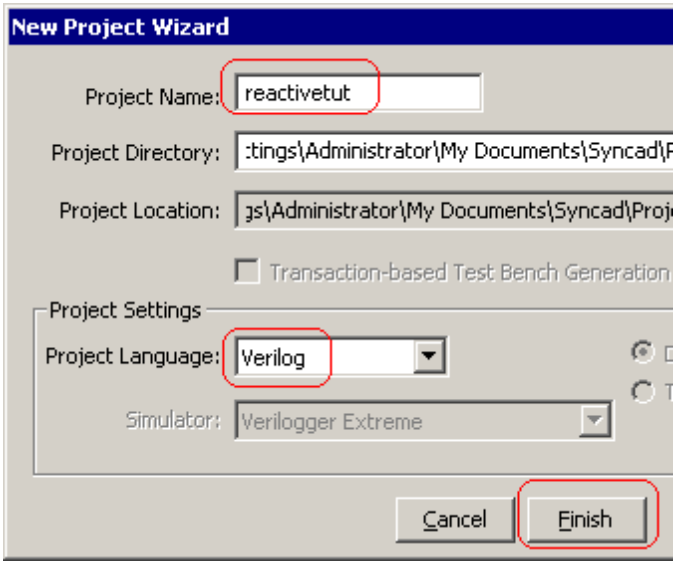
Step 1: Create a Project to hold the MUT

The project for the Reactive Test Bench Option is very simple because it just holds the model under test files and the generation language setting. If you are running the BugHunter interface, the project will expand to show a hierarchical view of the models and the stimulus-and-results diagram. Actel Libero users can skip this step because they will create the project in Libero, and Libero will launch WaveFormer Lite and extract the signals automatically.

Copyright © 2014, SynaptiCAD Sales, Inc.

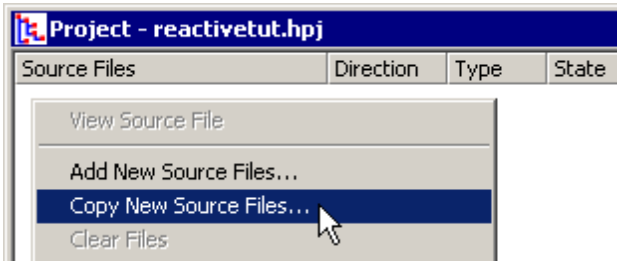
Create a Project

- Choose **Project > New Project** menu function to open the *New Project Wizard* dialog.
- In the **Project Name** box, type in the name. This will be both the name of the project and the directory where the project and associated files are stored.
- Use **Project Language** to set the parser language for the model under test files.
- Click the **Finish** button to create the project and close the dialog.

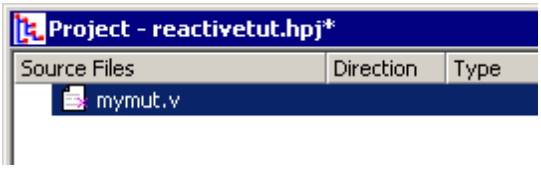


Add the Model Files

- In the **Project** window, under the **Source Files** section, right click and choose **Copy New Source Files** from the context menu. The picture shows WaveFormer's interface. BugHunter and TestBencher projects have a **User Source Files** folder that you can right click on. This will open a file open dialog.



- Use the Browse button to find the model under test files and add them to the project.



(Optional) Instantiate multiple MUTs in the Project

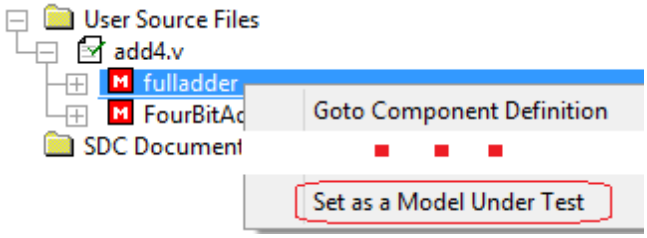
If your project only hooks up to one Model Under Test, MUT, that is the top level model in the source files then you can skip to the next blue heading. However, this product supports multiple MUTs and multiple instantiations of the same MUT. First build the project, then select your MUT and edit the port mappings of the new MUTs as desired. This should be done before creating the transactions so that you have all signals available during the transaction build step.

- Press the **Build Button** to parse the source files and populate the project window tree with module information.

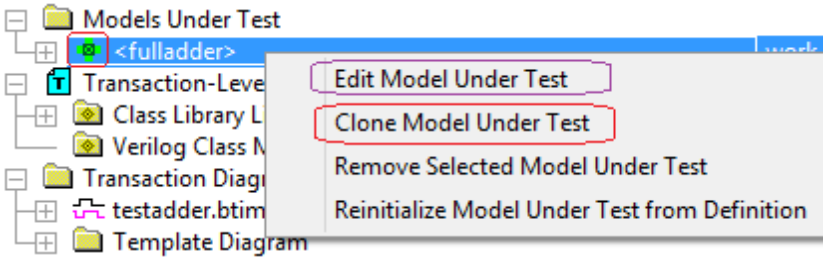


Copyright © 2014, SynaptiCAD Sales, Inc.

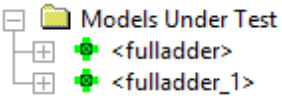
- Drill down into the User Source files node and right click on a MUT to open the context menu.
- Choose **Set as a Model Under Test.**



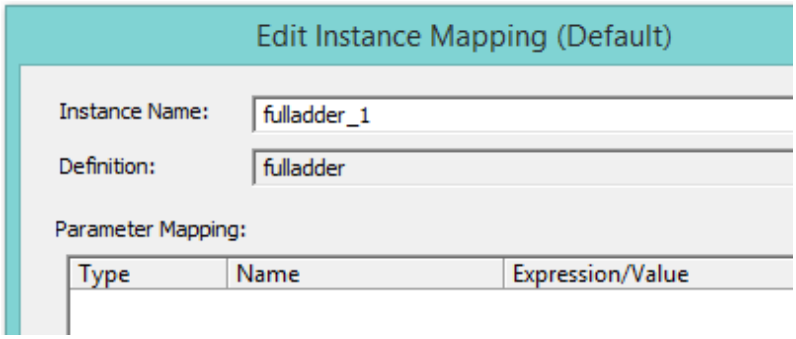
- Notice that the module has been added to the **Models Under Test** tree node. The green color indicates the port mappings are using defaults that match the port names of the MUT.



- Choosing **Clone Model Under Test** from the right click context menu (shown above), creates another instance of the MUT.

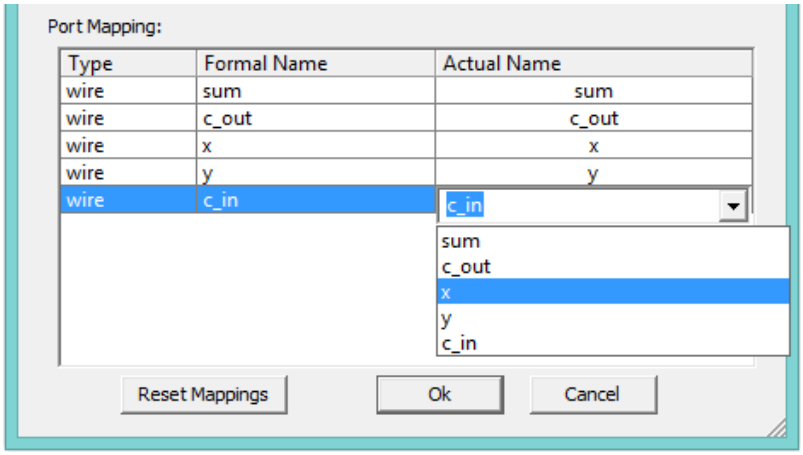


- The program guesses at the port mappings by matching same name signals. However, choosing **Edit Model Under Test** from the right click context menu (shown above), opens a dialog that allows the model instance name to be changed and the port and parameter mappings to be edited.



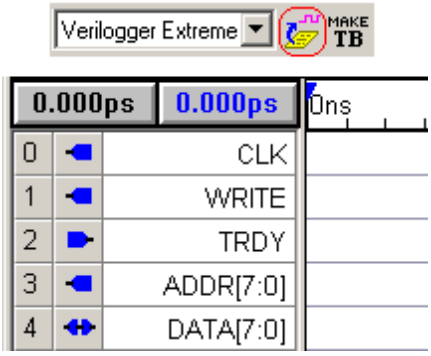
Copyright © 2014, SynaptiCAD Sales, Inc.

- Double clicking on the port name opens a drop-down that lets you choose a different port or type in a different name.
- After you edit the instance mappings the icon in the tree will turn RED, to indicate that the program will not overwrite the ports if other things change in the project.



Extract the MUT Ports

- Press the **Extract MUT ports into Diagram** button on the main window button bar to extract the top level ports from **mymut.v** and dump them into a timing diagram window.

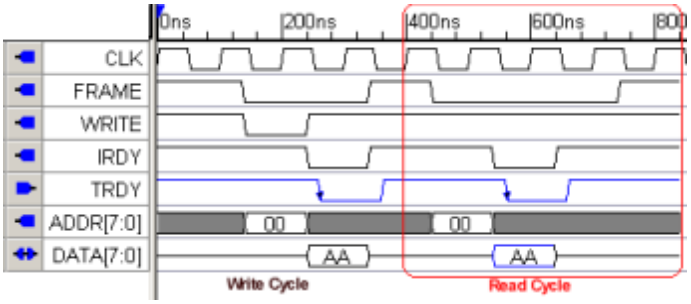


Step 2: Create the Reactive Diagram

Next draw the waveforms and other graphical objects in the diagram with the extracted signal information. You will want to read through the following chapters and do the Reactive Test Bench Tutorial to see the kinds of code that you can generate.

Draw the Stimulus and Expected waveforms:

Chapter 2: Transaction Waveforms and Signals 32 describes how to draw the waveforms. Since this option only supports single timing diagram test benches you can skip section 2.1 which discusses how timing diagrams are initialized. With the single timing diagram test bench, the diagram will initialized when you start drawing stimulus on to the test bench output signals.



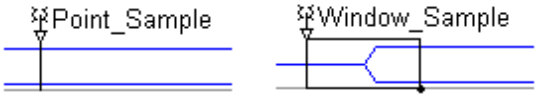
Reactive test bench generation also allows the option of creating "clock-based" test benches as well as the "time-based" test benches. Clock-based test benches delay in clock cycles instead of times,

Copyright © 2014, SynaptiCAD Sales, Inc.

allowing the user to change his clock frequency without needing to change his timing diagram. Clock-based test benches are also required when testing using high-speed "cycle-based" simulators. The clock setting is in the *Signal Properties* dialog of each signal.

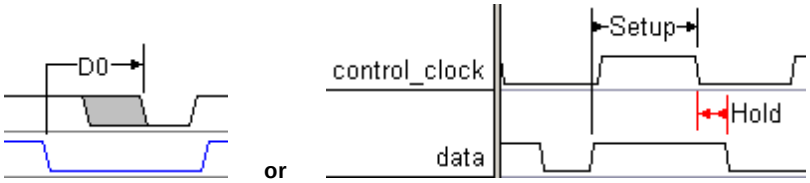
Add Samples (major construct for testing model output):

Chapter 3: Transaction Samples 49 describes how to add graphical Samples to test model output and change what stimulus is applied based on that output. During simulation, the code generated by the samples watch the output from the model under test and compare it to drawn states. The samples can perform a variety of functions such as pausing the simulation to debug a problem, reporting errors and warnings, user-defined actions, and triggering other samples.



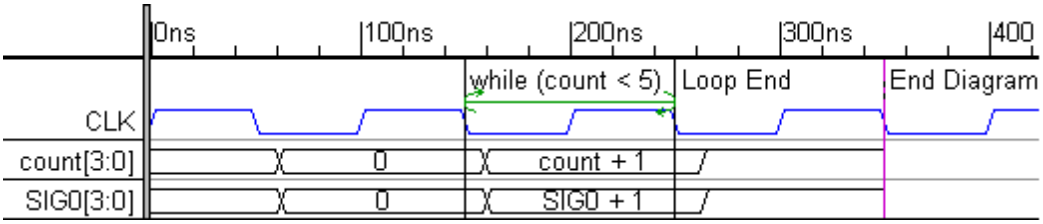
Add Delays, Setups, and Holds:

Chapter 4: Transaction Delays, Setups, and Holds 61 describes how code is generated for these parameters. The Setups and Holds will make code that monitors signal transitions that come back from the model under test during simulation and generate pass-fail reports.



Add Markers:

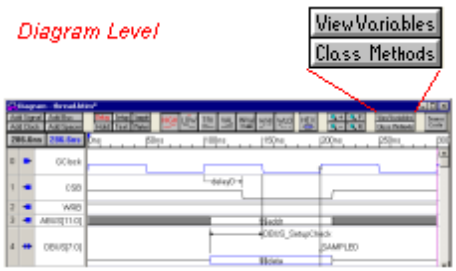
Chapter 5: Transaction Markers 70 describes how to add graphical Marker lines that can be used to wait for activity from the model under test and/or loop over a section of a diagram. Markers can also be used to call user-written HDL functions and tasks from within a diagram.



Add Variables and Class Methods:

Chapter 6: Classes and Variables 82 describes how to define variables that can be used by the Reactive Test Bench Generation option.

- Variables can be created using the **View Variables** button or directly inside certain objects like Samples, Delays, Setups, and Holds.
- Functions or Tasks can be embedded into the timing diagram by entering the code into the **Class Methods** dialog. This code can be called from an HDL Code Marker.



Copyright © 2014, SynaptiCAD Sales, Inc.

Organizing a Test Bench:

As your test benches become more and more complicated, it will become more difficult to model them using a single timing diagram. At some point, you may wish to upgrade to Test Bencher Pro. All of your timing diagrams will import into TestBencher Pro. Also the **Edit > Block Copy** can be used to copy sections of a large Reactive Test Bench diagram and then paste them into smaller reusable TestBencher transaction diagrams. We have included a few test bench designs in Chapter 10 that explore how to design test benches with both the reactive test bench option and with TestBencher Pro.

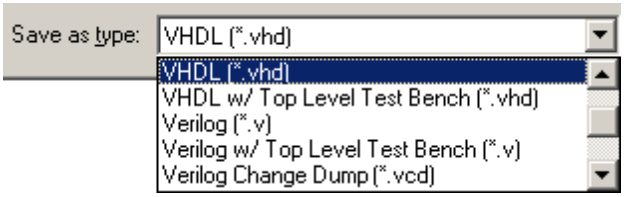
- Section 10.2 Waiting on Signal Transitions and Conditions 157
- Section 10.6: Testing a Counter Model 165

Step 3: Generate the Reactive Test Bench

For the Reactive Test Bench Option, the test bench generation step is just a simple export to a particular file format.

To export a timing diagram:

- Select the **Import/Export > Export Timing Diagrams As** menu option to open a special version of the *Save As* dialog which remembers the file type of the last file exported.
- Select a VHDL or Verilog export format from the **Save as type** box, enter a file name, and click Save button to generate the file.

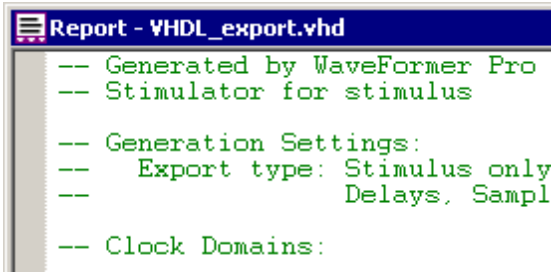


The plain VHDL or Verilog formats generates a stimulus module that can be included into a user-written top level test bench.

The **Top Level Test Bench** types instantiates the model in the *Project* window and the stimulus model within a top-level model. This top-level module can then be simulated without any additional setup.

The VCD format generates a stimulus file that can be read by other EDA tools and test equipment (not a test bench model).

- The generated file will open as a tab in the Report window so you can quickly see how changes in the diagram affect the generated code.

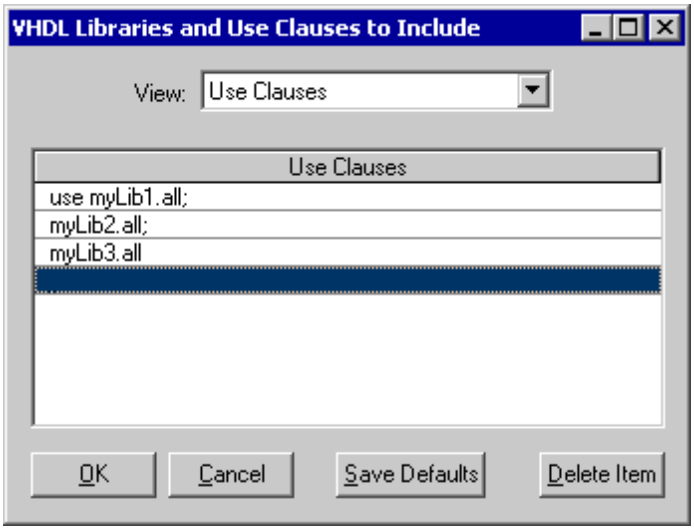


VHDL Library and Use Clause statement support:

If Libraries or Use Clauses need to be added to the generated model, then specify them in the *VHDL Libraries and Use Clauses to Include* dialog.

Copyright © 2014, SynaptiCAD Sales, Inc.

- Choose the **Options > VHDL Libraries and Use Clauses** menu to open the dialog. Changes made in this dialog will be applied to all diagrams that are exported to VHDL.
- From the **View** control select either **Use Clauses** or **VHDL Libraries** and enter the information on to a blank row.

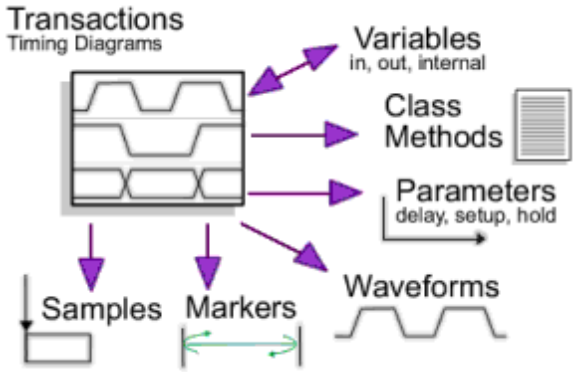


- If the 'USE' is omitted from a use clause, or the 'LIBRARY' from a library statement, WaveFormer will automatically add 'USE' or 'LIBRARY' before the clause in the source file for the diagram. If necessary, WaveFormer will also automatically add semicolons to the end of library includes and use clauses

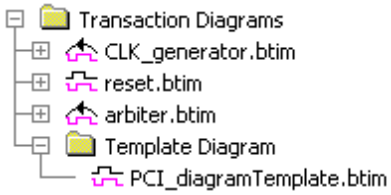
Copyright © 2014, SynaptiCAD Sales, Inc.

Chapter 2: Timing Transactor Basics

TestBencher uses graphical timing diagrams to generate reusable timing transactors (e.g., read cycle, write cycle, interrupt cycle) that can be called by a top-level test bench and applied to the model under test. Reactive Test Bench Option uses a single timing diagram to generate test benches. This chapter covers how to draw signals in a timing diagram, specify state values and clocking domains, verify sequences with sensitive edges, and other waveform related features. The next few chapters cover other graphical elements in the timing diagram such as samples 49, markers 70, delays, setups, and holds 61, and variables and class 82 methods used to define algorithmic functions.



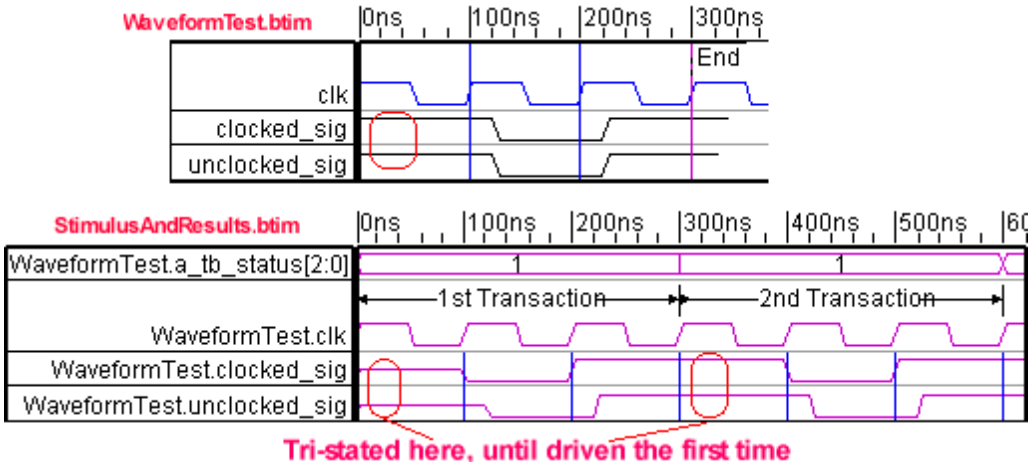
For TestBencher Pro users, the **Transaction Diagrams** folder in the *Project* window holds all of the transactors and the template diagram for the project. As shown in Step 3: Extract Ports into Template Diagram 14 and Step 4:Create Timing Transactor Diagrams 16, use this folder to create new diagrams, open diagrams for editing, and edit the transaction properties including the Master and Slave setting.



2.1 TestBencher Transactor Waveform Initilalization

In TestBencher Pro, it is important to remember that transactions do not automatically include an event at time zero and that code for signal stimulus is only generated at edge transitions. 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 (examine the red circles).

Copyright © 2014, SynaptiCAD Sales, Inc.

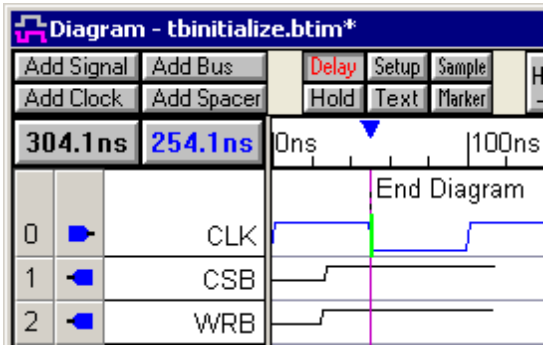


Specifying an Initialization State:

When signals require a specific initial state, there are two ways to do it: an initialization event and an initialization timing diagram.

- **An event** can be drawn at the beginning of the timing diagram to force the state initialization. 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 that initializes the signals CSB and WRB to a high state.



2.2 Drawing Waveforms

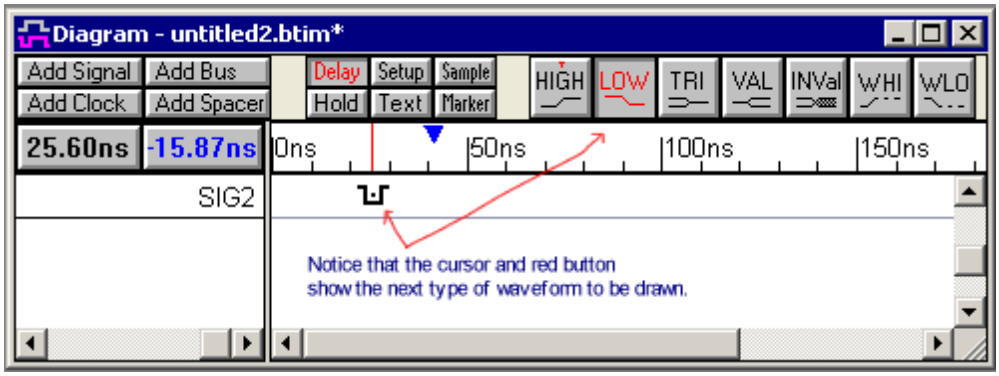
The timing diagram editor is always in drawing mode, so left clicking on a signal will draw a waveform. The red state button controls the type of waveform that is drawn (high, low, tri-state, valid, invalid, weak high, and weak low). The buttons toggle back an forth between two states, and the next state is indicated by the little red T on top. Click on the state buttons to set the toggle and next state.



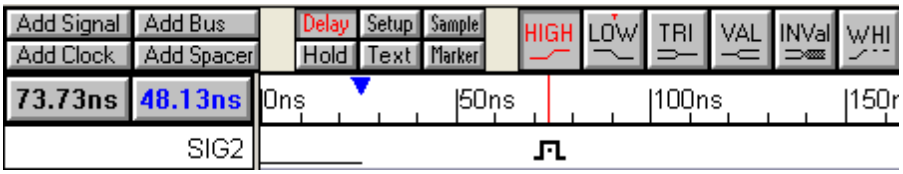
To draw the waveform of a signal:

- Place the mouse cursor inside the *Diagram* window at the same vertical row as the signal name. The red state button on the button bar determines the type of waveform drawn. The cursor shape also mirrors the red state button.

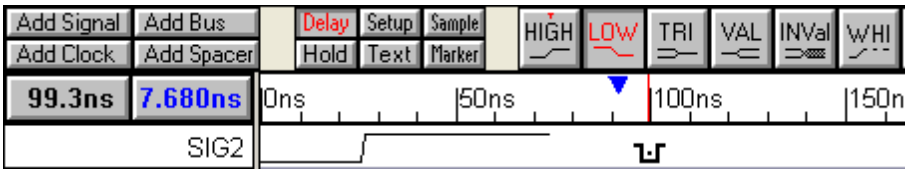
Copyright © 2014, SynaptiCAD Sales, Inc.



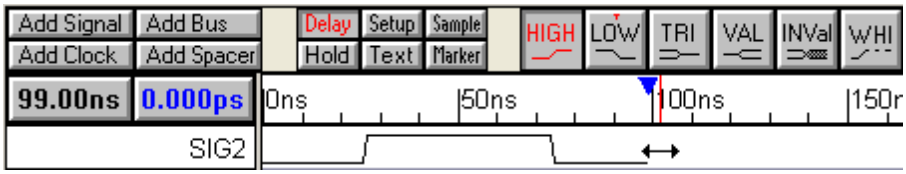
- Click the left mouse button. This draws a waveform from the end of the signal to the mouse cursor.



- Move the mouse to the right and click again to draw another segment.



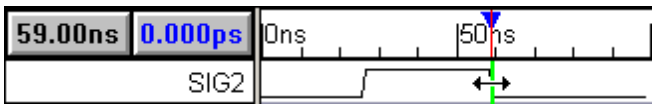
- Keep drawing from left to right across the diagram.



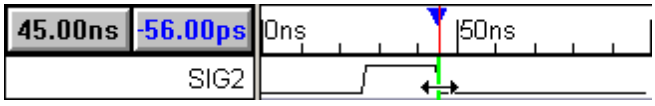
There are several mouse-based editing techniques used to modify existing waveforms. These techniques will only work on signals that are drawn. They will not work on generated signals like clocks and simulated signals that are covered in later chapters. Here are the three most popular techniques. More editing techniques and automated methods of generating waveforms are covered in the **Timing Diagram Editors Manual** Chapter 1: Signals and Waveforms.

1) Drag-and-Drop a Signal Transitions:

- To move one transition, click on the transition and drag it to the desired location.



- To move all of the transitions on one signal, hold down the **<1>** and **<2>** number keys while dragging. Holding down just the <1> key moves all the edges to the left, and the <2> key moves all the edges to the right.



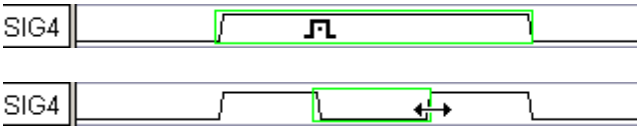
Copyright © 2014, SynaptiCAD Sales, Inc.

- To move transitions on different signals, first select the transitions by holding the **<CTRL>** while clicking on them. Then select and drag on the final selected edge to move all the selected transitions an equal distance in time.



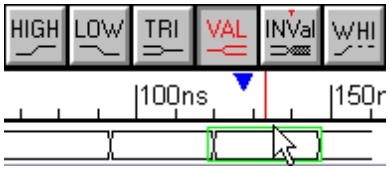
2) Click-and-Drag to insert a segment into a waveform:

- Inside of a segment, click and drag the cursor to insert a segment
- The inserted state is determined by the red state button.

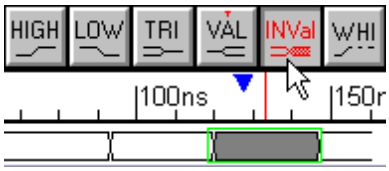


3) Change a segment's graphical state by selecting it and then pressing a state button:

- Click in the middle of the segment to select it (so that it has a green box around it).



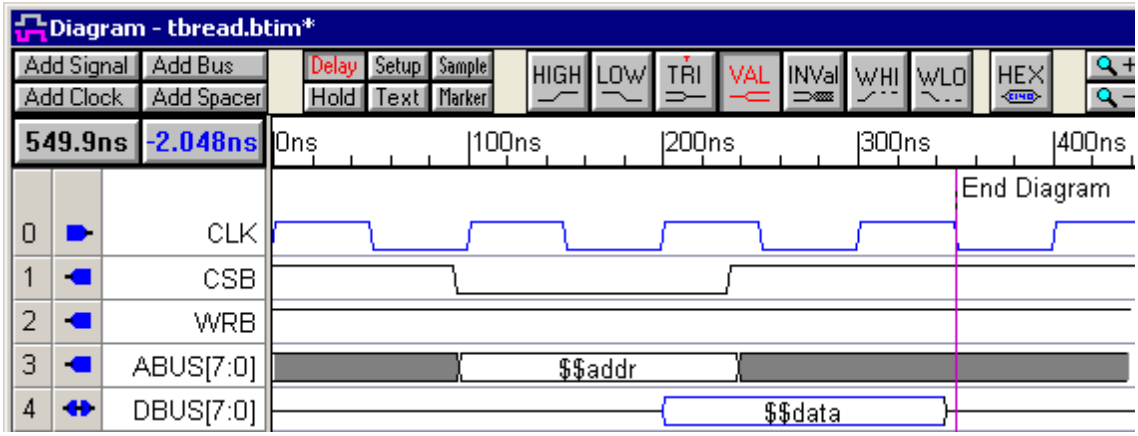
- Click on a state button to apply that graphical state to the segment. If you change a segment to the same level as an adjacent segment, the transition will turn red to preserve the edge data. This red transition can be deleted if necessary.



2.3 Waveform Colors and Bi-Directional signals

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 transactor and which are inputs to the transactor. In the diagram below, CLK is an input to the diagram, so its waveform is blue and the direction icon points to the right. The next three signals are outputs of the timing diagram (they will drive the MUT) and have black waveforms. The DBUS signal is bi-directional signal with mixed black and blue segments.

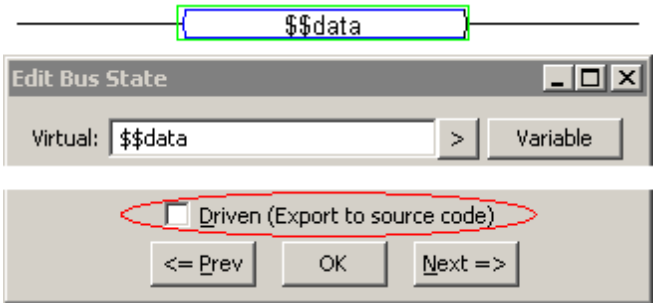
Copyright © 2014, SynaptiCAD Sales, Inc.



Setting Direction on Waveform Segments of a Bi-Directional Signal:

Each waveform segment on a bi-directional (**inout**) signal has a **Driven** property that determines whether the segment is an output or input of the testbench. By default, each segment has a direction of output so the waveform will draw as a black signal. If a segment is drawn using a tri-state graphical state, then the segment will not drive the MUT regardless of the Driven property setting. However, when the transactor needs to test the value coming back from the MUT, the expected waveform segments need to be inputs to the testbench. To change a segment to be an input segment (un-driven):

- Double-click on a waveform segment to open 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.

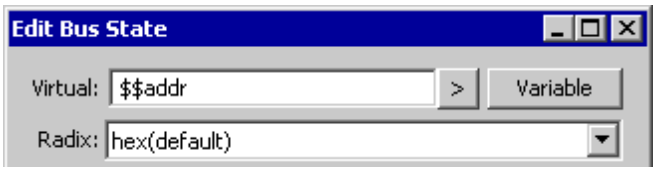
2.4 Waveform States with Variables and Conditions

The graphical state of a waveform segment controls the value of the segment in the generated test bench code, except for **valid** segments. 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 hardcode in a value, choose an existing variable in the timing diagram to drive the state, or in TestBencher define a state variable that can be passed into the transaction when it is applied. State values can also vary based on a conditional expression. If a waveform segment is drawn with a graphical state ***other than valid***, that graphical state will override any other state information entered through the *Edit Bus State* dialog.

Copyright © 2014, SynaptiCAD Sales, Inc.

Use the Edit Bus State dialog to edit the state of a valid signal segment:

- Double click on the segment of the signal to open the *Edit Bus State* dialog.

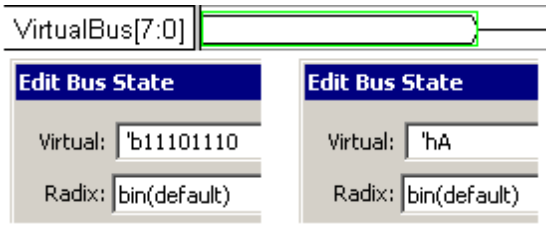


- The **Virtual** edit box accepts values, variables, and Boolean equations that meet the format shown in Appendix C: Language Independent Operators 182. For example, $$addr+@increment is an acceptable equation for the **Virtual** edit box.

Hardcode a state value:

Buses (multi-bit signals) use one of two methods to generate code. First, if a segment's value begins with a **'b** or **'h** then it is assumed that the number is a binary or hexadecimal number and the number will be translated to the appropriate VHDL or Verilog bus value (regardless of the radix of the signal). If the extended state value does not start with 'b or 'h, then the value is written out as it was entered, without any translation. This means that if a state has a value of ***10*** and a radix of ***Hexadecimal***, the effective value of the segment is ***16 decimal***. And if a state has a value of ***10*** with a radix of ***Binary***, the effective value of the segment is ***2 decimal***.

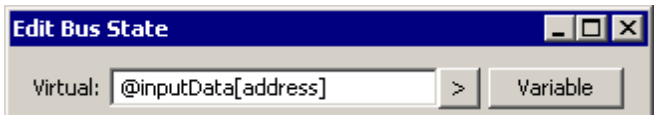
- Type the value in the **Virtual** edit box.
- Since this signal has a radix of binary, the `b11101110 is equivalent to entering just 11101110.
- If you entered 'hA into an 8-bit binary signal, the value would be converted to 00001010. The program automatically left pads missing bits.



Use a Diagram Level Variable to define the state during simulation:

Diagram level Variables are defined using the *Classes and Variables* dialog as shown in Section 6.1: Variables 82. Both TestBencher and the Reactive Test Bench 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. 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 the diagram.

- Either type in a variable name with an @ prefix to indicate it is a diagram level variable



- Or use the **> button** to display a list of variables, then pick one from the list to insert it into the Virtual box.

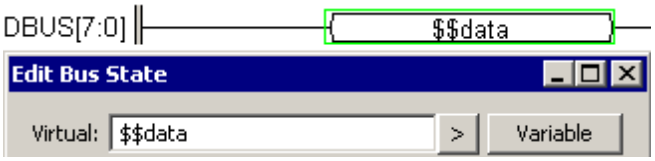
Use State Level Variable to define the state when the transaction is applied (Test Bencher Only):

TestBencher 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 *$$addr*. State variables are automatically added to the parameter list for the transaction call. The type and size for these variables are determined by the signal that is being driven. Each time the transaction is called, a new state value can be passed into the variable. The same state variable can also be used on several signals and the maximum size will be determined by the min and max of all of the signals used. For example, if $$addr appears on

Copyright © 2014, SynaptiCAD Sales, Inc.

SIG0[3:0] and SIG1[12:9], then the $$addr will have a size of [12:0].

- Type the variable name using a $$ prefix into the **Virtual** box.



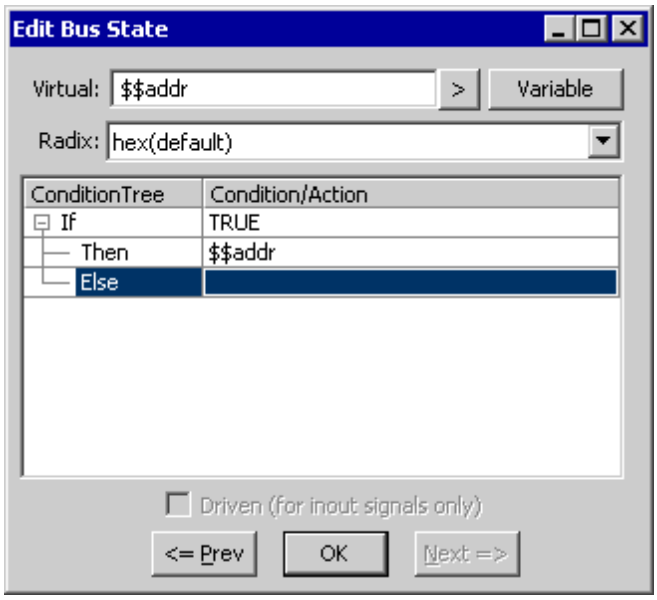
Do NOT use Parameter Variables to drive signal states:

Parameter based variables from delays, setups and holds should not be used to drive waveform states, because their type is fixed to hold ***time values,*** not ***state values***.

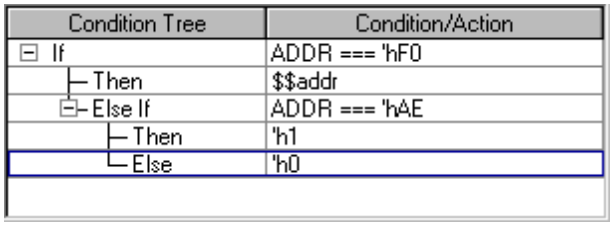
Conditionally Drive a state value:

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.

- Double-click on 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 transactor 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.

Copyright © 2014, SynaptiCAD Sales, Inc.

2.5 Simulated Signals

Signals can be defined using a Boolean equation of other signals in the diagram. These are quite useful for modeling glue logic in slave transactors. The embedded simulation engine properly simulates multi-bit signals, making it easy to model common interface logic such as configuration and mode registers. The simulator also uses true min/max timing models so that the resulting simulated signals can propagate uncertainty delay through the timing diagram. This feature is not supported in WaveFormer Lite with Reactive Test Bench Generation Option.

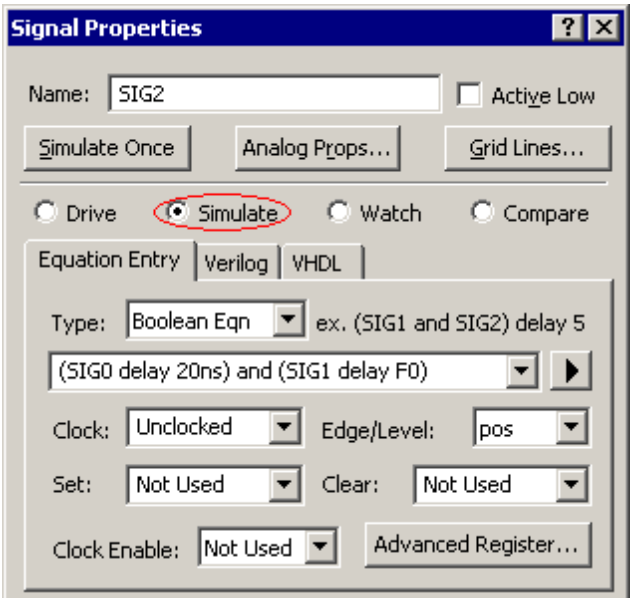
Code Generation for Simulated Signals:

Only signals that have a direction of **output** or **internal** will generate a Boolean and Registered logic code within the test bench.

Define a Signal's waveform using an Equation:

There is an entire chapter in the Timing Diagram Editors manual that explains in detail all of the requirements of the Simulated Signals feature. See Chapter 4 Simulated Signals. This is just a quick introduction.

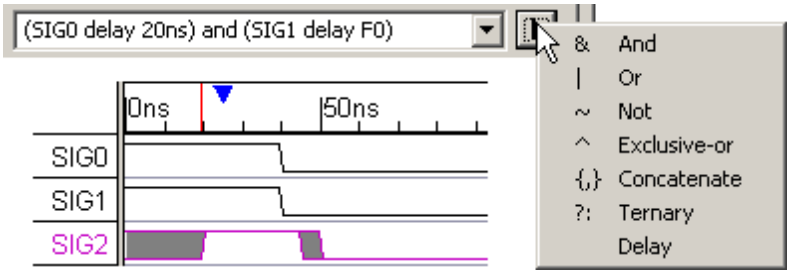
- Double click on a signal name to open the *Signal Properties* dialog.
- Select the **Simulate** radio button to allow the signal to be re-simulated each time an input signal is changed.
- Enter a Boolean equation and check any other attributes for the simulation.
- The signal will start to simulate as soon as an equation is entered (if you are in Auto Run and not Debug Run mode). Simulated signals waveforms are drawn in purple. If the waveform is grey there is an error in the equation, see the error section in Chapter 4 Simulated Signals of the Timing Diagram Editors manual.



Syntax

The Boolean Equation edit box accepts Boolean equations in VHDL, Verilog, and SynaptiCAD's enhanced equation syntax. All signal names are case sensitive.

- The quick fill box contains all of the SynaptiCAD operators.



Copyright © 2014, SynaptiCAD Sales, Inc.

- The **delay** operator takes a signal on the left and a time or parameter name on the right. If a parameter is entered then the equation will simulate using true min/max timing and puts grey uncertainty regions on the simulated signal.
- The **delay** operator can also be made to simulate min-only or max-only simulations by selecting the **Options > Diagram Simulation Preferences** menu to open the dialog and then choosing the appropriate setting from the **Timing Model** list.

2.6 Sensitive Edges - a Blocking Construct

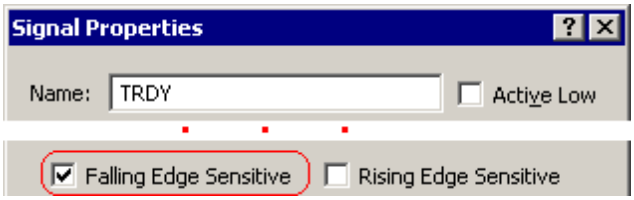
An input signal can be set to have sensitive edges, so that the test bench will wait on every falling or rising edge that the model under test generates. The sensitive edge controls are located in a signal's *Signal Properties* dialog. A signal with sensitive edges will be drawn with arrows on the sensitive edges. This sequence recognition is limited to single bit signals.

Alternatives to Sensitive Edges

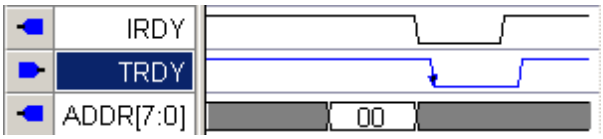
With Sensitive edges, there is no time-out feature, so the testbench can potentially hang if the model under test does not provide the edges as drawn. There are two other blocking constructs, samples and markers, that provide methods of waiting that are less demanding than the sensitive edge construct. See Section 3.3: Samples that Block Clock Domains 54 and Section 5.4 Wait Until Markers 74 for complete details on using these constructs to wait for events. Also, Section 10.2 Waiting for Signal Transitions or Conditions 157 has an example using all three blocking features.

Enable sensitive edges on a signal:

- Double-click the name of the signal to open the *Signal Properties* dialog.
- Check either **Rising Edge Sensitive** or **Falling Edge Sensitive** or both to indicate the blocking points for the testbench.



- Press the **OK** button to close the *Signal Properties* dialog. Notice the waveform edges now have arrows on the sensitive edges



The code generation for sensitive edges looks like the following:

```
VHDL:       -- Sensitive Falling Edge on signal: TRDY
              wait until falling_edge(TRDY) or (tb_DgmAborted);
Verilog:    // Sensitive Falling Edge on signal: TRDY
               @(negedge TRDY);
```

2.7 Controlling the Triggering Order of Parameters

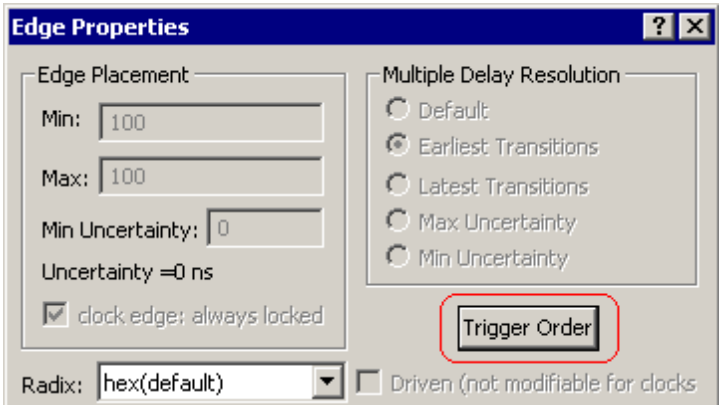
Edges on waveforms are responsible for triggering the markers and timing parameters that are attached to them. If more than one parameter or marker is attached to the same edge, then the triggering order can be set using the *Edge Properties* dialog. By default, the triggering order is the same as the order in which the objects were attached to the edge. 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. If a marker is relative to an edge, but not exactly on top of the edge, then order is based off of it's placement in the timing diagram and

Copyright © 2014, SynaptiCAD Sales, Inc.

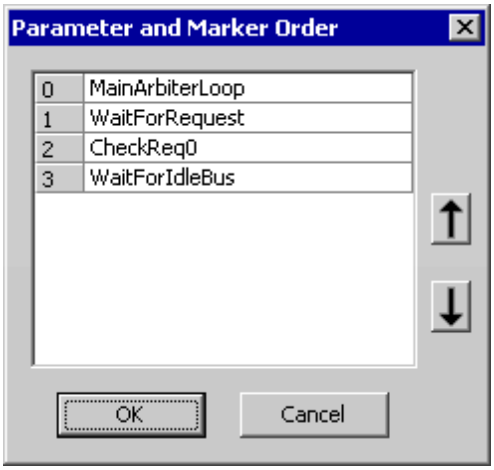
will not show up in the order dialog.

To order Parameters and Markers attached to the same edge:

- Double-click on 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.



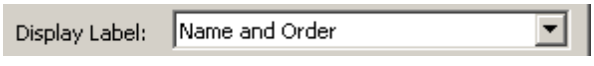
- Drag and drop the rows to arrange the parameters and markers in the desired order. You can also use the arrow buttons on the right side of the dialog to move selected items up or down.
- If you need to review the properties of an item before setting the order, you can double-click the name of the object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



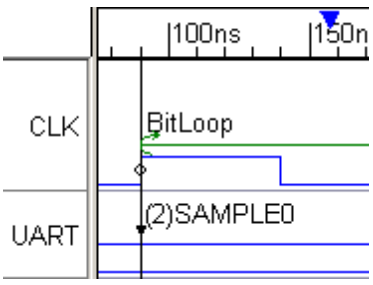
Displaying the order of parameter and markers in the timing diagram

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.

- Double click on the marker name or the parameter name to open the *Properties* dialog for that object.
- Choose **Name and Order** from the **Display Label** drop down.



- 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. In the image, the BitLoop marker will trigger first and then the SAMPLE0 will trigger second (the sample is within



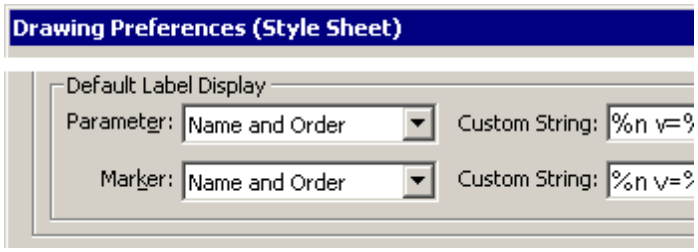
Copyright © 2014, SynaptiCAD Sales, Inc.

the loop).

Global Settings for Name and Order:

In addition to the individual parameter and marker settings, you can also specify the global default.

- Select the **Options > Drawing Preferences** menu to open the *Drawing Preferences* dialog.
- Select the **Name and Order** selection from either the Parameter or Marker *Default Label Display* boxes.



2.8 Adding Signals Manually

Most of the testbench signals will be automatically added by extracting the signal information from the model under test using the techniques that are discussed in Step 3: Extract MUT Ports into Template Diagram 14. 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.

Alternative: TestBencher Pro Sub-Projects

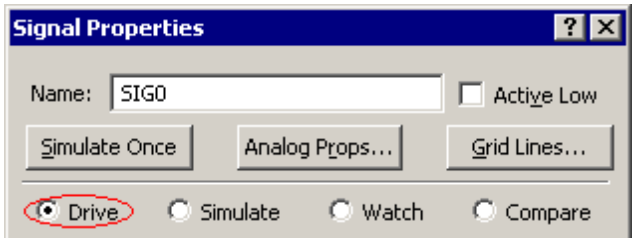
Since the generated bus-functional model provides stimulus and monitors simulation outputs of the MUT, the output signals of the bus-functional model have to match the signals that exist in the MUT. If the signals in the timing diagrams are named the same as in the MUT, then the matching will be automatic. If the signal names do not match, then you will have to create a sub-project and use the *Signal and Ports* dialog to define the signal mapping as covered in Section 9.1 Sub-Projects 137.

To add a Signal, Clock, Bus or Spacer:

- Press one of the **Add** buttons in the timing diagram editor to add a signal of that type. Spacers are just for making the timing diagram look nice and will not generate code.



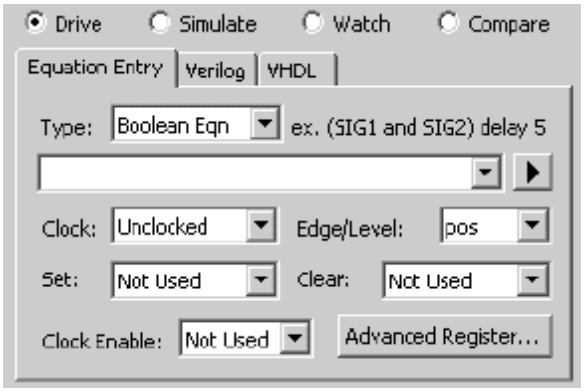
- Double click on the name of the new signal to open the *Signal Properties* dialog.
- For testbenches, the **Name** must exactly match the signal name in the MUT. If it is an internal signal, then the name can be any valid name supported by the generation language.



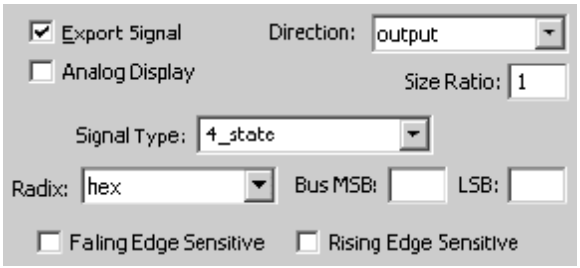
- **Drive** is the type of signal that can act as an input or output from the testbench. **Compare** signals are covered in the Timing Diagram Editors manual Chapter 9: Compare and Transaction Tracker Options. **Simulate** signals are used to model simple glue logic such as Boolean equations and registers. **Watch** signals are used by the BugHunter interface to display simulation results.

Copyright © 2014, SynaptiCAD Sales, Inc.

- Set the **Clock** and **Edge/Level** settings if the signal is to be a clocked signal. 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.
- You can also use a Boolean equation to describe the waveform of the signal as shown in the Timing Diagram Editor manual Section 2.5: Simulated Signals 39.

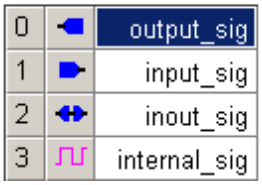


- Edit the **Signal Type**. This box uses SynaptiCAD's language independent types (that convert to the closest VHDL or Verilog type) and you can also type in VHDL user defined types.
- Edit the signal size using the **MSB** and **LSB** boxes.



- Edit the **Direction** using the drop down list box. The following directions are available:
  - **Output** indicates that the signal is an output from the diagram transactor (i.e. an input to the model under test).





  - **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 2.3 Waveform Colors and Bi-directional Signals 35). 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 transactor. These are typically used to compute intermediate values that don't directly drive the model under test.
- 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, clock domain, and language type for new signals can be set from the *Diagram Settings* dialog (see Section 9.8: Diagram Settings Dialog - Overview 148 for more information).

Copyright © 2014, SynaptiCAD Sales, Inc.

2.9 Transactor Clock Domains and Waveform Code Generation

Inside each timing diagram there may be **one *unclocked*** process and **several *clocked*** processes. A clocked 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 a timing 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 unclocked domain wouldn't get placed into this 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.

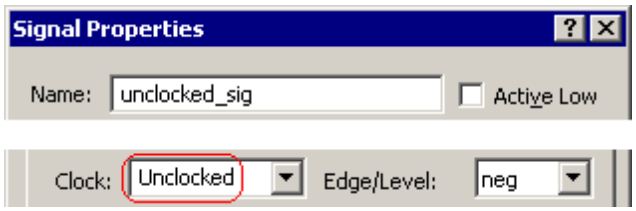
Overview of Graphical constructs and their clock domains:

Construct Type	Clocking Domain
Signals	Clock and Edge of signal (Signal Properties 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.
Delays attached to edge	If not unclocked, then Clock and Edge in Delay dialog. Otherwise, the starting edge of the delay sets the clock domain.
Setups	Signal and edge that is pointed to by the Setup
Holds	Signal and edge that is pointed to by the Hold
Markers attached to edge	The relative edge sets the clock domain
Markers attached to time	Unclocked

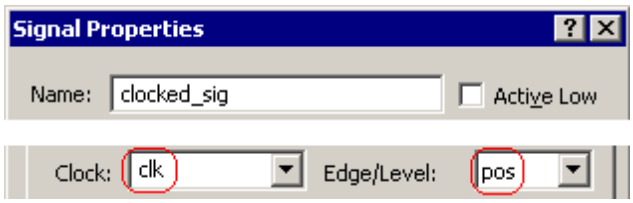
Waveform Code Generation:

The code that is generated for a waveform depends on three things: how it is drawn, the clocking domain, and the **Default Delay After Clock Edge** control in the *Diagram Settings* dialog.

- For **unclocked signals**, the waveform events will be generated at the times they are drawn.
- To check the clocking domain of a signal, double click on the signal name to open the *Signal Properties* dialog. If the **Clock** property is set to ***Unclocked***, then it is unclocked.



- To make a signal **clocked**, set the **Clock** and the **Edge/Level** setting in the *Signal Properties* dialog (double click on the signal name to open the dialog). The **Clock** can be any signal or clock in the diagram.



- For the **Reactive Test Bench Option**, a clocked signal will generate code such that the

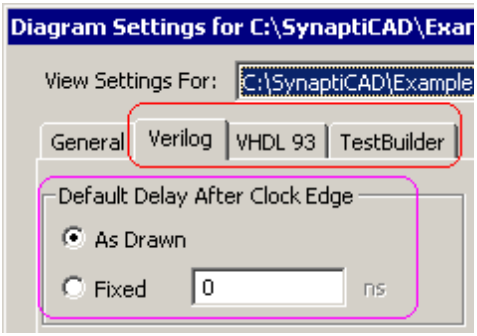
Copyright © 2014, SynaptiCAD Sales, Inc.

clocked signal's edges will sync up to the active edge on the "clock" signal.

- For **Test Bencher Pro**, a clocked signal's code also depends on the controls in the *Diagram Settings* dialog. Right click on the diagram name in the Project window and choose **Diagram Settings** from the menu.



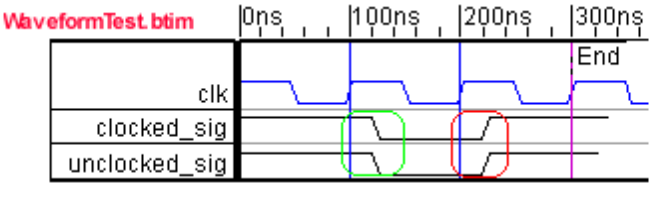
- Choose the language tab of your generation language.
- **Fixed 0ns**, means that the edges of a clocked signal will sync up to the active edge of the clock. If there is a delay, like **10ns**, then the clocked signal's edges will be driven 10 ns after the active clock edge.

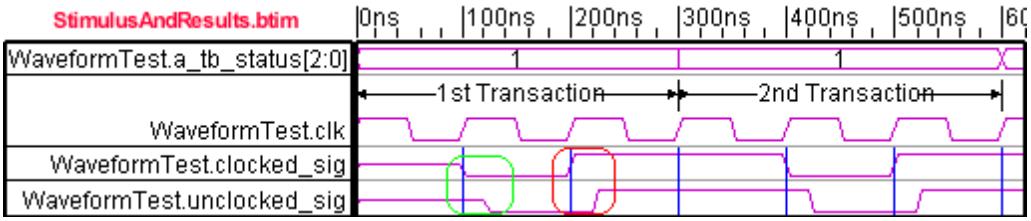


- **As Drawn** will delay the edges of the clocked signal relative to the the clocking edge based on the distance drawn. For example, if an edge is 5 ns after it's clocking edge, the code will wait for the clocking edge, then further delay for 5 ns before driving the signal edge. Below is the code generated by such a signal:

```
// Synchronize to clocking signal if necessary.
    @(posedge CLK0);
    #5;
    tbench.SIG0_driver <= 1'b0;
```

Below is an example of a timing diagram **WaveformTest** with both a clocked and an unclocked signal. The **Default Delay After Clock Edge** setting for the diagram is **Fixed 0ns**. The **Stimulus and Results** diagram shows the behavior of the test bench during simulation. Notice that the edges on the ***clocked_sig*** have moved to the positive edge of the ***clk*** signal in the simulated results, and the edges on the ***unclocked_sig*** generate code as they are drawn in the original transactor diagram.



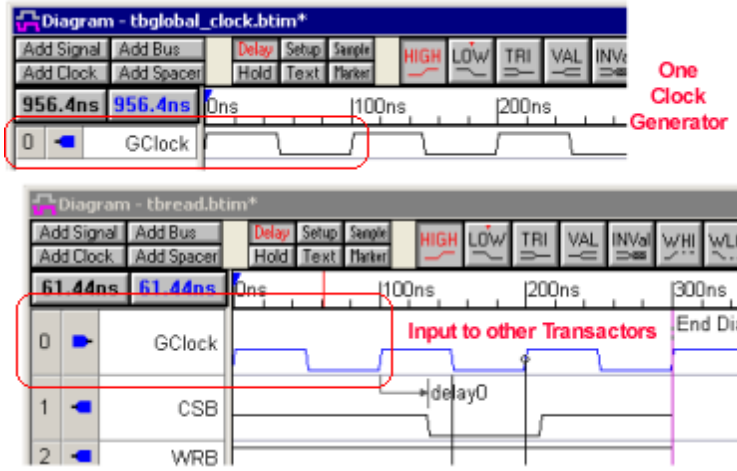


Output Clocks (Clock generators)

When creating a clocked test bench with TestBencher, 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

Copyright © 2014, SynaptiCAD Sales, Inc.

an input clock. This makes it easier to synchronize the transactions during simulation. If you choose a system clock when during the project creation, then TestBencher will create a diagram called ***SysClock*_generator.btim** to generate the clock, and also add an input clock to the template diagram (see Step 1: Create a New Project 8).



Each output clock has its own process that generates the clock during a simulation. This clocking process is in addition to any unclocked 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.

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** 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 (see 2.6 Sensitive Edges - a Blocking Construct 40).
- A Sample that has the **blocking** setting checked (see Section 3.3: Samples that Block Clock Domains 54) will block its clock domain until the sample completely finishes, including execution of its *then* or *else* action.
- And a **Wait Until Marker** 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 (see Section 5.4 Wait Until Markers 74).

2.10 Debugging Transactor Clock Domains

The most common problem with transactors is the accidental creation of a clock domain by putting a graphical construct such as a sample on the wrong clock edge while the rest of the transactor's objects are in another clock domain. Below are some ways of quickly checking on the clock domains.

Quick Check on the number of Clock Domains for each transactor

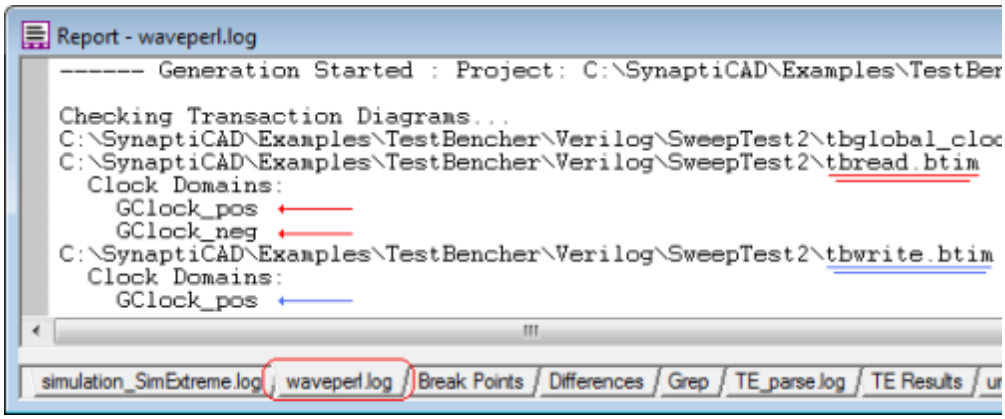
The waveperl.log file shows the names of the clock domains for each transactor in a project in one location. This display is useful to verify that transactors in a project are primarily operating within the same clock domain. Some additional clock domains may be employed by a particular transactor for checking purposes, but most of the transactors in a project TLM typically share a common clocking domain for synchronizing the application of stimulus waveforms to the models under test.

Copyright © 2014, SynaptiCAD Sales, Inc.

- Press the **Make TB** button to generate transactor code for each transaction timing diagram.



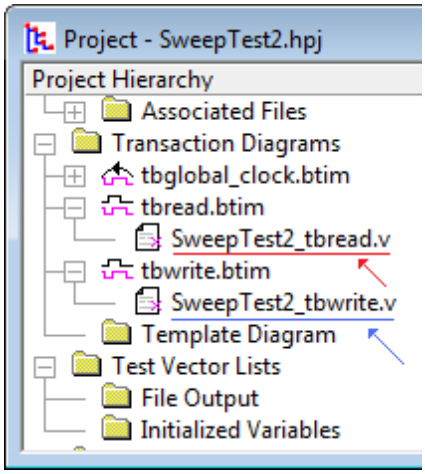
- Look in the **waveperl.log tab** in the ***Report*** window to see the clock domains for each timing diagram. If you cannot see the Report window, choose the **Window > Report** menu to bring it to the front.



Verifying Graphical Constructs are in proper Clock Domains

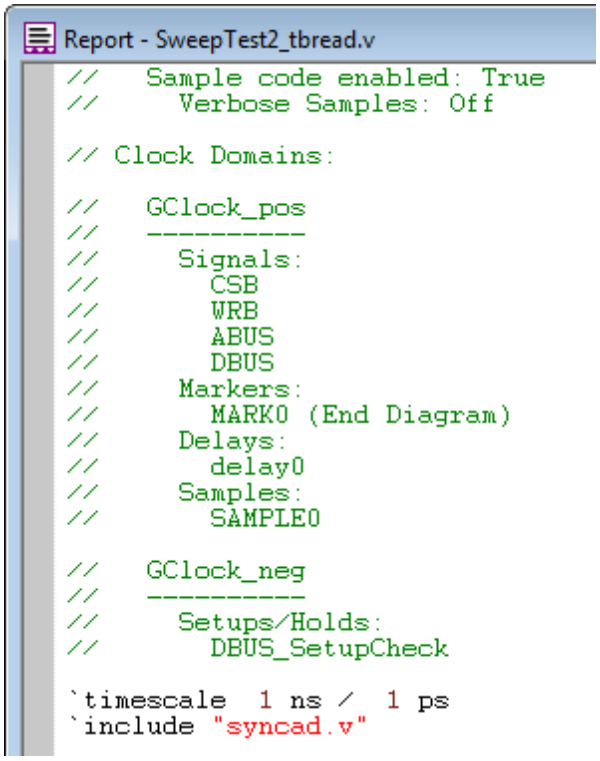
The comments at the top of each transactor's code quickly shows which signals and other graphical constructs are part of each clock domain. It's particularly important to verify graphical constructs are in the proper clock domain when time marker loops are used in a transactor (see below for more details).

- After generating the transactor code using the **Make TB** button as shown above, you can find the transactor code in the ***Project*** window under the ***Transaction Diagrams*** folder.
- Double click on code file for each timing diagram to see the generated code in the **Report** window.



Copyright © 2014, SynaptiCAD Sales, Inc.

- Read through the comments at the top of the transaction to see the clock domains and the objects they contain.
- If you have a Loop within a transactor, then all graphical objects that occur within the loop interval should typically be in the same clock domain. For example, one common mistake would be to create a loop in a clock domain called GClock_pos, but trigger a sample or a setup contained in the loop's time interval on the negative edge of GClock (thereby placing the sample or setup in the GClock_neg clock domain). In this case, the sample or setup would only happen once rather than every iteration of the loop (which is probably not the intended operation).



Copyright © 2014, SynaptiCAD Sales, Inc.

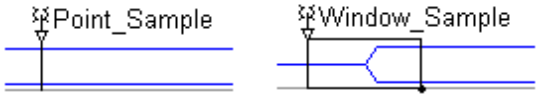
Chapter 3: Transaction Samples

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. Samples generate checking code within a transaction using either temporal expressions or procedural code that produces the same functionality as a complex temporal expression.

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.

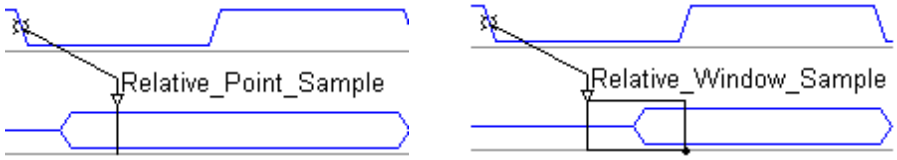
Monitoring Time

Samples that monitor a signal at a specific time are called ***Point Samples***. 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 during a time window. 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 3.2 51.



Triggering Process

Point and Window Samples can either be 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 event, the sample will trigger as soon as the transition occurs. If the triggering event does not occur during a transaction, the sample will not execute.



Sample Variables and Files

Samples generates two diagram-level variables, ***sampleName*_Flag** and ***sampleName***, that can be accessed by other graphical elements in the diagram (Section 3.5 58). For TestBencher, sampled values can also be written out to a file (Section 3.6 59).

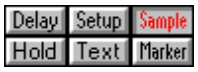
3.1 Adding a Sample

How you draw the sample determines whether it will be an Absolute Sample (triggered by time) or a Relative Sample (triggered by an event). 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*

Copyright © 2014, SynaptiCAD Sales, Inc.

Generation Options dialog discussed in Section 3.2 51.

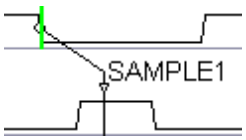
Draw a Sample and set the triggering event or time:



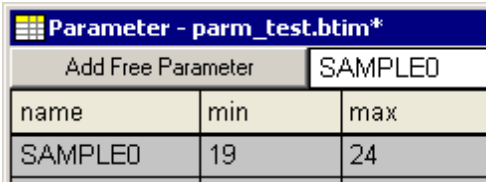
- Press the **Sample** button so that right mouse clicks will add samples to the diagram.



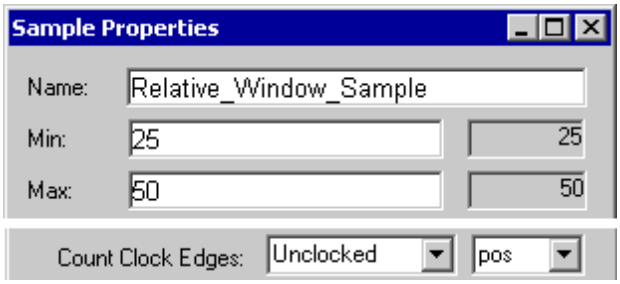
- To attach a sample to a time, make sure no edges are selected, then right click on a waveform to be tested. This will add a sample.



- To attach a sample to an edge, **left click** on the control edge to select it, then **right click** on the waveform to be sampled. This will add a sample between the control edge and the waveform to be tested.
- The exact time at which the sample is placed can be changed using the *Samples Properties* dialog discussed in the next step.
- When a sample is added to the diagram, it is also added to the Parameter Window.



Use the Samples Properties Dialog to set the monitoring time and properties of the sample:



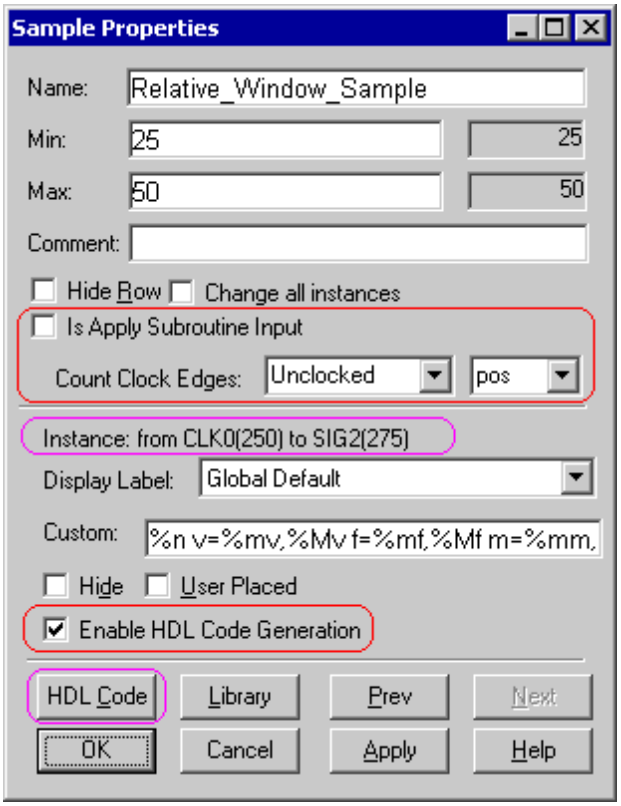
- Double-click the sample name to open the *Sample Properties* dialog.
- Samples can be cycle-based instead of time-based. The **Count Clock Edges** settings allow a clocking signal and edge to be specified for the sample, this setting also defines whether the **Min** and **Max** boxes are accepting times or clock cycles.



- 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.

Copyright © 2014, SynaptiCAD Sales, Inc.

- For TestBencher Pro, the **Is Apply Subroutine Input**, if checked, 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.
- The **Instance** describes the triggering event for the sample, which is either the time for an Absolute sample, or the triggering signal and time offset for a Relative sample. The triggering events are set by how the sample is drawn.
- 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 3.2: Sample Conditions and Actions 51.



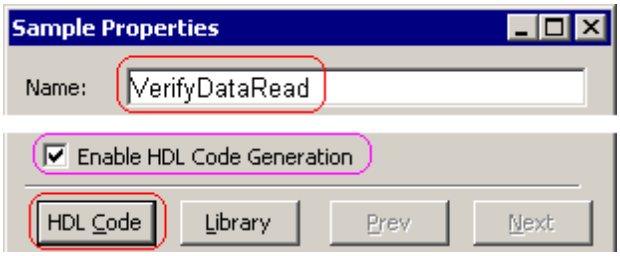
The *Sample Properties* dialog also has many settings that control how the sample is displayed in the timing diagram. Since Samples are parameters (just like delays, setups, and holds) most all of the parameter editing techniques also work for samples. These features are covered in the **Timing Diagram Editor** on-line help in Chapter 5: 5.3 Display Settings for Parameters, 5.4 Drawing and Editing Parameters, 5.5 Hiding Parameters, and 5.6 Parameter Window.

3.2 Sample Checking 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.

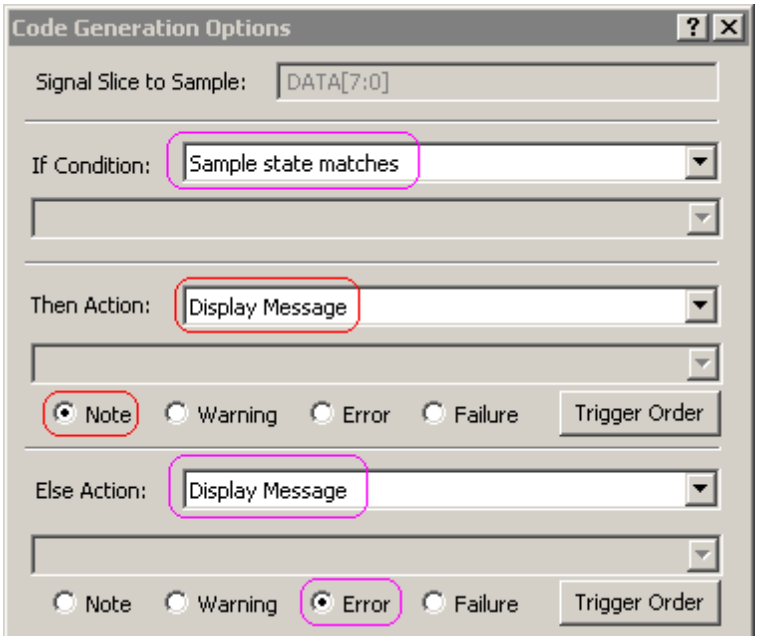
To define the condition and actions of a sample:

- Double-click on the Sample name to open the *Sample Properties* dialog.
- Verify that **Enable HDL Code Generation** is checked, so that the code can be generated for the sample.
- Press the **HDL Code** button to open the *Code Generation Options* dialog.



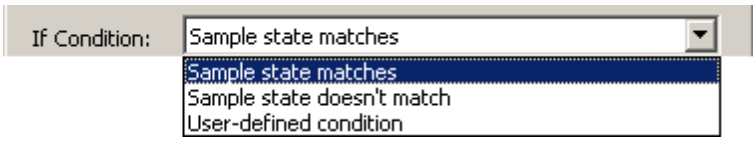
Copyright © 2014, SynaptiCAD Sales, Inc.

- The code for a Sample is modeled like an if-then-else statement. If you choose **User Defined** for the ***if condition*** or the ***actions***, then the edit box below will become active and allow you to enter code. The code will usually either call a class method that has been defined for the transaction (Section 6.1: Variables 82) or is a short HDL expression, like ***SAMPLE0_process_Flag and SAMPLE1_process_Flag***, that makes use of the internally generated sample variables (Section 3.5: Using Sample Variables 58).



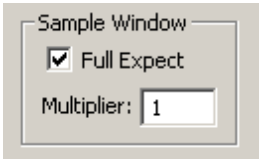
The If Condition:

- The **If Condition** controls what the sample checks for.



- For Point Samples, the test bench will compare the drawn state of the expected signal to the waveform that comes back from the MUT.
  - ***Sample state matches:*** If the monitored value matches the expected value (as drawn in the timing diagram), 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 (as drawn in the timing diagram, the *Else Action* will be taken otherwise the *Then Action* will occur.

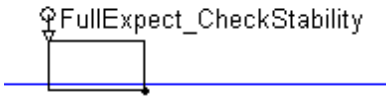
- For Windowed samples, the **Full Expect** box controls when the sample will trigger its action. If **Full Expect** is checked, then it will wait until the end of the window to trigger the action, and this is shown graphically by a dot at the end of the window. If unchecked, the sample is a Simple Sample and the the action will trigger any time during the window when the "if condition" becomes true, and this is indicated by the dot in the middle of the window. If a ***sample is clocked***, then the value of the waveform will be sampled at each clock edge that occurs within the sample's window.



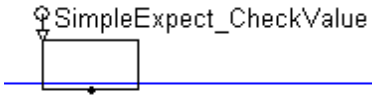
If a sample window is drawn over a stable section of waveform:

Copyright © 2014, SynaptiCAD Sales, Inc.

- A Full Expect sample will require the MUT waveform to be stable for the entire windowed time in order for the condition "sample state matches" to be true.

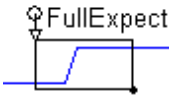


- A simple sample will be true if at any time during the window the MUT waveform goes to the value of the drawn waveform.

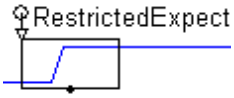


If a sample window is drawn over a section of waveform with one or more transitions:

- For a Full Expect sample, all transitions must match and the MUT waveform cannot have any more transitions than the drawn signal.



- A simple Sample will trigger true as soon as the MUT has produced the matching transitions in the same order. If an unexpected transition occurs, the condition will immediately fail and execute the *Else* action. If not enough transitions are detected, the sample will time-out at the end of the window and the *Else* action will be executed.

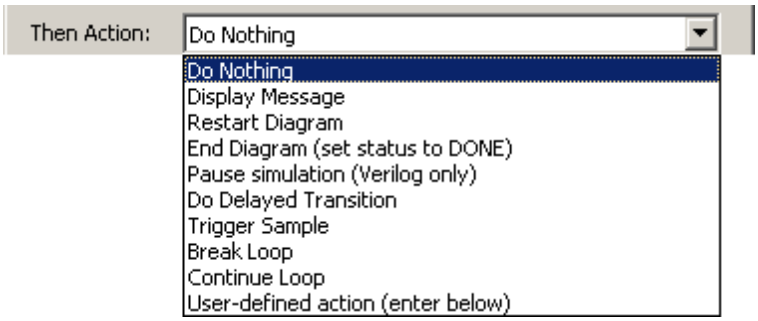


Extending the sample window with a multiplier

- The **Multiplier** property extends the window of time for a sample. The difference between the **min** and **max** values (in the Sample Properties dialog) 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.

The Actions:

- The **Then Action** and **Else Action** 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 (via $stop command).
  - ***Do Delayed Transition:*** Creates a delayed state transition (see Section 3.4: Samples

Copyright © 2014, SynaptiCAD Sales, Inc.

Trigger Delayed Transition or Another Sample 56) that triggers based on the results of the *If Condition*.

- ***Trigger Sample:*** creates a triggered sample (see (see Section 3.4: Samples Trigger Delayed Transition or Another Sample 56) 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 3.5: Using Sample Variables 58 for more information.

- The level setting effects the beginning of the message that will be sent to the simulation log file.

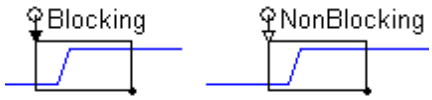


- The **Trigger Order** button opens a dialog that lets you rearrange the order in which markers and samples triggered off the same edge will begin to execute.

The **Blocking** control is described in Section 3.3 Samples that Block Clock Domains 54. The **Store Sampled Value As Subroutine Output** is covered in Section 3.5 Using Sample Variables 58. And the **Enable Variable** control is covered in Section 3.6 Storing Sample Values in User Defined Variables 59. The default actions for the if-then-else boxes are set in the *Diagram Settings* dialog (Section 9.8: Diagram Settings dialog 148)

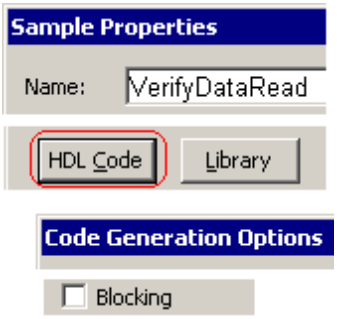
3.3 Samples that Block Clock Domains

A 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 completes. If **Blocking** is disabled, the other graphical elements will continue to function regardless of whether the sample condition is satisfied. Blocking samples are drawn with a solid arrowhead, and non-blocking samples have a hollow arrowhead.



Set the Blocking Control:

- Double-click on the Sample name to open the *Sample Properties* dialog.
- Press the **HDL Code** button to open the *Code Generation Options* dialog.
- Check the **Blocking** box at the bottom of the dialog.



Copyright © 2014, SynaptiCAD Sales, Inc.

What gets Blocked:

Samples block other graphical objects that are triggered from the same clocking domain. If a blocking sample is triggered off the rising edge of a clock signal, then the sample can block all other elements that are also attached to the same positive edge. The sample will block until the "if condition" triggers. If the clock is a system clock for other signals in the diagram, even the driving waveforms will be blocked until the the sample finishes executing. Elements that are in a different process/clock domain, continue to run in parallel. The source code for a timing diagram will explicitly state each object's clocking domain. To view the source code:

- Save the timing diagram to generate new code.
- Then Press the **Source Code** button to open an editor and view the code.



- In the top of the generated source code file, there is a **Clock Domain** comment section which lists the triggering processes for the diagram.
- In this code, SAMPLE0 can block all of the signals, markers, and delays that are triggered from the positive edge of GClock and that occur after SAMPLE0 triggers. However, SAMPLE0 cannot block DBUS_SetupCheck, because that construct is triggered from the negative edge of the clock.

```
// Clock Domains:

//   GClock_pos
//   ----------
//     Signals:
//       CSB
//       WRB
//       ABUS
//       DBUS
//     Markers:
//       MARK0 (End Diagram)
//     Delays:
//       delay0
//     Samples:
//       SAMPLE0

//   GClock_neg
//   ----------
//     Setups/Holds:
//       DBUS_SetupCheck
```

Code Generation and Clocking domains for 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.

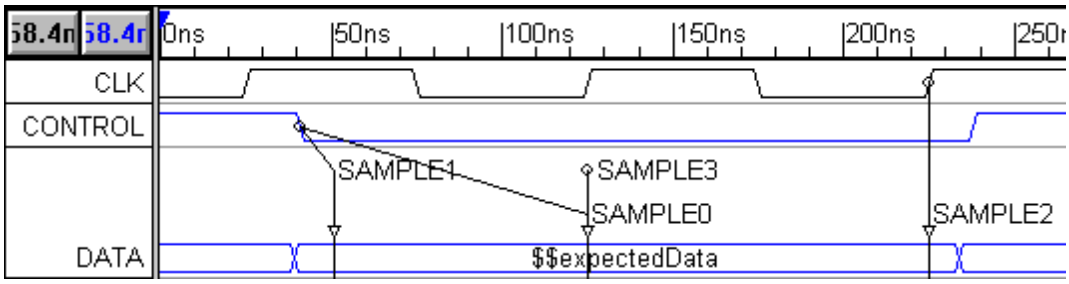
The sequence process that triggers a sample is determined from the combination of the triggering edge and the *Samples Properties* dialog ***clock*** and ***edge type*** settings.

Triggering Edge	Clock and Edge Type	Triggering Sequence
Time based trigger	ignored	Unclocked Sequence
Attached to an edge	Unclocked or matches triggering edge	Triggering edge's sequence
Attached to an edge	Different than triggering edge	Clock sequence from dialog with a level sensitive check on

Copyright © 2014, SynaptiCAD Sales, Inc.

		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 positive clock edge.
  - **SAMPLE0**: at second positive 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 immediately when the CONTROL signal goes negative (compare this to the behavior of SAMPLE0 above, which triggers synchronously with the clock).
- **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 after the beginning of the transaction.

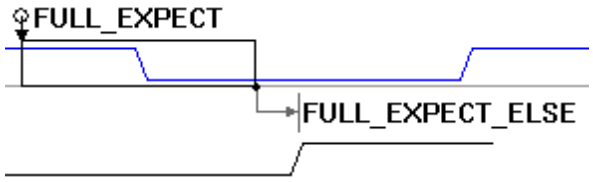
More examples and other Blocking Constructs

Section 10.2: Waiting for Signal Transitions or Conditions 157 shows an example using all three blocking features: samples, Wait Until Markers, and Sensitive Edges.

3.4 Samples Triggering a Delayed Transition or Another Sample

Samples can be used to trigger delayed state transitions or other samples. Chains of samples can be built up to test for complex patterns in the simulation data. When a sample in the chain triggers its ***then*** *or* ***else actions***, the next sample/delay in the chain will execute.

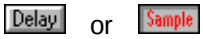
If a sample is triggering a delay, then that sample conditionally controls the signal transition. This is especially useful if several conditions must be met prior to a transition on a control signal. An alternate, non-graphical method for conditionally triggering transitions using conditional delays is discussed in Section 4.1: Delays 61.



Copyright © 2014, SynaptiCAD Sales, Inc.

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.

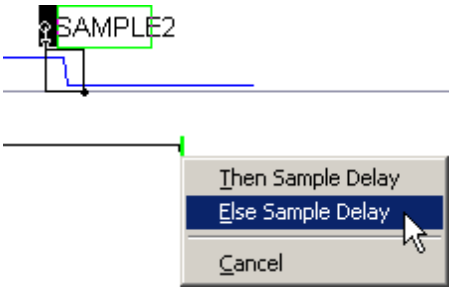
Method 1: (Recommended) Add a Delay or Sample to relative to a Sample

- Click the **Delay** button or the **Sample** button so that right clicks will create a parameter of the desired type. 

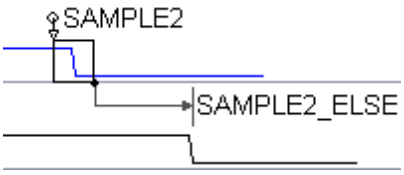
- Click on the sample name to select the sample that will trigger the new parameter.



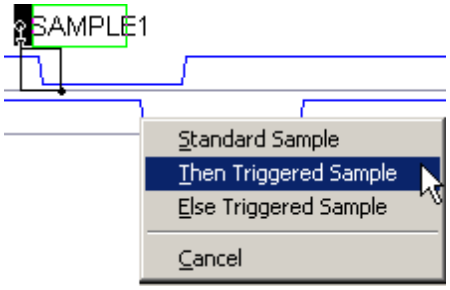
- To **add a delay**, ***right-click*** on the state transition that you want to be conditionally delayed. This will open a context menu. Choose either **Then Sample Delay** or **Else Sample Delay** to create the new conditional delay.



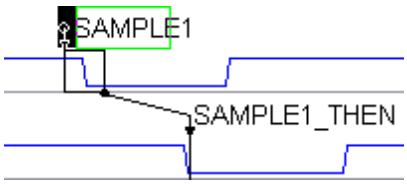
  - The delay will be added from the end of the sample to the edge that you right clicked on. It will be named after the sample and the action that triggers it.



- To **add a sample**, 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.



  - The sample will be added between the first sample and the new waveform



- The new sample or delay can be edited just like regular samples and delays, by double clicking on the name to open the *Properties* dialog.

Method 2: Using the Code Generation Options dialog to add a triggered delay or sample.

- Double-click the name of a sample to open the *Sample Properties* dialog.

Copyright © 2014, SynaptiCAD Sales, Inc.

- 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.

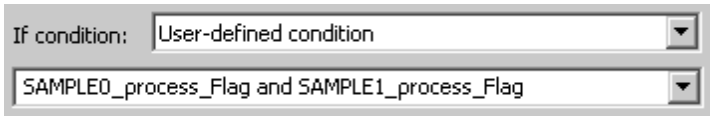
The delay ending position can be moved to other signal transitions by selecting the delay then dragging and dropping the right handle of the delay to the new transition. Triggered samples can be edited just like regular samples.

3.5 Using Sample Variables

Two sample variables are automatically generated for each sample: ***SampleName*_process_Flag** and ***SampleName***. The ***SampleName*_process_Flag** variable is a Boolean flag that indicates whether the sample condition was true or false. ***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.

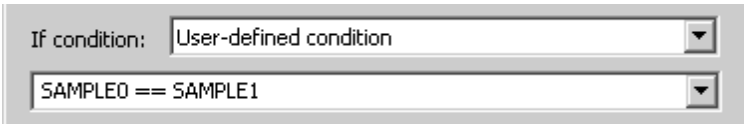
Using a Sample's Flag condition to trigger the "if condition" of a subsequent sample:

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_process_Flag and SAMPLE1_process_Flag***.



Using a Sample's Value in the Diagram

You can also use a sample's value variables 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.



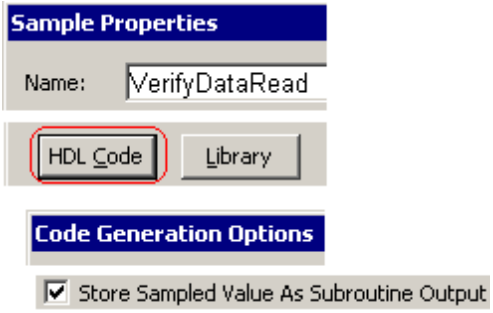
The size of the sample value will depend on the size of the signal that it is sampling. So for the above equation to work, the types of the signals sampled by SAMPLE0 and SAMPLE1 must be the same, or you will get a type mismatch error when you compile your test bench.

Copyright © 2014, SynaptiCAD Sales, Inc.

TestBencher can export sample variables as outputs from the timing diagram:

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.

- Double-click on the Sample name to open the *Sample Properties* dialog.
- Press the **HDL Code** button to open the *Code Generation Options* dialog.
- Check the **Store Sampled Value as a Subroutine Output** box at the bottom of the dialog.



In TestBencher, 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:

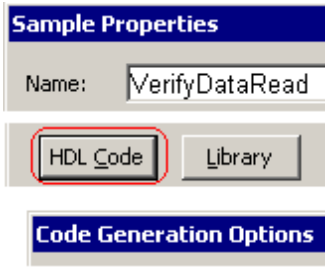
- In Verilog, TestBencher will automatically create a variable in the top-level project that contains the diagram. The variable will be named ***transactionName_sampleName*** and at the end of the transaction the sample value will be passed out to this variable.
- In all the other languages, you must 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.

3.6 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 complex data structures. In TestBencher, the sampled value can also 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.

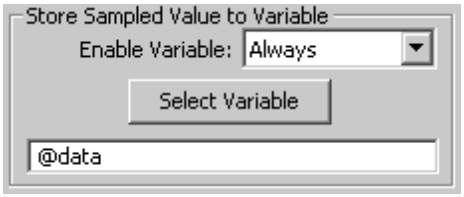
To store a sample value in a user defined variable:

- Create a diagram level variable as shown in Section 6.1: Variables 82. You must do this first before you can make the sample store its value in the 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.

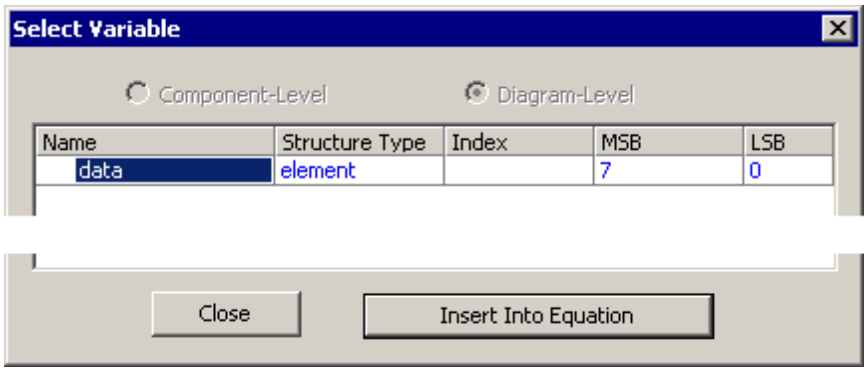


Copyright © 2014, SynaptiCAD Sales, Inc.

- Select the desired enable option from the **Enable Variable** dropdown, to specify the condition under which the sampled value should be stored in the variable. **Always**, **Never**, **Then**, and **Else** are the possible options. The Then and Else options specify that the data will be stored only if the *Then Action* or *Else Action* is executed, respectively.



- Press the **Select Variable** button to open the *Select Variable* dialog.
- Select a variable from the list and press the **Insert Into Equation** button to copy the variable into the *Code Generation Options* dialog.



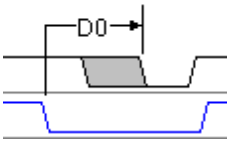
Copyright © 2014, SynaptiCAD Sales, Inc.

Chapter 4: 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. This chapter will cover delays, setups, and holds (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 TestBencher Pro and Reactive Test Bench timing diagrams, so they are covered separately in Chapter 3: Transaction Samples 49.

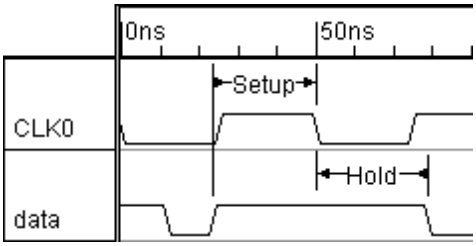
Delays

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. The signal properties can also be used to create a continuous check between two signals.



4.1 Delays

A delay specifies a fixed time or a number of clocking signal edges between two signal transitions. Delays can also conditionally drive their second edge. In TestBencher, the value for the delay time can be passed to the delay code during the simulation. One use of this is to perform a sweep test, where the delay value is progressively increased until a circuit failure occurs.

Delays create two diagram level variables

For each delay in a diagram, two variables are created in the transactor, ***delayname*_min** and ***delayname*_max,** that can be used in other parameters and in user-defined HDL code statements.

Only Delays triggered by Input signals generate edge-triggering code

Delays with trigger edges on input (blue) signals that force an output (black) signal will generate transactor code that may conditionally drive the output signal's edge. Also, if a delay is part of a chain

Copyright © 2014, SynaptiCAD Sales, Inc.

of delays that has its base delay triggered from an input signal, then the rest of the delays in the chain will generate test bench code. On the other hand, if a delay is put between two output signals and the delay is not part of a chain of delays, then the delay will be evaluated during code generation and the signal transition times will just be hard-coded in the transactor.

Delay Parameter's Clock Domain

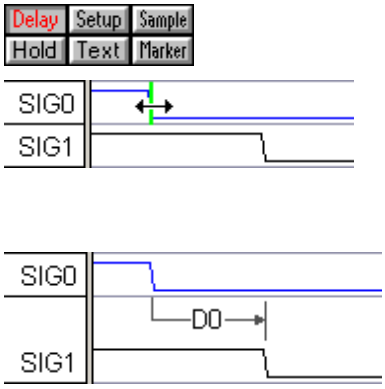
Delays are placed in a clock domain based on the same rules that apply to Samples (See Section 3.3 Samples that Block Clock Domains 54). The only difference is that it is not possible to create a delay that is not attached to an edge. Therefore delays are never triggered directly from the unclocked sequence, but from the clock domain of the edge the delay starts from (i.e. the starting edge of the delay is the triggering edge).

Triggering Edge	When Delay Activates
Attached to an edge on an unclocked signal	Activate if and when the triggering signal edge occurs
Attached to an edge on a clocked signal	Activate at next clock edge (not at triggering signal's edge) if a level sensitive check succeeds. The triggering signal is checked against the expected value that the signal transitions to in the diagram.
Attached to a sample action	Activates if and when sample condition is satisfied

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 2.9 Transactor Clock Domains and Waveform Code Generation 44.

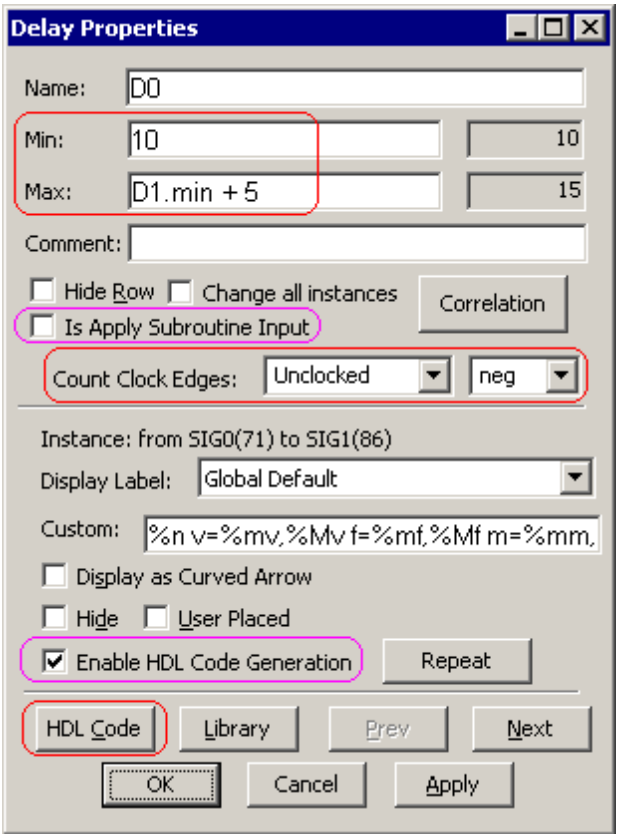
To add a Delay:

- Press the **Delay** button so that right clicks will add delays.
- **Left click** on the first transition in time to select it. This must be a blue input signal or an edge within a chain that starts from an input signal in order for the delay to generate conditional code, otherwise the delay just sets edge placement for code generation.
- **Right click** on the second transition to add the delay.

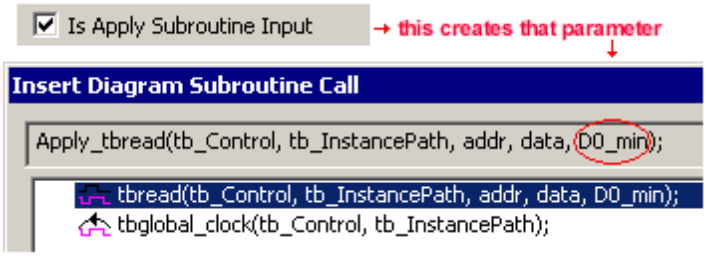


Copyright © 2014, SynaptiCAD Sales, Inc.

- Double click on the delay to open the *Delay Properties* dialog.
- Enter the **min** or **max** times (or time formulas like **D1.min + 5**). See the Timing Diagram Editor's Manual Section 2.5 Time Formulas for Clocks and Parameters for information on the syntax of formulas.
- At simulation time, only one value min, max, or typical (average of min & max) will be used. In TestBencher, the *Diagram Settings* dialog (discussed in Section 9.8: Diagram Settings Dialog (TestBencher Only) 148), 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.
- Delays can be clocked by specifying a clocking signal in the **Count Clock Edges** box and an **edge type** (pos, neg, or both). If a clock is specified, then the **min** and **max** boxes will specify the number of edges instead of time values.



- For TestBencher Pro, the **Is Apply Subroutine Input**, if checked, generates input parameters to the timing transactor Apply call that can be used to specify the values to use for the min and max settings of the delay. See Step 5: Define the Sequencer Process 17 for more information on the *Insert Diagram Subroutine Call* dialog.



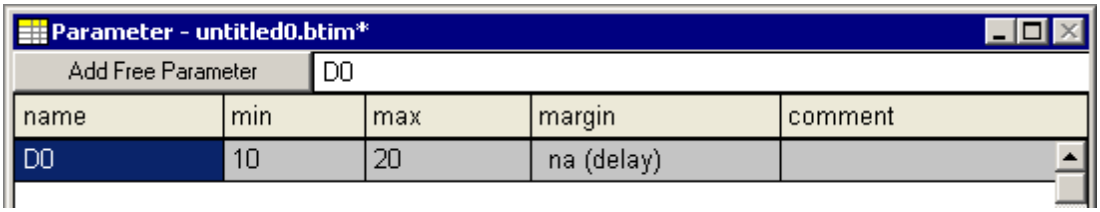
- The **Enable Code Generation** enables/disable the "edge-triggering" code for this delay. The delay's diagram level variables are created regardless, as these values may be referenced by other parameter's formulas.

Copyright © 2014, SynaptiCAD Sales, Inc.

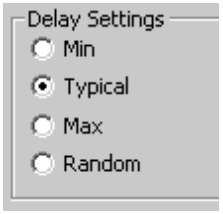
- Press the **HDL** code button to open the *Boolean Condition for Delay* dialog. This stores the condition that is checked before the delay drives the second edge. By default, the condition is TRUE (e.g. the default is for the Delay to always cause an edge transition when it's triggering condition occurs).



- You can type in the text for a new condition. 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.
    - **Alternative Approach**: If the condition is to be based on state values that occur during simulation, then it can also be done with a delay that is triggered from sample parameter so that the resulting diagram shows the test graphically rather than as a hidden test within the delay's Boolean condition dialog (see Section 3.4: Samples Triggering a Delayed Transition or Another Sample 56).
- Notice that the Delay has also been added to the *Parameter* window. The *margin* field has no meaning for delays and is marked as "na".



Selecting a time value for delays during simulation



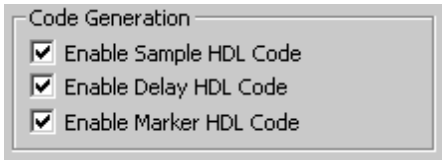
- The **delay settings** (Section 9.8 Diagram Settings Dialog 148) determine how delay times will be computed during diagram code generation. This setting determines whether min, typical, max, or random values will be used for the delay time.
- The **random** setting allows a random value to be computed from within the range of the min and max specified for the delay. All delays within a transactor will be randomized when this setting is selected, but delays can be fixed by setting their min and max values equal.
- Random values are generated using a uniform probability distribution. For VHDL projects, the default seed for the random number sequence is set in the sequencer process for the project and can be manually overridden by a subsequent call to InitializeUniform(seed1, seed2) by the user. For Verilog, the standard system task, $random, is used to generate random numbers.
- The **RandomDelays** project located in the**SynaptiCAD>Examples>TestBencher** directory demonstrates how to randomize delays within a range of time or a range of clock cycles, and also how to pass in a delay time to a transaction from an Apply call.

Copyright © 2014, SynaptiCAD Sales, Inc.

More Information on Delays:

- The Timing Diagram Editor's Manual Chapter 5: Delay, Setup and Hold Parameters has more information on controlling how the delay draws itself and on how to move a delay to different edges after they are drawn. Also, this chapter explains all of the timing analysis features of delays and how they affect the timing diagram. It also covers when an edge is driven by multiple delays ending on the same edge (e.g. this is useful when you want to drive an output signal's edge based on whether an edge transition occurs on one of several input signals).

- The HDL code generation for all delays, samples, and markers in a timing diagram can be disabled through the *TestBencher Diagram Settings* dialog. See Section 9.8: Diagram Settings Dialog (TestBencher only) 148 for more information on this feature.

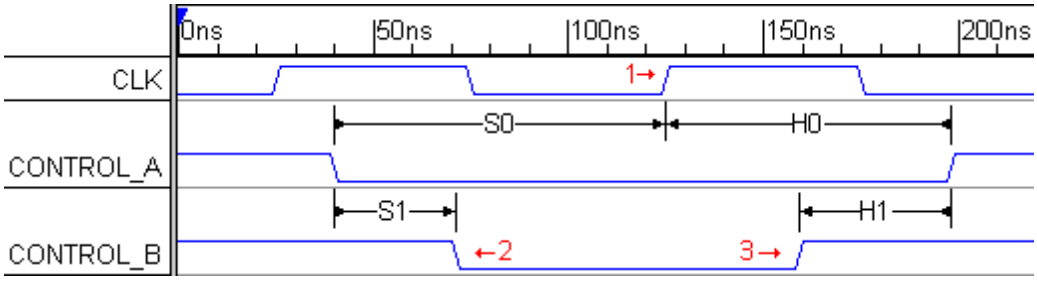


4.2 Setups and Holds

Setups and Holds check timing requirements for a design. **Setups** are the minimum time necessary that a signal must stay 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 4.3 Creating Continuous Setups and Holds 67.

Setups and Holds Clock Domains

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.

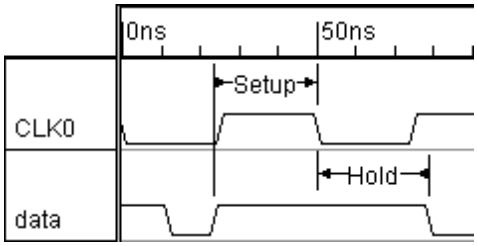
To create a setup or hold:

- Press the **Setup** or **Hold** button so that right clicks will add that type of parameter.

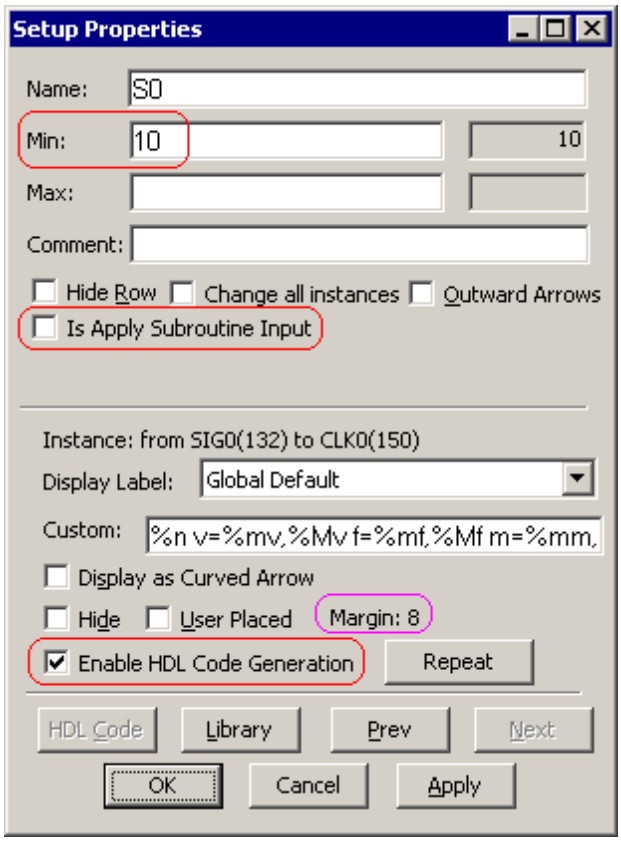


Copyright © 2014, SynaptiCAD Sales, Inc.

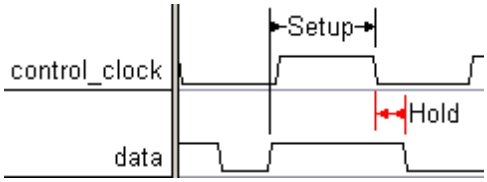
- **Left click** on the ***data* signal** to select it, then **Right click** on the ***control* signal** to add the parameter. To remember this order, remember that data signals have setup/hold constraints whereas control signals like clocks don't.



- If drawn correctly, the arrows will point to the control signal.
- Double click on the parameter to open the *Properties* dialog.

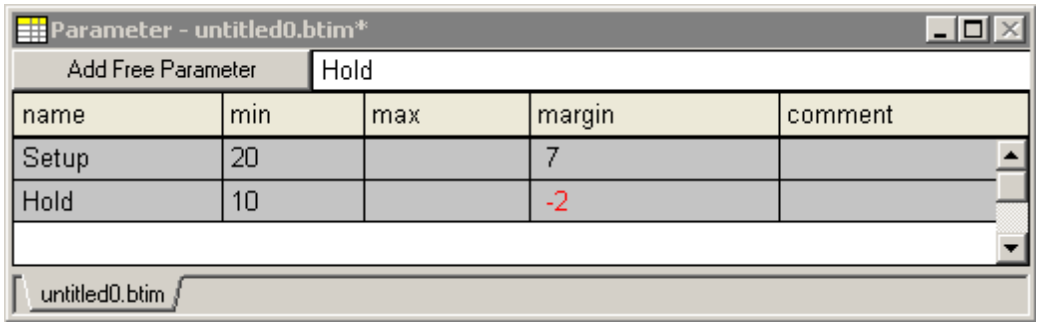


- Enter a **min** value, which is is the minimum time that the data signal has to be stable before or after the control edge, and it can be a time value or time formula. The **max** value is rarely used, but if specified the data signal must occur between the min and max times. See the Timing Diagram Editor's Manual Section 2.5 Time Formulas for Clocks and Parameters for information on the syntax of the formulas.
- For TestBencher Pro, the **Is Apply Subroutine Input**, if checked, 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 setup or hold.
- The **Enable Code Generation** enables code generation for this setup or hold.
- **Margin** indicates the amount of safety margin available before the constraint condition is violated.When a constraint is violated, it will turn red in both the *Diagram* window and the *Parameter* window. This just shows the margin of the drawn timing diagram; during simulation, margin errors will go to the simulation log file.



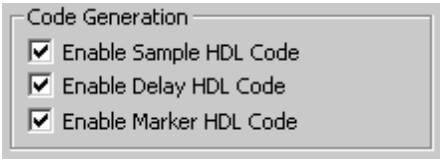
- The **Outward Arrows** checkbox changes the direction that the arrows on the parameter are drawn. This does not affect code generation but it is a popular graphical feature.
- Notice that the Setup or Hold also gets added to the *Parameter* window.

Copyright © 2014, SynaptiCAD Sales, Inc.



More Information on Setups and Holds:

- Every setup and hold has two variables, ***parametername*_min** and ***parametername*_max**, that can be used in other parameters and in user-defined HDL code statements.
- The Timing Diagram Editor's Manual Chapter 5: Delay, Setup and Hold Parameters has more information on controlling how the setup and hold parameter draw themselves and on how to move a parameter to different edges after it is drawn.
- The HDL code generation for all delays, samples, and markers in a timing diagram can be disabled through the *TestBencher Diagram Settings* dialog. See Section 9.8: Diagram Settings Dialog (TestBencher Only) 148 for more information on this feature.



4.3 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 in 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 section will describe how to create a continuous setup or hold that will persist for the duration of a timing transaction's execution.

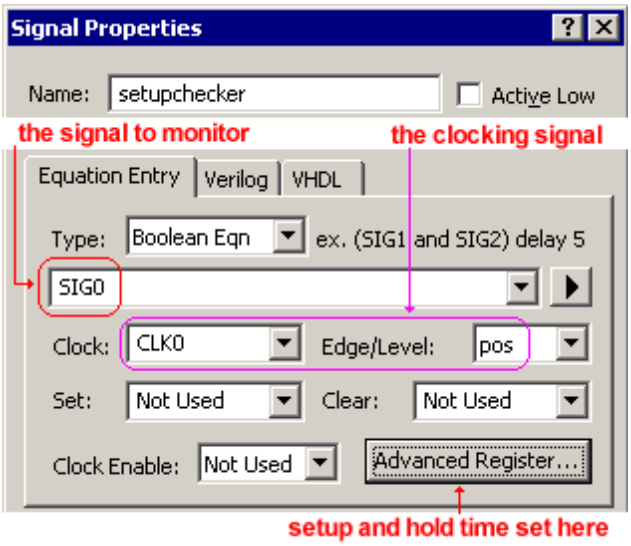
This feature is currently only supported in VHDL and Verilog, because it uses a feature called simulated signals. The simulated signals can be used to create registered and latch logic, so there are several optional controls that you can choose to use. This feature is not supported in WaveFormer Lite with Reactive Test Bench Generation Option.

To Create a Continuous Setup or Hold:

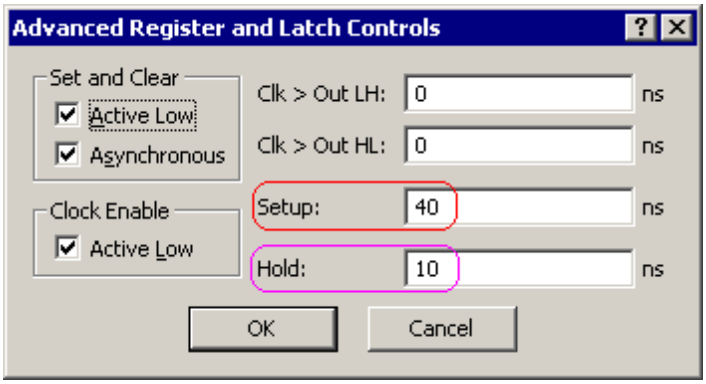
- Create an extra signal that will hold the equation that performs the continuous setup or hold on the two signals in the design.
- Double-click on the extra signal to open the *Signal Properties* dialog.

Copyright © 2014, SynaptiCAD Sales, Inc.

- In the Boolean Eqn box, enter the name of the signal to monitor (pick the non-clock if one of them is a clock).
- In the **Clock** box, select the other signal and choose an **Edge/Level** to compare the signals on.
- (OPTIONAL) The **Set**, **Clear**, and **Clock Enable** are optional signals that model the set, clear, and clock enable lines of the register or latch. These lines can be active low or high and synchronous or asynchronous depending on the settings in the Advanced Register and Latch Controls dialog.



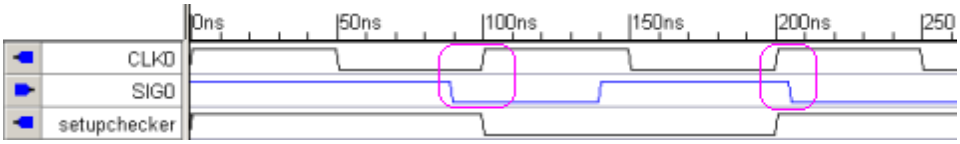
- Press the **Advanced Register** button to open the *Advanced Register and Latch Controls* dialog.
- Enter a Setup time, a Hold time, or both times.



- **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 simulation.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 simulation.log, shown in the report window.

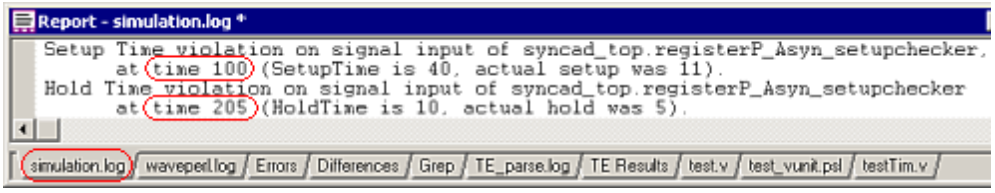
Continuous Setup and Hold After Simulation

After simulation, the graphical waveform for the extra setup-checker signal will just be the output of the D flip-flop that was created when you entered the Boolean equation and the Clock Signal. Since this is not relevant to the continuous setup or hold check, you can, if you wish, hide the signal by selecting the signal name and choosing the **View > Hide Signal** menu.



The continuous setup or hold will cause messages to be sent to the Report Window **simulation log** tab. In the above diagram there is both a setup and hold error.

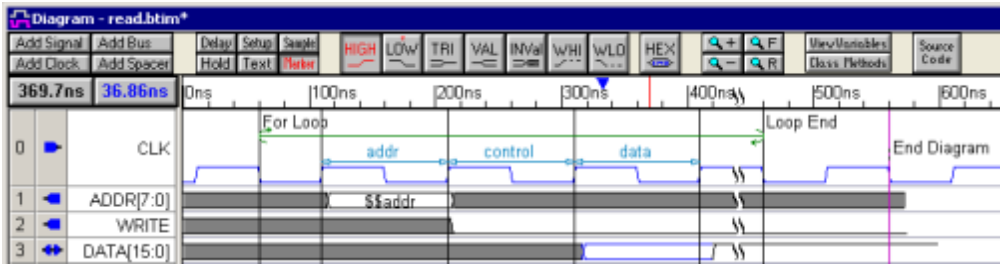
Copyright © 2014, SynaptiCAD Sales, Inc.



Copyright © 2014, SynaptiCAD Sales, Inc.

Chapter 5: Transaction Markers

Markers are vertical lines that display their placement time in the timing diagram. Once a marker is added to the timing diagram, its behavior can be altered to specify actions to be taken by the transaction during execution. These actions can include identifying the end of a transaction, creating loops in the transaction, creating pipeline sections, executing HDL code, blocking a process until an event occurs, and pausing the simulation.



Markers are triggered either by the unclocked process or by the clocked process of the edge they are attached 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 2.9 Transactor Clock Domains and Waveform Code Generation 44).

5.1 Adding a Marker to a Diagram

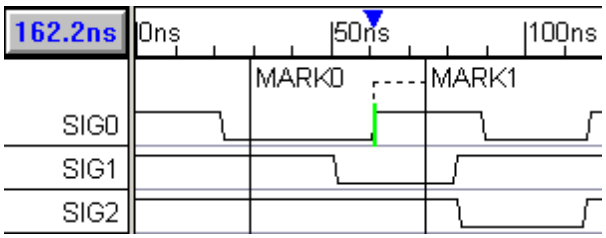
As with samples, markers can be absolute or relative. An **absolute marker** is attached to a specific time and is triggered by the unclocked process. A **relative marker** is attached to a specific edge, and is triggered by the clocking domain of the edge (See Section 2.9 Transactor Clock Domains and Waveform Code Generation 44).

To place a marker in a diagram:

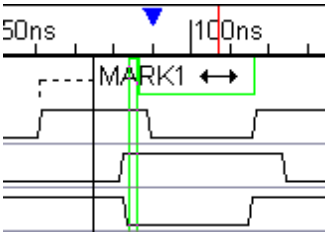
- Press the **Marker** button so that right clicks will add marker lines to the diagram.



- To attach a marker to a ***time***, make sure no edges are selected, then right click on a waveform to add a marker.
- To attach a marker to an ***edge***, select the control edge, then right click. If the edge is moved, the the marker will stay a fixed distance from the edge



- To move a time marker, drag and drop the time marker line or name

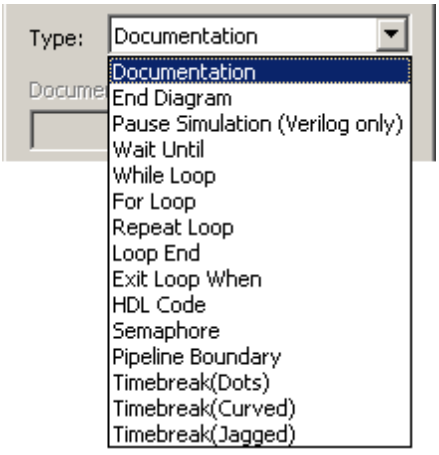


Copyright © 2014, SynaptiCAD Sales, Inc.

Editing a Marker's Properties:



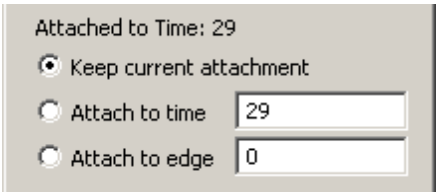
- Double click on the marker to open the *Edit Time Marker* dialog.
- The **name** field for markers can be blank, so that by default the marker only displays its line. This is the only object besides *spacer signals* that can be nameless.



- The **Marker Type** controls the function of the marker. 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.
- *Wait Until* causes the process that triggers the marker to block until the condition becomes true.
- *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.
- *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.

Change the Attachment of the Marker:

The **Attach to time** controls are used to change the attachment or placement of a marker. Notice that at the top of the controls there is a readout that describes the current attachment to either the time or to the signal-and-time of the current attachment.



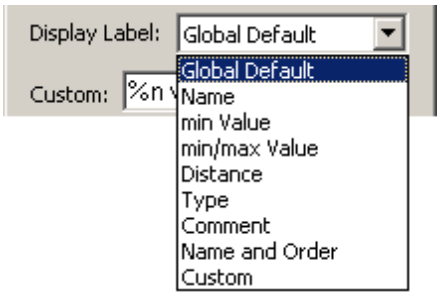
- To move a relative marker to the exact edge time, type **0** into the **Attach to edge** edit box.
- To attach to a new edge, check the **Attach to edge** radio button and click **OK** to ***close the dialog*** and enter into an edge selection mode. As you move the cursor, a green bar will hop to the closest edge. Left click on the edge that you want to attach the marker.

Copyright © 2014, SynaptiCAD Sales, Inc.

- To attach to a new time, check the **Attach to time** radio button and enter a time into the edit box.

Display Name or any other property:

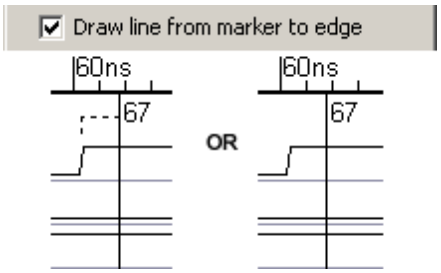
- Markers can display any of their properties listed in the **Display Label** box. The global default is set in the **Options > Drawing Preferences** menu.
- The **Custom** label displays the string in the **Custom** box with the control codes replace by values. Notice that the default string lists all of the possible control codes so you do not have to remember them, just edit the default string.
- The **Type** label for documentation markers is Blank so that you can just have a line and no other label.




```
name = %n
min value = %mv
max value = %Mv
distance = %md
type = %t
```

Edge lines for Markers:

- **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.



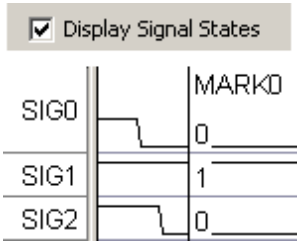
Display Label Position

- The Marker Label can be dragged and dropped so that it appears between two signals. Once this is done you can check the **Auto-Adjust Display Label Position** to return label placement control to the editor.



Display Signal States with the Marker

- Markers can permanently display the signal state values if the **Display Signal States** box is checked in the *Marker Properties* dialog. Otherwise the signal states are just shown when the mouse hovers over the marker line.



Copyright © 2014, SynaptiCAD Sales, Inc.

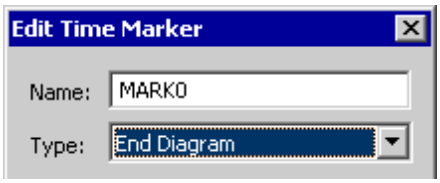
5.2 End Diagram Markers

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 (or for terminating the transaction early than the drawn waveforms). In TestBencher, the 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.

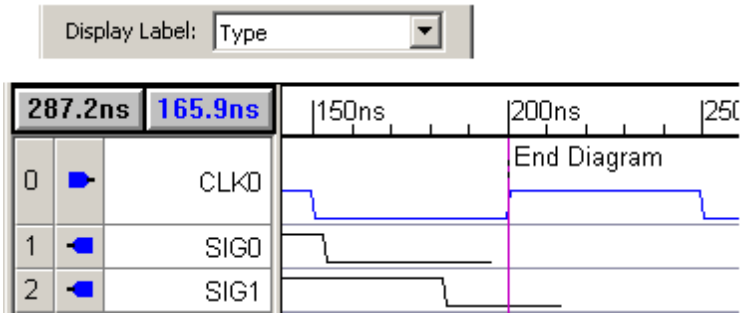
If there are no End Diagram markers, then the longest non-clock signal will determine the end of the timing diagram. If there is more than one End Diagram Marker then the earliest one will determine the end of the timing diagram. End Diagram Markers are displayed using a purple line.

To modify a 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.



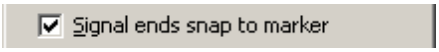
- Since there is usually only one End Diagram marker in a diagram, it useful to set the **Display Label** to **Type** so that the marker will display "End Diagram" rather than its name.
- Notice that the end diagram marker is drawn in purple to indicate that it will stop the simulation.



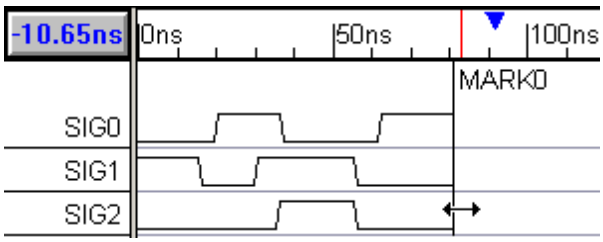
Graphically making waveforms end at common time:

The End Diagram Marker will make the testbench code end at the Marker time, however the drawn timing diagram may not look as neat as the the generated code. If you want all the signals to redraw themselves so that they end on the Marker line, use the following feature:

- Double click on the Marker to open the *Edit Time Marker* dialog and check the **Signal ends snap to marker** box and close the dialog.



- Then immediately, drag-and-drop the marker and notice that the ends of all the drawn signals snap to the new position of the marker.



Copyright © 2014, SynaptiCAD Sales, Inc.

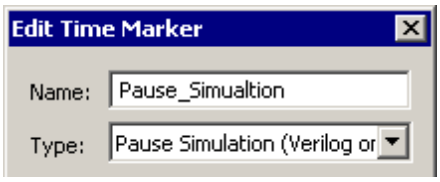
5.3 Pause Simulation Marker (Verilog Only)

A Pause Simulation marker will pause the entire simulation when it reaches the marker. This marker is like 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 (see Section 5.6 HDL Code Marker 77).

To specify a Pause Simulation marker:

- Add a marker to the timing diagram. The marker will pause all processes in the entire model.
- Double-click on the marker to open the *Edit Time Marker* dialog.
- From the *Marker Type* drop-down list, choose **Pause Simulation (Verilog only)**.

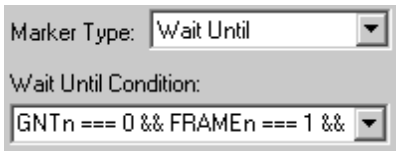


5.4 Wait Until Marker

Wait Until markers provide a mechanism for indefinitely blocking the execution of one clocked process within a transaction. This type of marker blocks the transaction until its condition becomes true. Blocking samples also block the execution of a process, but they have a time-out built into the window and multiplier settings. Wait Until markers do not time-out. The process that gets paused will be the triggering process of the Marker (see Section 2.9 Transactor Clock Domains and Waveform Code Generation 44).

To specify a Wait Until condition:

- Add a marker that is attached to a signal or clock edge in the diagram. The edge will define the clock domain that is to be blocked.
- Double-click on the marker to open the *Edit Time Marker* dialog.
- From the *Marker Type* drop-down list, choose **Wait Until**.
- Then in the **Wait Until Condition** edit box, enter a condition. The condition can be any equation in the generation language that evaluates to TRUE or FALSE at simulation time.



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.

Alternative Blocking Constructs (Sensitive Edges and Samples)

There are two other constructs that can be used to block the execution of a clock domain:

- A **sensitive edge** 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 (see 2.6 Sensitive Edges - a

Copyright © 2014, SynaptiCAD Sales, Inc.

Blocking Construct 40).

- A Sample that has the **blocking** setting checked (see Section 3.3: Samples that Block Clock Domains 54) will block its clock domain until the sample completely finishes, including execution of its *then* or *else* action.

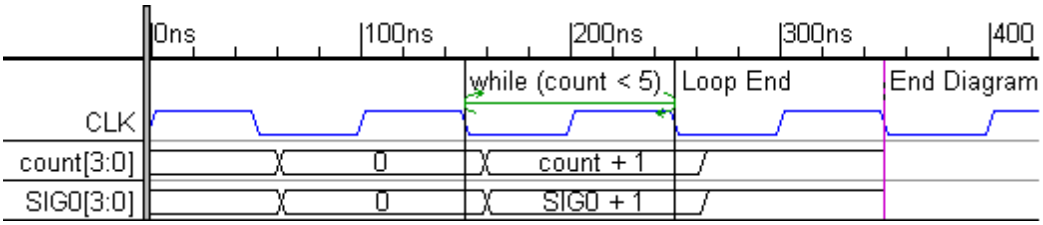
Also, Section 10.2 Waiting for Signal Transitions or Conditions 157 has an example using all three blocking features.

5.5 Loop Markers

Loop markers are used to create sections in the transaction that are repeated during simulation. For example, a loop can be used to create a burst read transaction that has to determine, at simulation time, the number of cycles needed to complete the read cycle (see Section 10.3: Burst Mode Transactions 159). TestBencher and the Reactive Test Bench Option support ***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.

Loop Markers and Clock Domains

Loop Markers only loop over the clocking domain that they are placed in. The following timing diagram demonstrates how a marker loop might not loop over everything in the diagram. The while loop will loop over the **CLK_neg** clocking domain since the Begin and End loop markers are attached to falling edges of CLK. The CLK_neg domain also contains the **count** signal, because that signal has its Clock and Edge properties set to **CLK** and **neg**. Each time the loop is executed, the address of the count signal will be incremented. However, SIG0 is outside the CLK_neg clock domain, because its Clock property is set to **unclocked**. The **SIG0+1** equation will be evaluated once and remain static through out the rest of the iterations of the loop.



Triggering Order of Markers and other objects:

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 trigger 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 2.7 Controlling the Triggering Order of Parameters 40).

Create a Loop using Markers:

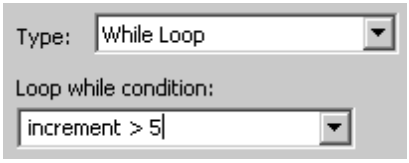
The same process must trigger both the beginning and ending loop markers (see Section 2.9 Transaction Clock Domains and Waveform Code Generation 44). For clocked transactions, this means that the begin and end markers need to be attached to the same edge type of a given signal. For unclocked loops, both markers should be attached to a time. When the timing diagram editor recognizes the beginning and ending of a loop it will draw a green loop line between the markers.

- 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.

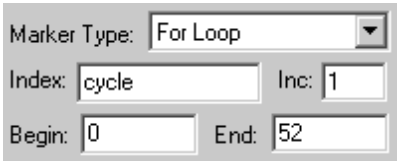
Copyright © 2014, SynaptiCAD Sales, Inc.

- Double-click on the marker **on the left** to open the *Edit Time Marker* dialog. Choose one of the loop types to define the beginning of the loop:

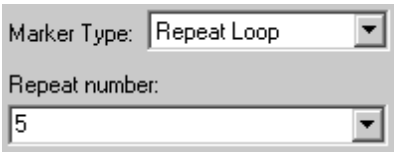
- **While Loop** marker, when matched with an *End Loop* marker, will execute continuously over a sequence of test vectors until a defined condition is met. The **condition** can be any equation in the generation language that evaluates to a TRUE or FALSE at simulation time.



- **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.

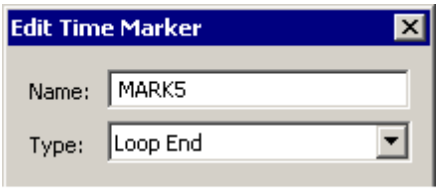


- **Repeat Loop** marker will execute for a specified number of iterations.

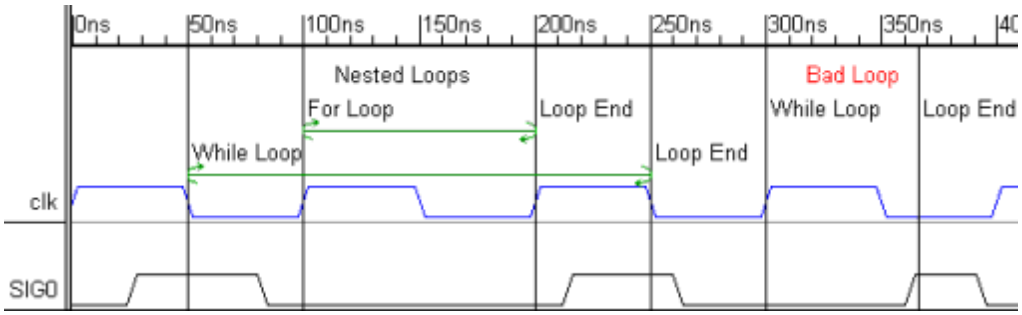


- Press the **OK** to close the dialog.

- Double-click on the marker **on the right** to open the *Edit Time Marker* dialog.
- Choose the **Loop End** marker type.
- If you set the **Display Label** to **Type**, then the marker will display "Loop End" instead of the marker name.
- Press the **OK** button to close the dialog.



- If the markers are triggered from the same clock domain, then TestBencher will draw a loop line between the markers. If there is no loop line, check the attachments of each marker. In the example below, the bad loop will not work, because the while marker is triggered by the rising edge clk process and the loop end marker is triggered by the unclocked process.

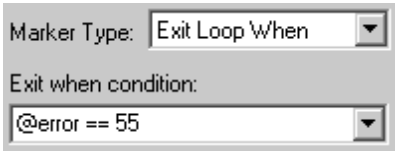


Break out of a loop early using an Exit Loop When marker:

The *Exit Loop When* marker will terminate the inner most loop that both graphically surrounds it and that is triggered off of the same clock domain.

Copyright © 2014, SynaptiCAD Sales, Inc.

- Add a marker to the timing diagram. It should be relative to the same signal and edge type (or to time) as the markers that form the loop to be broken.
- Double-click on the marker to open the *Edit Time Marker* dialog.
- Choose the **Exit Loop When** marker type and add a condition on which to exit. The condition can be any equation in the generation language that evaluates to a TRUE or FALSE at simulation time.



Samples can Break Loops or Skip Past Section of the Loop:

- Add a Sample to the timing diagram, as shown in Section 3.1: Adding a Sample 49. It should be relative to the same signal and edge type (or to a time) as the markers that form the loop.
- Double click on the sample to open the *Sample Properties* dialog.
- Press the **HDL Code** button to open the *Code Generation Options* dialog.
- Define an **If Condition** as shown in Section 3.2: Sample Checking and Actions 51.
- In the **Then Action** or **Else Action** boxes choose one of the following actions:
  - ***Break Loop*:** stops the loop that immediately surrounds the sample.
  - ***Continue Loop*:** returns to the beginning of the loop immediately surrounding the ample, skipping the last part of the loop.

5.6 HDL Code Markers

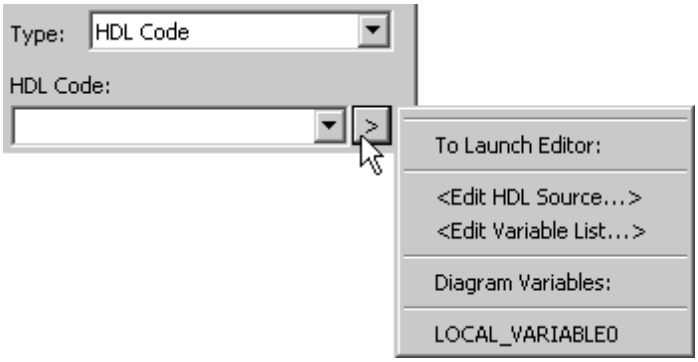
HDL code markers are used to make calculations and execute code that is not easy to represent graphically. HDL code markers have a limited amount of space for typing, so it is usually just used to type in the name of a function to call. The code box accepts direct HDL code in the transaction generation language. You can make calls to class methods (Section 6.3: Class Methods 87), library subroutines (Section 9.4: Libraries and Use Clauses 141), or insert any code that is valid within the context of a process (VHDL), an always block (Verilog), or a 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.
- Choose **HDL Code** from the Type box.
- Then either type in the source code into the **HDL Code** box.

OR

- Press the **>** button and select **<Edit HDL Source...>** from the menu to enter multiple lines of source code.



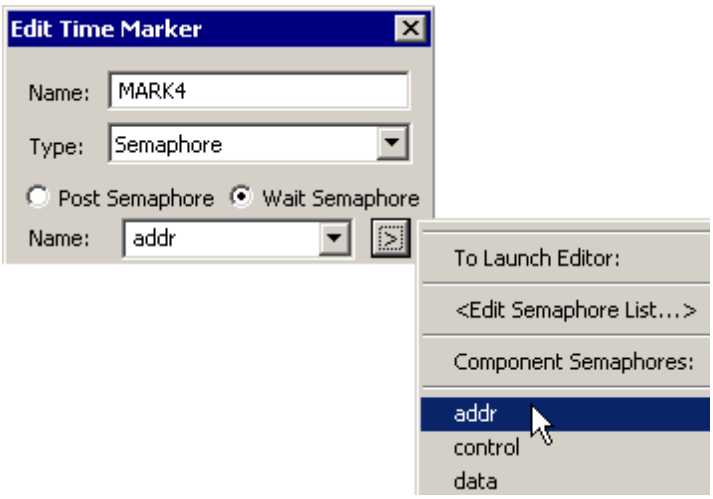
Copyright © 2014, SynaptiCAD Sales, Inc.

5.7 Semaphore Markers (TestBencher Pro only)

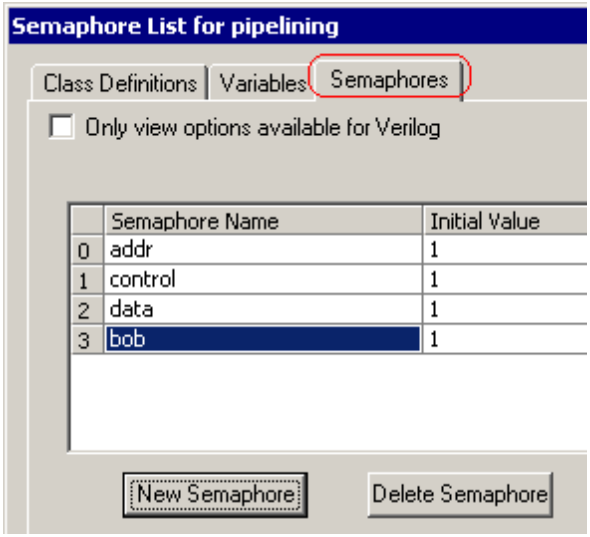
In Test Bencher Pro, semaphore markers are used to secure critical regions during a transaction. Semaphore markers can be created for transactions that are part of a project that contains semaphores. There are two types of Semaphore Marker: *wait* and *post*. A **Wait Semaphore Marker** waits for the 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.
- Choose **Semaphore** from the **Type** box.
- Select either **Wait** or **Post** radio buttons to define whether the semaphore variable will be incremented or decremented when the marker is encountered.
- In the **Semaphore name** box, either enter the name of the semaphore variable or use the **>** button and choose a variable from the menu list.



- To create a new semaphore variable with a specific initialization value, press the **>** button and choose **<Edit Semaphore List...>** menu to open the *Semaphore List* dialog.
- Use this dialog to delete semaphore variables and to edit the initialization value.

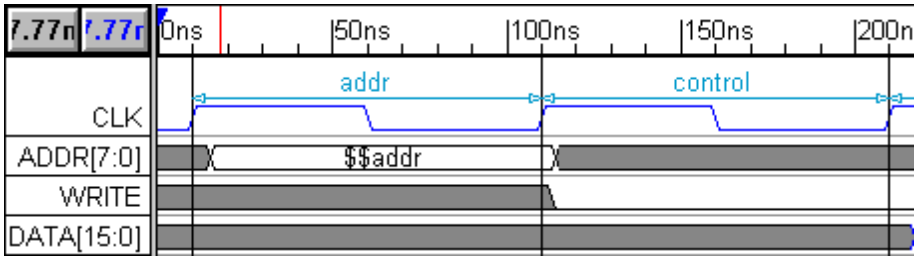


5.8 Pipeline Boundary Markers (TestBencher Pro only)

In TestBencher Pro, Pipeline Boundary Markers can be used to create pipelined transactors. These markers will automatically create a semaphores for each critical region (if the semaphores do not already exist in the project). A critical region is created for each stage of the pipeline. Loop markers can be placed within a critical region - but only if the complete loop is inside the region. In other

Copyright © 2014, SynaptiCAD Sales, Inc.

words, the loop cannot overlap the pipeline stage - it can be entirely inside the stage or entirely around the outside of the stage.

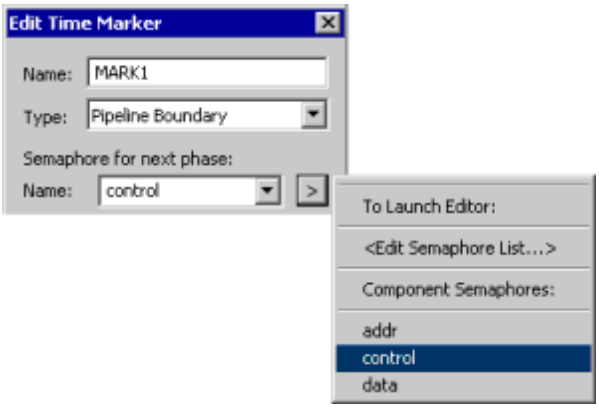


The **Semaphore Name** specified in the *Marker Properties* dialog represents the next pipeline stage and is displayed in the diagram between the start and end boundaries of the stage. The next Pipeline Boundary marker in the diagram (either the start of the next stage, an **End Boundary** pipeline marker or and **End Diagram** marker) represents the end of the stage. In the diagram above, for instance, the first marker is a Pipeline Boundary marker with **addr** selected as the semaphore. This stage is ended by the **control** Pipeline Boundary marker - which also starts the control stage. Section 10.1: Pipeline Transactions 154 shows an example of a project that has different timing diagrams that all share the same pipeline.

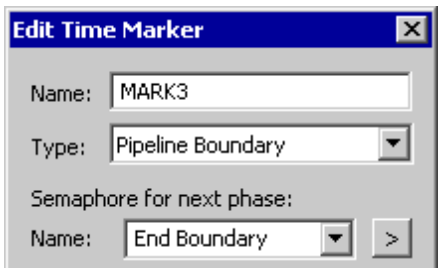
To Create a New Pipeline Stage:

You must create at least two markers in the same clock domain (see Section 2.9 Transactor Clock Domains and Waveform Code Generation 44). That means the beginning and ending markers should be attached to the same edge type of the clock signal or both attached to times.

- First, place a marker at the time or edge where the stage should begin. Double-click the marker to open the *Marker Properties* dialog.
- Choose **Pipeline Boundary** from the type box.
- Enter the name of the stage in the *Semaphore for next phase* box or use the **>** button and <Edit Semaphore List> to open the *Semaphore List* dialog as shown in the previous section.
- Close the Marker dialog.

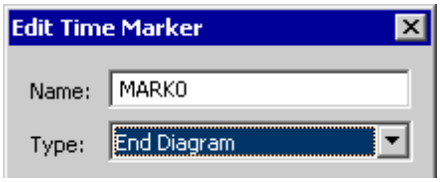


- Place a second marker at the time or edge that will end the pipeline stage. This marker must be one of the following types (otherwise a start stage without and ending marker for the stage will produce an error):
  - Either another **Pipeline Boundary** with either the *Semaphore for Next phase* of **End Boundary** (if it's the last stage in the pipeline) or the ***name*** of the next adjacent stage.

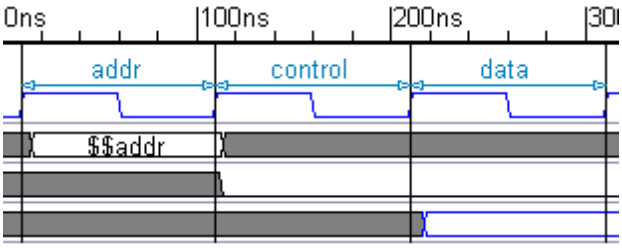


Copyright © 2014, SynaptiCAD Sales, Inc.

- Or an **End Diagram** marker which automatically ends the stage. This is a nice feature to use when the last pipeline stage is the last event in the diagram.



- A properly drawn pipeline stage will have a blue line between the pipeline markers that lists the name of the stage.



Instance Count:

The **Instance Count** of a diagram represents the number of instances of the transactor that are created during simulation (each transactor instance can run simultaneously). TestBencher will automatically set the **Instance Count** for the diagram during generation to the number of pipeline stages in the diagram. 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 diagram Instance Count is set in the Diagram Settings 148 dialog.

- Select **Project > Default Diagram Settings...** from the main menu to open the *Diagram Settings* dialog.
- The **Count** determines the number of instances of a transactor. These instances will be automatically instantiated in the test bench.



5.9 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.

Code Generation for Documentation and TimeBreak Markers:

If **Verbose Markers** is checked in the *Diagram Settings* dialog, then a message is written to the simulation log file when the documentation or time break marker is reached (see Section 9.8: Diagram Setting Dialog (TestBencher Only 148)).These markers also generate a comment line in the source code for the transaction.

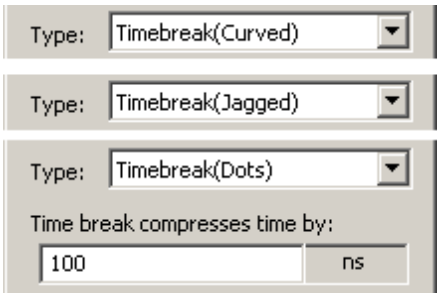
Documentation Markers:

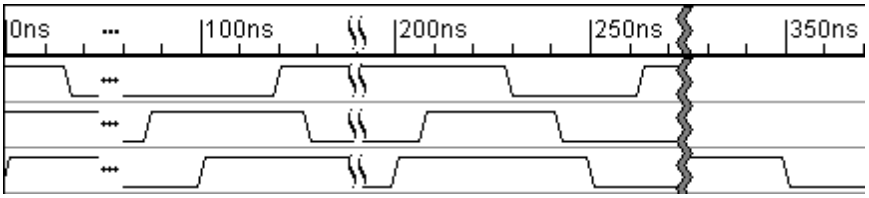
When a marker is first added to the timing diagram, its type is **Documentation**. Double click on the Marker to open the *Edit Time Markers* dialog and use the controls to change the appearance of the marker.

Copyright © 2014, SynaptiCAD Sales, Inc.

TimeBreak Markers:

- Add a Marker to the diagram and double click to open the *Edit Time Markers* dialog.
- Choose one of the **Timebreak** types to make a dotted, curved, or jagged line.
- To compress time to the right of the marker, enter a time into the **time break compress time by** box. The compressed time still exists, but it does not display on the screen. This only affects the drawn timing diagram and does not have any effect on the generated test bench code.

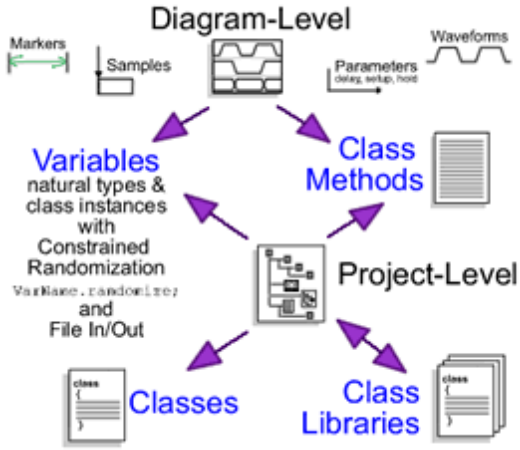




Copyright © 2014, SynaptiCAD Sales, Inc.

Chapter 6: Classes, Variables, and Files

Both TestBencher Pro and Reactive Test Bench support **diagram-level variables**, which allow data to be passed to different objects within the timing diagram, and **diagram-level classes**, which can be called from the transaction code to perform algorithmic functions that are not easy to define graphically. TestBencher Pro also supports **project-level variables** and **classes** so that data and algorithms can be shared by different transactions within the project.



During simulation, random number generators can drive the values for variables. The *Constraints* dialog is used to define limits of the random number generation for random variables. A variable can also be used to read from or write to ASCII text files.

Test Bencher Pro also supports **Class Libraries,** which can be shared between **different projects** (see Section 9.1: Sub-Projects 137). These libraries are used to define 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.

6.1 Variables

In TestBencher, variables can be local to either the diagram or to the project's Transaction-Level Model (Reactive Test Bench only supports diagram-level variables). Diagram-level variables are declared within the transactor source code, and project-level variables are declared within the Project's TLM. Simple variables can be created while editing diagram objects. For example, adding a signal state with a prefix of $$ creates a state variable.The *Classes and Variables* dialog enables the creation of more complex variable types. Variables can be used anywhere that you type in HDL code including: Boolean equation and Condition boxes, Sample values, Loop control variables, HDL code markers, class methods, and waveform state values.

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 (depending on the abilities of the target generation language).

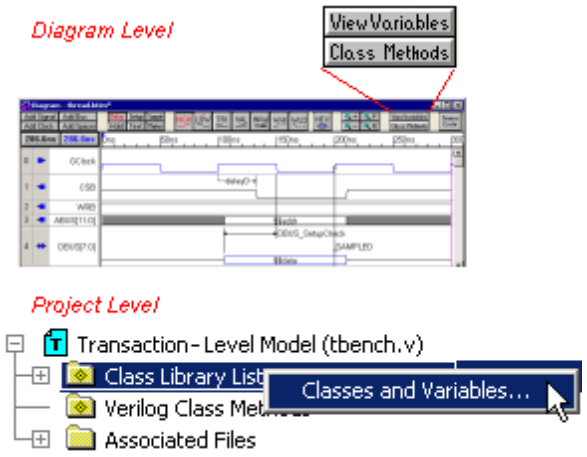
Create Variable using the Classes and Variables Dialog:

- Open the *Classes and Variables* dialog from either the diagram (for diagram-

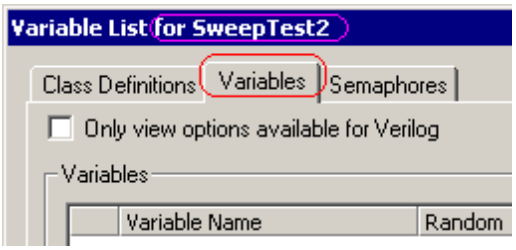
Copyright © 2014, SynaptiCAD Sales, Inc.

level variables) or from the Project (for project-level variables):

- For diagram-level variables, press the **View Variables** button in the *Diagram* window.
- For project-level variables, right click the **Transaction-Level Model** folder (or most of the folders in that section) and select **Classes and Variables** from the context menu. It can also be opened by the **Project > Classes and Variables** menu or by double-clicking on the **Variable List** (if variables already exist).

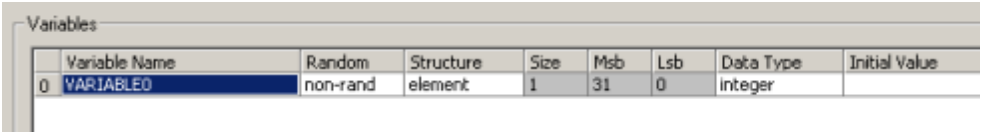


- Select the **Variables** tab. Depending on where you opened the dialog and which objects are defined at that level, the dialog may have several tabs or just the Variables tab.
- Notice that the title bar of the dialog indicates the timing diagram or project file that the variables are stored within.



- Press the **New Variable** button near the bottom of the dialog to add a new variable. You can also just click on a 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 6.12: Variable and Class Field Properties 107. In particular, the **Data Type** property can be either one of the builtin types for the generation language or a Class type as defined in Section 6.2: Classes 86.

Diagram-Level Variable Overview:

There are three types of variables that can be used inside a transaction: diagram variables, state variables, and parameter-based variables. 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	Advantages	Disadvantages

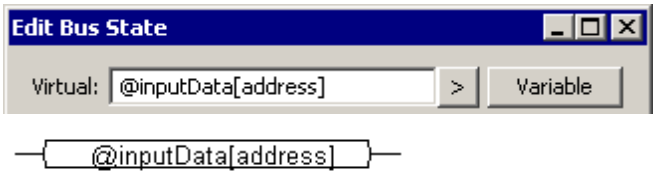
Copyright © 2014, SynaptiCAD Sales, Inc.

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 is controlled by the signal type to which the variable is attached.
Parameter	Input	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.

Using Diagram Variables

Diagram Variables are created using the *Classes and Variables* dialog as described above. These are the most versatile structures for holding information, complex data structures, and for passing information into and out of the transaction.

- In a waveform state, put an @ symbol prefix in front of the diagram variable 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 *Edit Bus State* dialog.

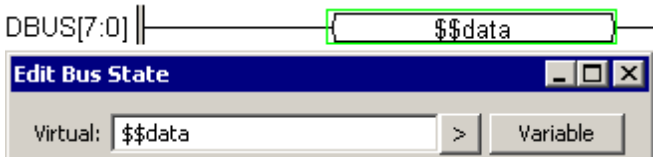


- In a block of code, just use the diagram variable name. Some examples of code that might use a variable are class methods, condition statements for samples and markers, and marker loop parameters such as begin, end, and increment values.
- To use a diagram variable to store a sample value, use the **Variable** features of sample's *HDL Code Generation* dialog as described in Section 3.6: Storing Sample Values in User Defined Variables 59.

Using State Variables (TestBencher Only)

TestBencher state variables defined in the *Edit Bus State* dialog by typing in a variable name that begins with two dollar signs, like *$$addr*. See Section 2.4 Waveform States with Variables and Conditions 36 has more information on creating state variables. State variables are automatically added as an input 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 transactor is called, a new state value can be passed into the variable.

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



- State variables are added to the parameter list for the transaction

```
// Apply_tbwrite( addr , data );
   Apply_tbwrite( 'h55 , 'hee );
```

Copyright © 2014, SynaptiCAD Sales, Inc.

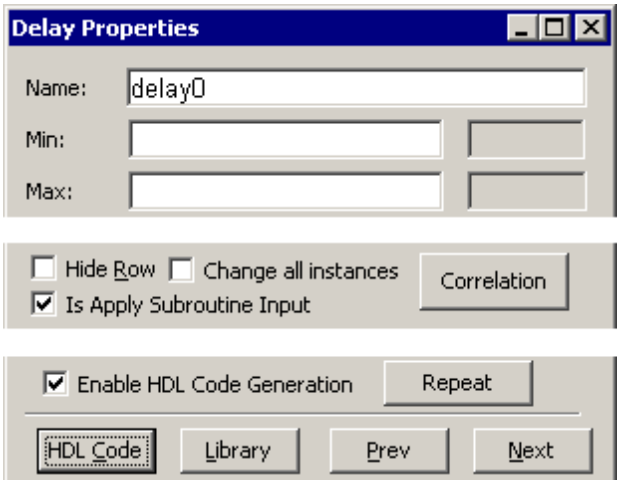
call.

Using Parameter Variables

Delays, setups, holds, samples, and free parameters have two variables, ***parametername*_min** and ***parametername*_max** that can be used to hold information and to pass information into a transaction. These variables pass time (either in real times or in clock cycles) to transactions. 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 reference the parameter variable inside a diagram, just use ***parameterName*_min** or ***parameterName*_max** in any code block, condition statement, or in the min or max boxes of a parameter. Inside other parameters you can also use the name without the min/max suffix, and TestBencher will make the best guess as to which value you want.

- To pass a parameter variable into a timing diagram, first add a parameter to the diagram, then double click on the parameter to open the *Parameter Properties* dialog.
- Check **Is Apply Subroutine Input** to cause a subroutine parameter to be added to the transaction apply call.
- Verify that **Enable HDL Code Generation** is checked so that code will be generated for this parameter.
- When you insert the diagram transaction call into the Sequencer Process, the call will include a parameter for the delay value.



- Below is some code that would loops over a transaction calls and sweeps a delay value over a specified range..

```
for (delay0 = 32.0; delay0 > 15.0; delay0 = delay0 - 5.0)
  begin
     // Apply_tbwrite( addr , data , $realtobits(delay0_min) );
        Apply_tbwrite( 'h55 , 'hee , $realtobits(delay0) );
     // Apply_tbread( addr , data , $realtobits(delay0_min) );
        Apply_tbread( 'h55 , 'hee , $realtobits(delay0) );
  end
```

Scope rules for 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 TLM file, but the access depends on the language you are using. Chapter 11: Language Specific Details 170 covers the details on accessing variables.

Copyright © 2014, SynaptiCAD Sales, Inc.

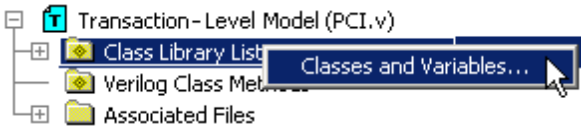
6.2 Classes

In TestBencher, project level classes can be created to hold variables and class methods that act on those variables. Since basic Verilog and VHDL do not directly support classes (as opposed to C++ and SystemVerilog), the generated code does not always look like class code, but inside the TestBencher Project and the Sequencer code that uses the class, variables and methods act like what you would expect a class to act like. Once a Class is created, it can be used when creating variables just like any native data type of the language (Section 6.1: Variables 82).

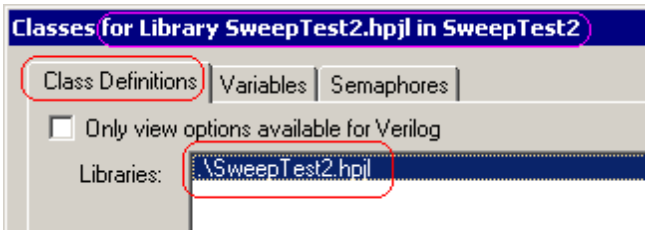
Class fields can use any of the generation language's native data types or a user-defined data type (another class). Each field can have a structure of a simple element or depending on the language a queue, an array, or an associative array. For example, you can create an ethernet packet class, where each field represents a different portion of the packet's payload. 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.

Create or edit a class:

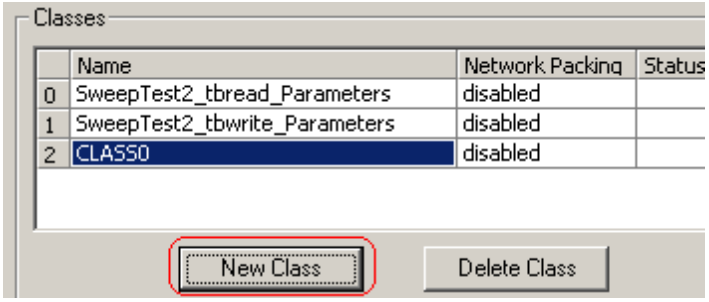
- Right click on the **Class Library List** folder and choose **Classes and Variables** to open the *Classes and Variables* dialog.



- Select the **Class Definitions** tab, and notice the top of the dialog describes which classes are being edited.
- Then ***select the Library*** that will contain the class. Each project has a default library named after itself (seeSection 6.4: Class Libraries 91 for more information on libraries).



- The Classes list shows all of the classes within the selected library.
- Press the **New Class** button to add a new class to the Classes list.
- Double click on any cell to edit a property of the class.



- Selecting a class name causes the *Class Fields* list to display the fields of the class. It also causes the **Constraints** and **Class Methods** buttons to act on the selected class. See below for adding fields, methods, and constraints to the classes.

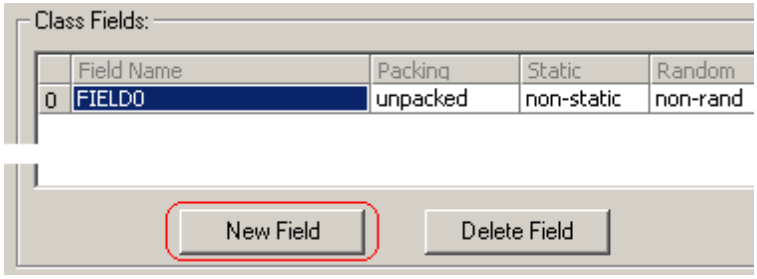
Add or edit a field of the class:

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

Copyright © 2014, SynaptiCAD Sales, Inc.

on the generation language it can have complex structures like array, associative array, and queue.

- ***Select a class*** to cause its fields to be displayed in the *Class Fields* list.
- Either, press the **New Field** button to add a field to the *selected class*.



- OR, press the **Define from Template** button to open a file dialog. When you choose a test vector file, the program will read the header for the file and create a data structure that has the same fields as the column headers of the file. See Section 6.9 Input Data Dynamically from a File (VHDL) 100 describes the format of the file.



- Double-click on the cell for any field property to edit it. Section 6.11 Variable and Class Field Properties 107 defines the usage of the 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

Add or edit a method to a class:

- ***Select a class*** so that the Class Methods dialog will know which class to act on.
- Press the **Class Methods** button to open a dialog for entering the class method. Section 6.3: Class Methods 87 describes this dialog.



Add or edit a constraint to a class for random number generation:

- ***Select a class*** to edit.
- Press the **Constraints** button to open a dialog for entering the constraints of the selected class. Section 6.5 Constrained Random Number Generation 92 describes this dialog.



6.3 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 Transaction-Level Models, 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.

Copyright © 2014, SynaptiCAD Sales, Inc.

Scoping Level and Access to Diagram, Project, and Class level methods

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).

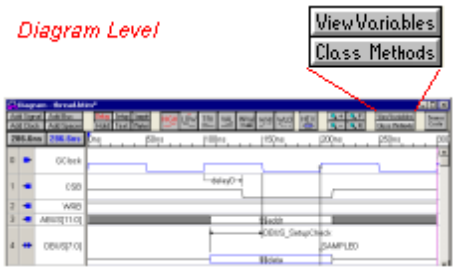
For TestBencher, **class-level class methods** can be called from the same scoping level as the variable that instantiates the class. These methods are a part of the class definition and are stored in the respective class library. The methods are generated in the class source code definition.

For TestBencher, **project-level class methods** can be accessed from the Sequencer Process in the TLM file. 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.

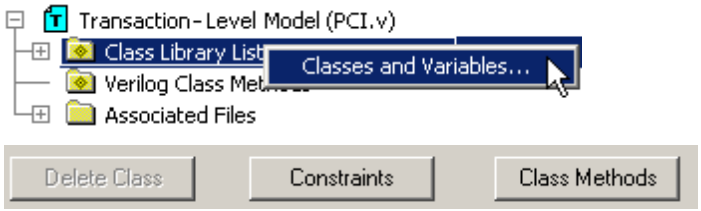
Open the Class Methods dialog from the object that will own the class method:

The *Class Methods* dialog is used to create and edit user-defined methods. Open the *Class Methods* dialog from the object that you want to define the class method for:

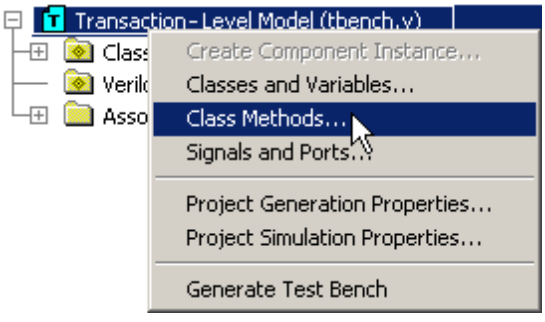
- For ***diagram-level methods***, press the **Class Methods** button located in the *Diagram* window. These are the only methods supported by the Reactive Test Bench Generation option.



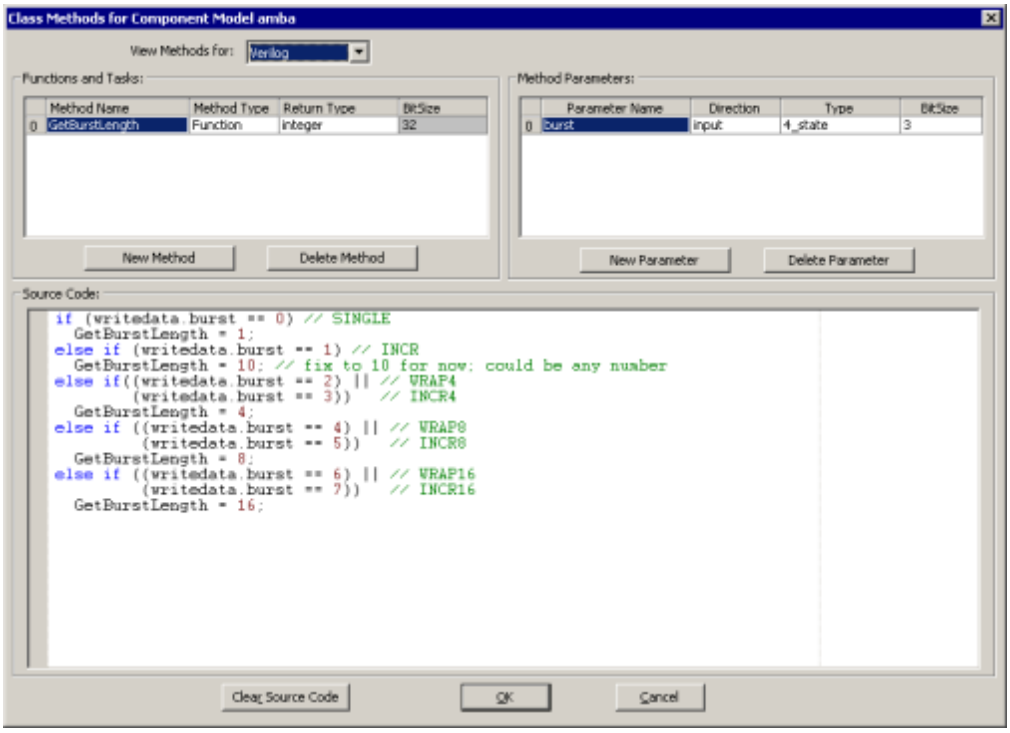
- For ***class-level methods***, open the *Classes and Variables* dialog by right clicking on the Class Library list. Select a library and a class, then press the **Class Methods** button.



- For ***project-level methods***, right click on the **Transaction-Level Model** folder in the *Project* window and choose **Class Methods** from the context menu.



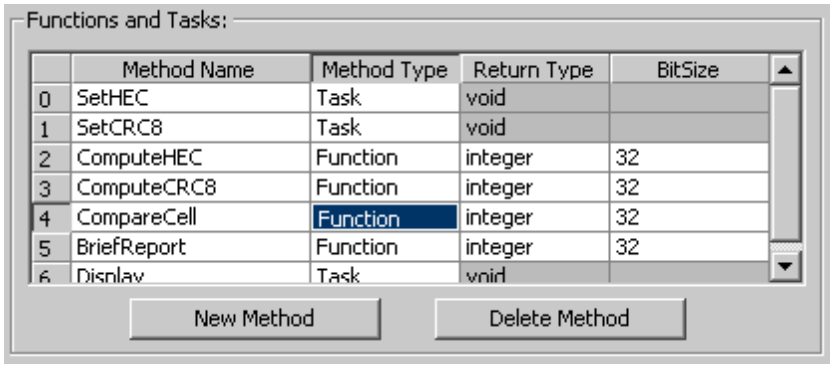
Copyright © 2014, SynaptiCAD Sales, Inc.



Create a Class Method and define its type and return type:

Selecting a different class method name changes the contents of the other sections of the dialog.

- Select the Language for the method in the View Methods box at the top of the dialog.
- Press the **New Method** button to create a new method with default values.
- Double-click on the cell to edit that property.
- **Method Name** is the name of the method.



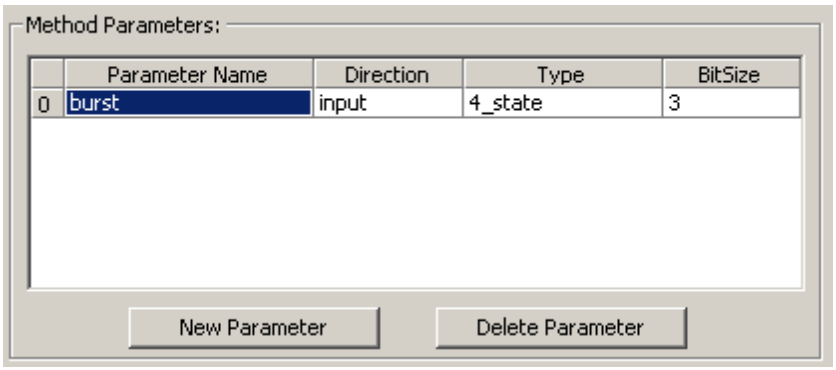
- **Method Type**: ***Tasks*** perform an operation on the parameters that they are passed, but do not specifically return any value. ***Functions*** perform an operation on the parameters that they are passed and return a value.
- **Return Type** is the data type of the value that is returned by a function. This can be any of the SynaptiCAD language independent types listed in Section 6.13: Language Independent Types 109.
- **BitSize** is 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.

Specify the Parameters for the Selected Class Method:

The parameters represent data that is passed into the method.

Copyright © 2014, SynaptiCAD Sales, Inc.

- Select a class in the **Functions and Tasks** section of the dialog.
- Press the **New Parameter** button to create a new parameter with default properties.
- Double-click on a cell to edit that property.
- **Parameter Name** is the name of the parameter.

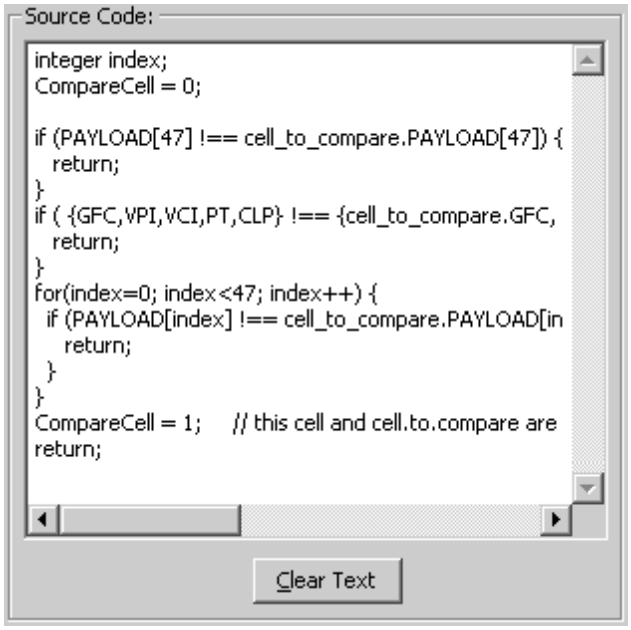


- **Direction** is
- **Type** is the data type of the parameter. This can be any of the SynaptiCAD language independent types (6.13: Language Independent Types 109), or any of the classes in the project (Section 6.2: Classes 86).
- **BitSize** is the size in bits of the parameter. This value is only editable if the *Type* is a bit type.

Add the Source Code for the selected Class 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 Transaction-Level Model to be language independent. Add the class method source code:

- Select a class in the **Functions and Tasks** section of the dialog.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.



Using Diagram-level Class Methods

- To use a diagram-level class method in an **HDL Code marker**, type in the name of the class method along with any parameters into the code box of the marker (Section 5.6: HDL Code Markers 77).

Copyright © 2014, SynaptiCAD Sales, Inc.

- 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.

Using a Project Level Class Method

- Double Click on the **Transaction-Level Model** folder to open the top-level test bench file and search for the **User Defined Class Methods** code block. This is where the code for project-level will be defined.

```
// User Defined Class Methods
function integer GetBurstLength;
  input [2:0] burst;
  begin
  ... the source code body
endfunction
// End $UserDefinedMethods
```

- Search for the **Sequencer** comment, then enter code that calls the class method.

```
// Transaction Sequencer - After this comment
....
  repeat (5) // number of bursts to perform
     begin
      ahb_master0.ApplyRandom_idle;
      writedata.randomize; // randomize all fields
      burstLength = GetBurstLength(writedata.burst);
....
```

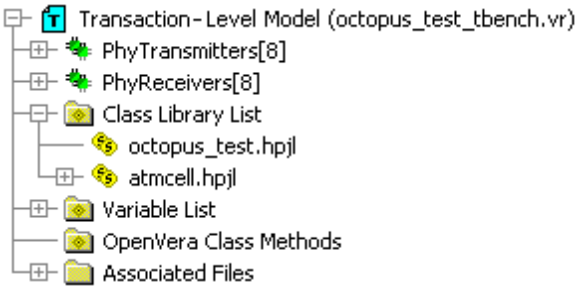
6.4 Class Libraries

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.

To create, edit, add, or remove a Class Library use the Classes and Variables dialog:

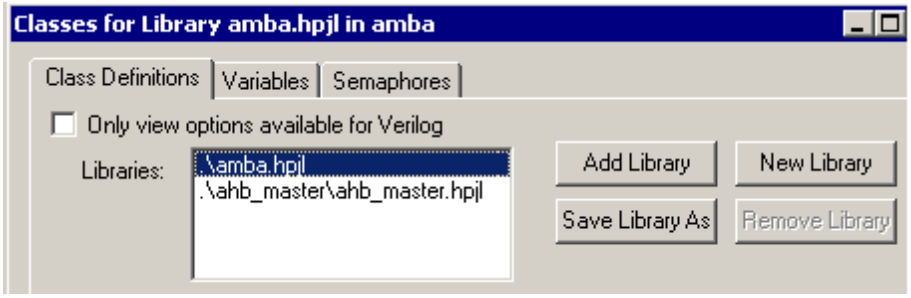
All of the class libraries for a project are listed in the **Class Library List** folder in the *Project* window. All of the class libraries for a particular project share the same scope, so the class names must be unique for a particular project. For example, if two libraries contain a class named *ATMCell*,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

- Open the *Classes and Variables* dialog by either:
  - Double clicking on a library in the **Class Library List** folder, in the *Project* window.
  - OR if no projects are open choose the **Project > Classes and Variables** menu.
- Note that whether or not a project is open, modifying a class library will affect any project that includes the library.



Copyright © 2014, SynaptiCAD Sales, Inc.

- Use the buttons in the Library section to create, edit, add or remove libraries.



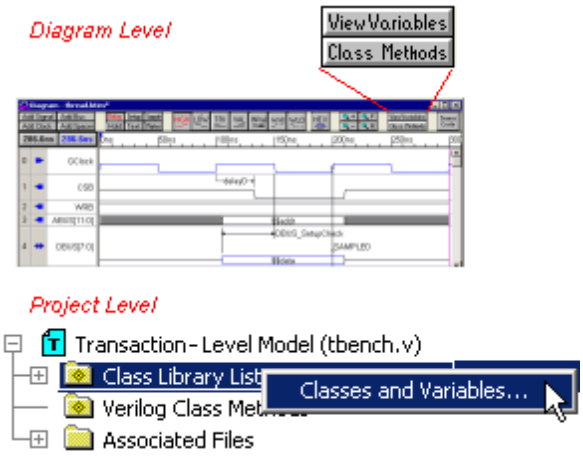
- By default, the *Class Definitions and Variables* dialog is displayed in a language independent view. This means that options may be shown that are not available for the currently selected language. This is provided because class definitions are language independent and can be shared between projects.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

6.5 Random Number Generation and Constraints

Random values can be generated for variables by calling the **randomize** method on them (e.g my_variable.randomize). TestBencher will only generate a randomize method for a variable if at least one of the fields of the variable's class has the Random property set to rand or randc. When randomization is used, constraints can be used to specify the valid ranges of the random numbers generated.

Open the Classes and Variables dialog to set the Random Property and define any Constraints:

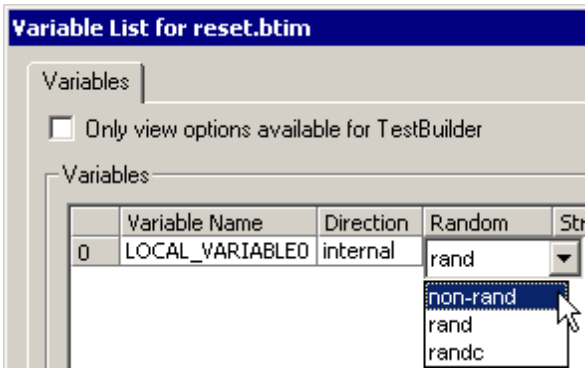
- Open the *Classes and Variables* dialog from either the diagram (for diagram-level variables) or from the Project (for project-level variables):
  - For diagram-level variables, click the **View Variables** button in the *Diagram* window.
  - For project-level variables, right click the **Transaction-Level Model** folder (or most of the folders in that section) and select **Classes and Variables** from the context menu.



- Select the **Class Definitions** tab or the **Variables** tab depending on what you are editing.
- If you working with a class, select the library and class that you want to constrain.

Copyright © 2014, SynaptiCAD Sales, Inc.

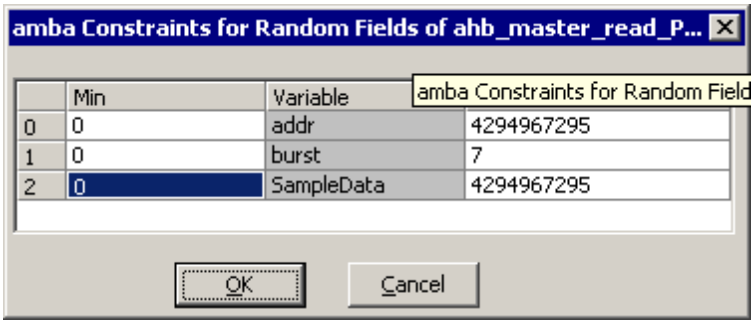
- Set the **Random** field the variable or class parameter to one of the Random properties. This causes a ***randomize*** method to be created. This can be called from the Sequencer process of the the value needs to be randomized. See Section 6.12 Variables and Class Field Properties 107 for more information.



- If you want to constrain the range of random values for the class, go to the **Class definitions** tab and press the **Constraints** button to open the Constraints dialog.



- Notice that the title bar of the dialog indicates the class that you are editing.
- Enter a min and max constraint for each field.



Randomizing a variable's fields using the Randomize Function

- Double-click on the **Transaction-Level Model** folder in the *Project* window to open the template file and scroll down to the **Sequencer Process** as show in Step 5: Define a Sequencer Process 17. Then call the **randomize** function for the variable.
- For Verilog:

```
VariableName.randomize;
```

- For VHDL:

```
randomize(VariableName);
```

- If the variable is an instance of a class with several fields, then the **randomize** function will only set values for the fields that have a Random property. You must subsequently set the non-random fields.

```
// Example of mixed random and non-random
writeData.randomize; //sets rand fields
writeData.addr = 0xffff; // set a non-rand field
Apply_write(writeData); //use the variable
```

Randomizing parameters to a transaction call

Every transactor (timing diagram) has a class that contains the parameters to that transactior's Apply call. Variables of this class can be randomized.

- Double-click on the *Transaction-Level Model* folder in the *Project* window to open the template

Copyright © 2014, SynaptiCAD Sales, Inc.

file and scroll down to the **Sequencer Process** as show in Step 5: Define a Sequencer Process 17.

- If all of the timing diagram parameters are random variables, then just call the ApplyRandom function instead of the Apply call for the transaction.

```
ApplyRandom_DiagramName;
```

- If the parameters for the diagram are mixed random and non-random, then randomize a variable of the class first, set the non-random properties, and finally pass in the variable through a normal apply call as demonstrated above in the ***Randomizing a Variable's Fields*** section.

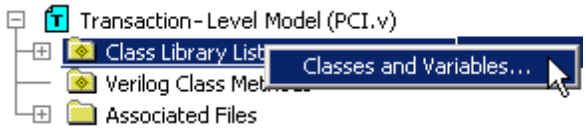
6.6 Data Packing and Bit Streams

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.

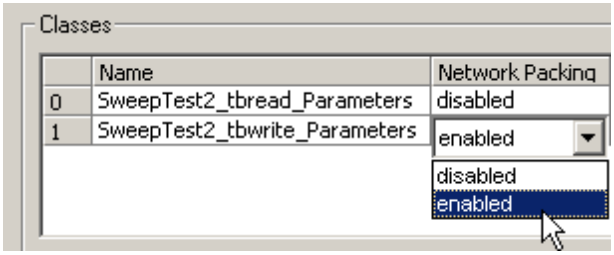
When creating structure definitions, packing properties can be individually specified for each field of the structure. Packing can be enabled individually for each field and when enabled there are several different ways the field can be packed. These options, which are listed below, control the order 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.

Set the Packing Properties for a Class

- Right click on the **Class Library List** folder and choose **Classes and Variables** to open the *Classes and Variables* dialog.

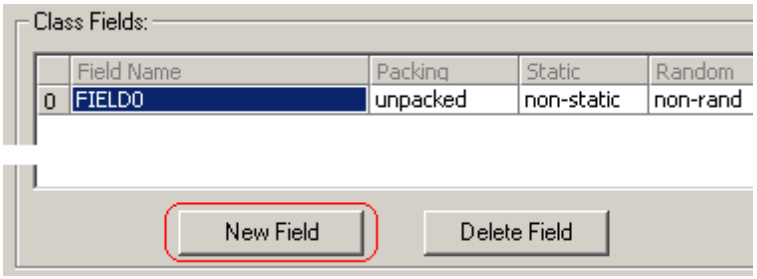


- Select a Class Library and a Class so that the fields of the class are displayed in the Class Fields region.
- In the **Classes** section, the **Network Packing** control, if enabled for the class definition, will reverse the order of the bytes in the resulting packed byte array.

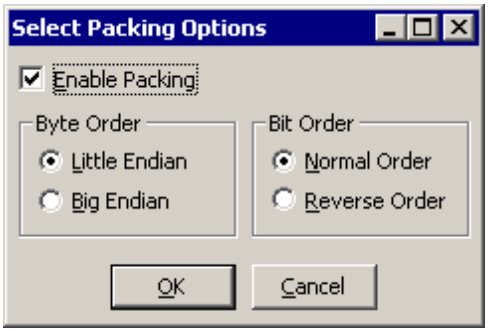


Copyright © 2014, SynaptiCAD Sales, Inc.

- In the **Class Fields** section, double click on the **Packing** cell of the field to open the *Select Packing Options* dialog.



- Check **Enable Packing** to activate the controls in the dialog.
- **Byte Order** controls how the bytes are packed in to the structure. Little Endian (default) or Big Endian. If Big Endian is used, then for each field:
  - each pair of bytes is swapped (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 (if the size of the field is not a multiple of 32, then this step will not be performed).
  - For Example if you have a 32 bit word 01 02 03 04 in hex, then little Endian would write it out as 01 02 03 04, and Big Endian would write it out as 04 03 02 01.



- **Bit Order** determines the order of the bits in the stream. **Normal order** arranges the data so that the most significant bit of each byte is first in the vector. **Bit Reverse** arranges the data so that the least significant bit of the vector to be packed first and the most significant bit to be packed last.

Chapter 11: Language Specific Details 170 discusses the language specific features of data packing that TestBencher supports.

6.7 SynaptiCAD Test Vector File Format

The next four sections cover file input and output for the test benches, and each uses the SynaptiCAD Test Vector Spreadsheet format. For either direction, you will first create the data file and then TestBencher will use the information in the header of the file to create the file access class and variables. For output files, only the first two lines of the file need to be created. For input files, you will also need to write in the data values.

```
[Vectors]  Radix=hex
&AddrOut    %DataRead[7:0](bin)
A4          10110011
54          11011010
....
```

The Test Vector Spreadsheet format is a tab-separated format that can be read or written by a normal spreadsheet program. Test vector files can also be created by defining signal names and sizes in a timing diagram and then saving the diagram in the **Test Vector Spreadsheet without timing** format.

Copyright © 2014, SynaptiCAD Sales, Inc.

Rules for the SynaptiCAD Test Vector File Format:

- There can be other lines above the line that contains the **[Vectors]** statement. Once the [Vectors] is found the rest of the file needs to be in the specified format, until another bracketed section or the end of file is found.
- A default radix may be specified for all fields of the file, but if none is specified then a default radix of **hex** is used. This specification is made with the **Radix=*type_of_Radix***. Each column in a file can individually override the default radix by placing a **(*radix*)** after the name and size information.

```
[Vectors] Radix=defaultRadix
```

- The supported radices are:
  - **hex** (hexadecimal)
  - **bin** (binary)
  - **dec** (decimal)
  - **real**

- The line after the **[Vectors]** line contains the **field names** and optionally the **direction**, **size** and **radix** of the class field. Put one **TAB** character between each field name column if you are using an editor to create the file. If you are using a spreadsheet program, export the file in a tab-separated output format. There should be no spaces within the column information.
  - If no size is given, it is assumed to be a single bit vector.
  - If no radix is stated, then it is assumed to be the default radix for the file.
- If no direction character is used, it is considered an output. The direction characters are @ = input, & = output, and % = in-out.

```
(direction)fieldName1[MSB:LSB](Radix)<TAB>fieldName2<TAB>fieldName3
```

- For input files, the rest of the rows should have data in the columns that match the size and type of the column names. Each data value is separated by a <TAB> character.
- See the next four sections for techniques to use the data files.

Creating a Test Vector file from a Timing Diagram

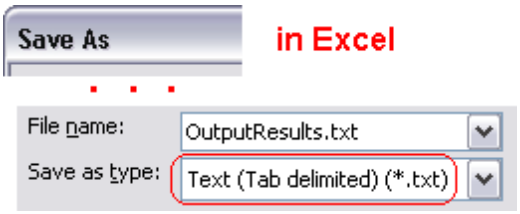
- The easiest way to make a Test Vector file is to create a timing diagram with the signal names and sizes representing the column headers and export it. Export the diagram using the **File > Save Timing Diagram As** menu and select the file type: **Test Vector Spreadsheet without Timing**.

Creating a Test Vector file with Excel

- Another way is to use Excel or a text editor.

	A	B
1	[Vectors]	Radix=hex
2	AddrOut[7:0]	DataRead[7:0] (bin)
3		

- In Excel, we saved as an *Other Format* of type **Text (Tab Delimited)**. This created a text file were each column is separated by a TAB and there are **no spaces** within the column information.



Copyright © 2014, SynaptiCAD Sales, Inc.

6.8 Output Data to a File (VHDL & Verilog)

VHDL and Verilog test benches can write test result data to a file. The output file is a tab-separated format called the SynaptiCAD Test Vector Spreadsheet format. This file can be read by a normal spreadsheet program.

```
[Vectors]   Radix=hex
AddrOut     DataRead[7:0](bin)
A4          10110011
54          11011010
....
```

To output to a file, you first create the output text file that defines the names and sizes of the data and add the file to the **File Output** folder. TestBencher automatically parses the file and creates the project-level classes and variables needed to write to the file. Inside a timing diagram, you can use a **Sample** to write the data to the file using the **@FileName_out.ColumnName** syntax. Each time the timing diagram is applied, the file variables in that diagram will write to the next line of the output file.

1) Create an output file with a header which will be used to make an output class

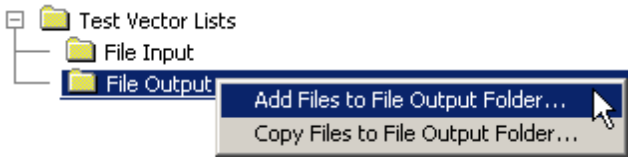
Create a tab separated text file with two lines using one of the methods described in Section 6.7 SynaptiCAD Test Vector File Format 95 (exporting this file from a timing diagram is generally the best method). These two lines define the names, sizes, and radices of the data to be written out.

```
[Vectors]   Radix=hex
AddrOut     DataRead[7:0](bin)
```

2) Add the file to the File Output Folder to automatically create the file class and variables:

When a file is added to the project's ***Test Vector List - File Output*** folder, TestBencher will automatically parse the file and create a *Class* definition with a field for each column of the file. TestBencher also instantiates the class as a file variable in the project. This file variable can be accessed anywhere in the project.

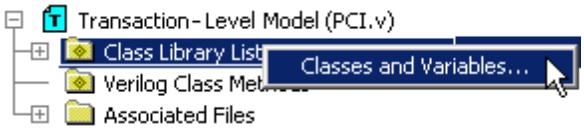
- Under the **Test Vector Lists** folder, right click on the **File Output** to open the context menu.



- Choose one of the menu options to attach the file to the project and create the file class and variables. The **Add Files to** menus will add the file and path to the project. The **Copy Files to** will make a local copy of the file in the project directory.

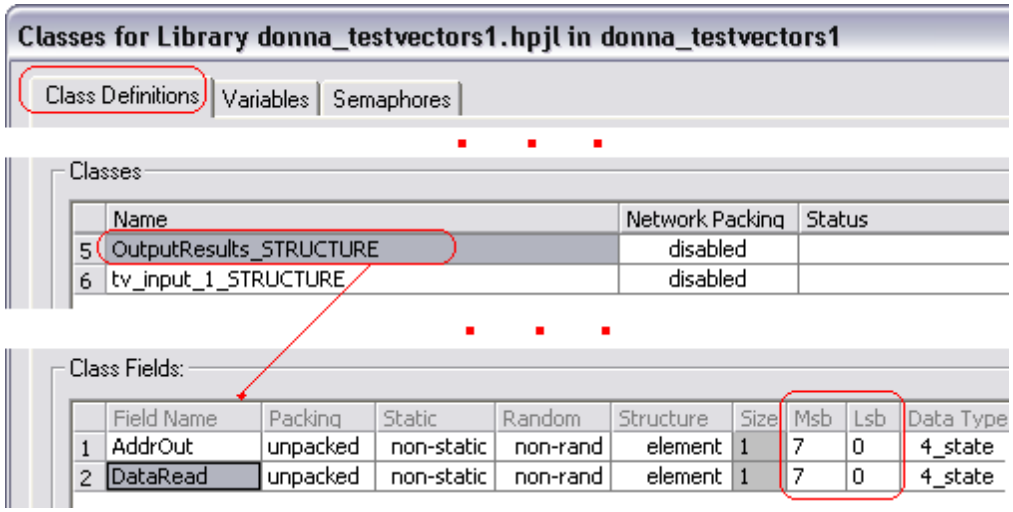
Investigate the class and file variable that were created:

- Right click on the **Class Library List** folder and choose **Classes and Variables** to open the *Classes and Variables* dialog.

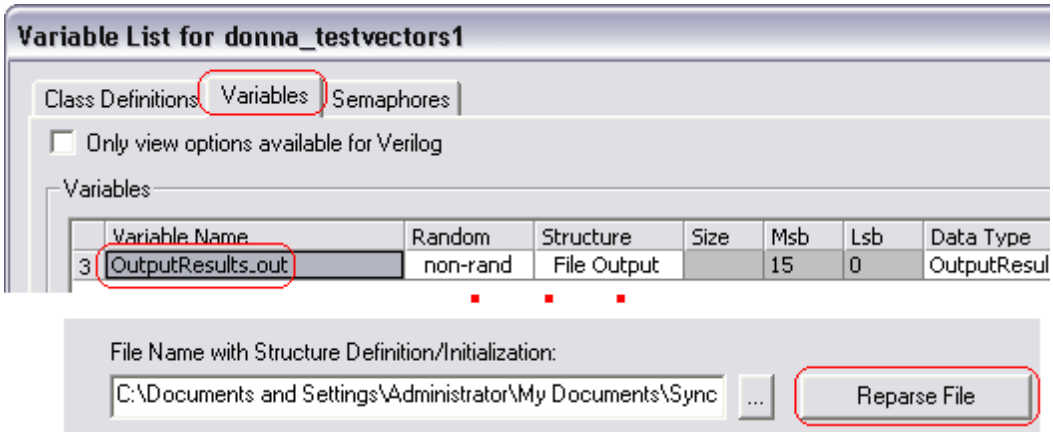


- In the **Class Definitions** tab, notice that a class was created and named ***filename*_STRUCTURE**. The fields of the structure are the column headings of the file.

Copyright © 2014, SynaptiCAD Sales, Inc.



- In the Class Fields section, make sure the field names and sizes match what you want. A blank class field usually indicates an extra space in the test vector file.
- If you need to change the output file, double click on the file in the File Output list to open it, make the desired changes, and save it. Next click on the **Variables** tab, select the file variable, then press the **Reparse File** button to update the class structure created from the file.

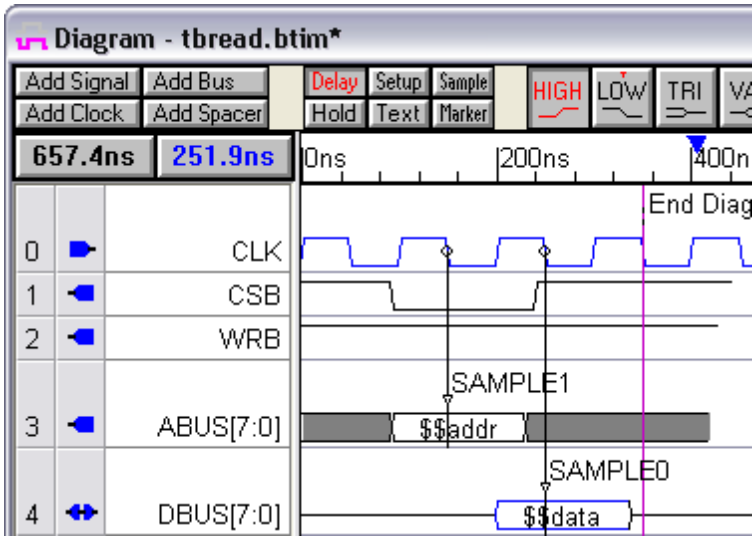


3) Add Samples to write to the File

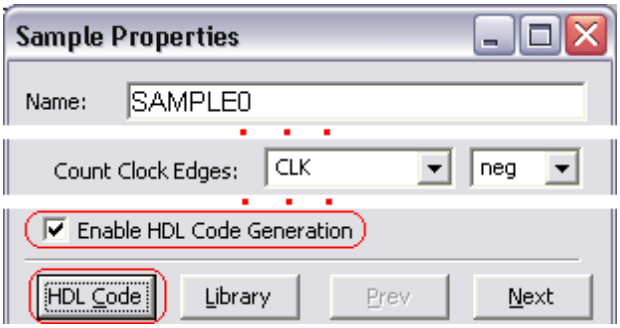
A file variable can be accessed using the **@FileName_out.ColumnName** syntax. One of the easiest ways is to let a sample do the coding for you. For output files, the data captured in the file output variable (one line of data in the file) is written at the end of any transaction that references the file variable.

Copyright © 2014, SynaptiCAD Sales, Inc.

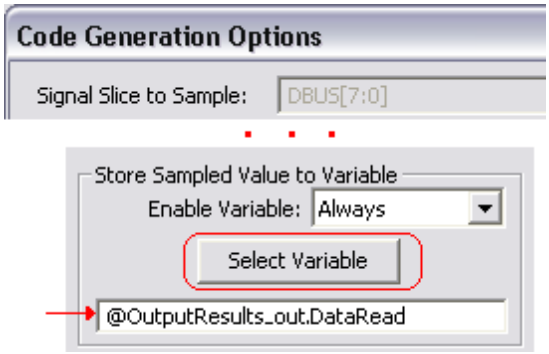
- Put a Sample parameter on the signal or bus whose value is to be written to the output file.
- In this example, we used two samples: one for the data bus and one for the address bus.



- Double click on the Sample to open the *Sample Properties* dialog.
- Check the **Enable HDL Code Generation** box.
- Press the **HDL Code** button to open the *Code Generation Options* dialog.



- Choose an **Enable Variable** setting like **Always, Else,** or **Then** (but not **Never**).
- Press the **Select Variable** button to select the variable that you want to store the bus value to.



- As a transaction executes, sampled values are stored into the the file variable. Each time the transaction completes, the stored values will be written to the output file.
- After simulating the test bench, you can view the output file by double clicking on the file name under the **File Output** folder in the Project window.

How to force a line to be written to the output file:

By default, only one line is written to an output file when a transactor finishes executing an Apply call. If you want to write more than one line to the file during a single transactor Apply call (e.g. you want to write multiple values inside a for-loop), add a HDL Code time marker that calls a function like this:

For VHDL:

Copyright © 2014, SynaptiCAD Sales, Inc.

```
WriteToFile(filename_out, filename_out_handle);
```

For Verilog:

```
filename_out.WriteToFile(testbench.filename_out_handle);
```

This will write the current values in the file variable output structure (presumably these values should have been previously set by some samples or other code constructs in the transactor) to the output value.

The reason we don't just make a write call after each sample sets a value in the file output variable structure is that it is quite common that the various fields of the structure need to be captured at different times during the transaction. For example, the sample that captures an address value could execute during one clock cycle and the sample that captures the associated data could execute in a subsequent clock cycle after some acknowledgment signal is received.

6.9 Input Data Dynamically from a File (VHDL)

VHDL test benches can read data from a file during simulation. This is not supported in Verilog, but there is a way to mimic this feature by initializing an array during the test bench build process as described in Section 6.10 (Verilog) Initialize Array from File 102 or Section 6.11 (VHDL) Initialize Array from File 104.

To read data from a file, first create a Test Vector data file and put it in the **File Input** folder. TestBencher will parse the file and create a project-level variable of type **File Input** to read from the file. This variable is defined by a class with fields (one field per column in the test vector spreadsheet file). Inside a timing diagram, the ***@FileName_in.ColumnName*** notation can be used to access the file variable in waveforms, delays, setups, holds, markers, and the conditional and action fields of Samples. Each time a diagram transaction is applied, TestBencher will read the next data line from the input file.

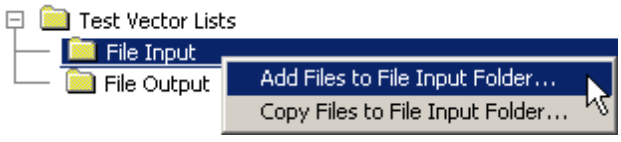
1) Create a data file:

Create a tab separated text file with data as described in Section 6.7 SynaptiCAD Test Vector File Format 95 using a spreadsheet program or a timing diagram. The data values read from the test vector file during the simulation run are not allowed prefixes that indicate the radix type (e.g. A4 is allowed for a hex value in the file, but 'hA4 is not). Below is an example of a legal test vector input file:

```
[Vectors]  Radix=hex
Addr       Data[7:0](bin)
A4         10110011
54         11011010
....
```

2) Add Test Vector File to project to Automatically create file class and variable:

- Under the **Test Vector Lists** folder, right click on **File Input** to open the context menu.

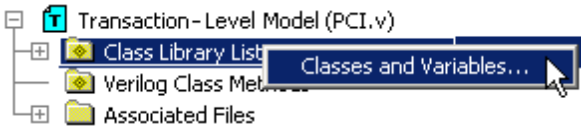


- Choose one of the menu options to attach the file to the project and create the file class and variables. The **Add Files to** menus will add the file and path to the project. The **Copy Files to** will make a local copy of the file in the project directory.

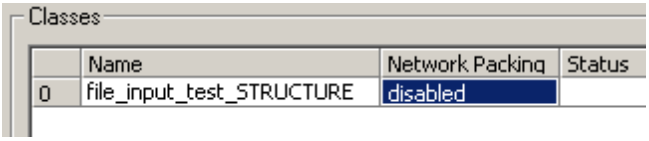
Copyright © 2014, SynaptiCAD Sales, Inc.

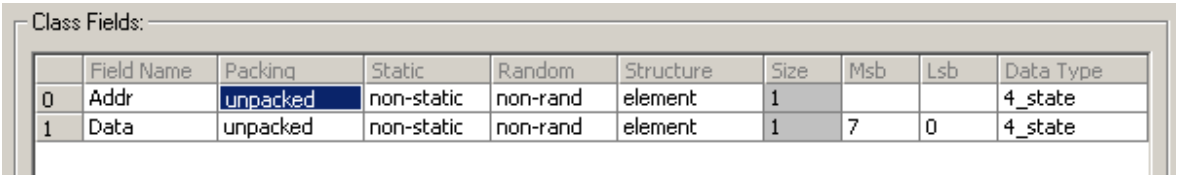
Investigate the file class and variable that was created:

- Right click on the **Class Library List** folder and choose **Classes and Variables** to open the *Classes and Variables* dialog.

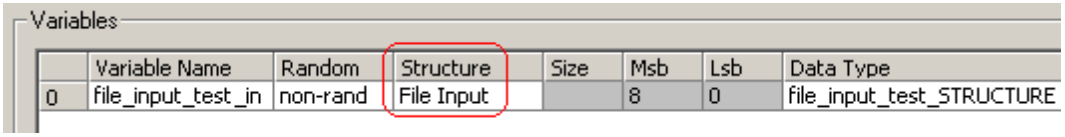


- In the **Class Definitions** tab, notice that a class was created and named ***filename*_STRUCTURE**. The fields of the structure are the column headings of the file.

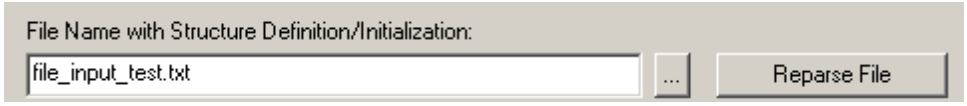




- In the **Variables** tab, notice that a file variable named ***filename*** was created. Its Structure type will be **File Input**. The Data Type will be the class that defines the fields of the file, ***filename*_STRUCTURE**.



- At the bottom of the Variables tab is the **File Name with Structure Definition/Initialization** box that shows the name of the associated file.
- If you make changes to the input file's header (e.g. signal names, signal sizes, etc), select the row containing the File Input variable and press the **Reparse File** to force TestBencher to update the class and variables associated with this file.



3) Accessing the File Variable:

Once a file variable is created at the project level, its fields can be accessed in any of the project's timing diagrams. Use the ***@filevariablename_in.fieldname*** to read or write to a field. This code can be entered anywhere a variable can be used, such as in a waveform state (Section 2.4 Waveform States and Conditions 36), samples (Section 3.2 Sample Checking and Actions 51), parameters (Chapter 4: Transaction Delays, Setups, and Holds 61), and in markers (Section 5.6 HDL Code Markers 77).

Causing an input line to be read from a file:

- **Apply a Timing Diagram:** If an input file variable is accessed within a timing diagram transactor, one line will be read from the file into the file variable each time the transactor begins a transaction. If the file variable is referenced again in the same transaction, the values will come from the original read of the file. The next time a transactor is called that references the file variable, the next line will be read from the file.
- **Force a Read:** By default, only one line is read from a file when a transactor is triggered by an

Copyright © 2014, SynaptiCAD Sales, Inc.

Apply call that references an input file variable. So even if you put the reference inside a loop in a transactor, the same value will be used on each iteration of the loop, by default. If you want to read another line from the file inside a loop in a transactor, you will need to tell the transactor to read the next line from the file. One way to do that is to add an "HDL Code" time marker inside the loop and have it execute a call similar to this:

```
ReadFromFile(filename_in, filename_in_handle);
```

This says read the next line from the file into the *filename*_in file variable. This will fill the fields of this structure with the new address and data values from the next line in the file.

For output files, a line of data is written at the end of any transaction that references the file variable.

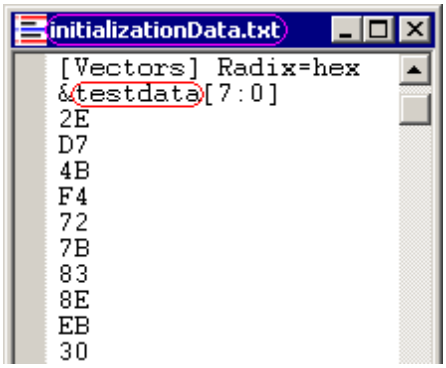
6.10 (Verilog) Initialize Array from File

In Verilog, Project-level array variables can easily be initialized with data from a file during the test bench generation process. For VHDL, see Section 6.11 (VHDL) Initialize Array from File 104 (the way to access the variables differs between the languages). The data is read from the file during test bench generation (not at simulation time) and written in the code of the top level test bench. This technique is very useful for Verilog test benches, because the Verilog code generator does not support reading file input during simulation as VHDL does (i.e. Section 6.9 Input Data Dynamically from a File (VHDL) 100).

In Verilog, you will first add the data file to the **Initialized Variables** folder, then press the **Make TB** button to generate the code. Inside a timing diagram, use ***@columnname*[index]** notation to access the file variable, where *index* is the row number of the data to use.

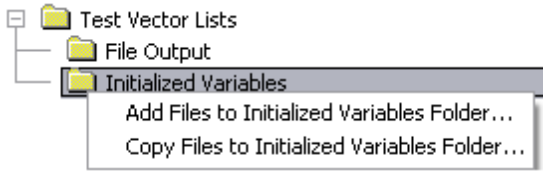
1) Create a data file:

- The data file should be in the spreadsheet file format with ***only one column*** that contains the data for the array. The file format is described in Section 6.7 SynaptiCAD Test Vector File Format 95. Note that the column name should NOT be the same as a signal name used in the a diagram. The column name will become the variable name and needs to be unique.



2) Create a project-level array variable linked to the data file:

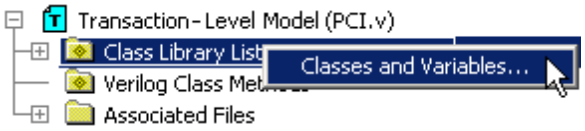
- Under the **Test Vector Lists** folder, right click on **Initialized Variables** to open the context menu. Put the data file in this folder using either the **Add...** menu which adds the file without moving it, or the **Copy...** menu which copies the file to the Project directory and then adds it.



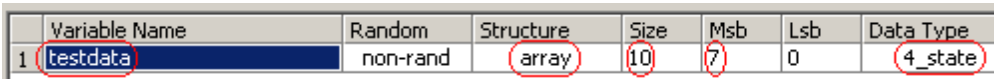
Investigate what got created when you added the file:

Copyright © 2014, SynaptiCAD Sales, Inc.

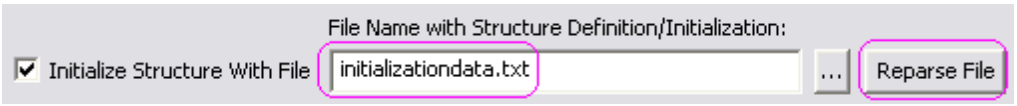
- Right click on the **Class Library List** folder and choose **Classes and Variables** to open the *Classes and Variables* dialog.



- Select the **Variables** tab, and notice that a variable has been created that is named after the **column name** of the data in the file. The Structure is of type **array**, the size indicates the number of rows of data, and the MSB/LSB/Data Type should match the information in the file.

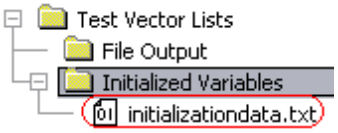


- At the bottom of the dialog, notice that **Initialize Structure with File** box is set and the file name is listed.



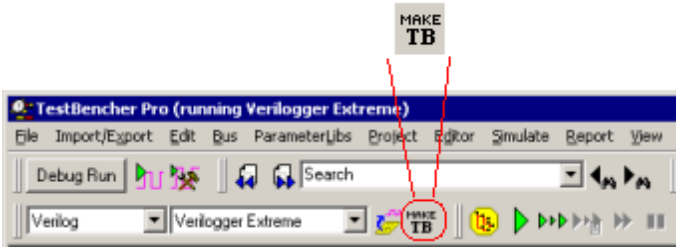
- If at some point the data file column name or size is changed, you can use the **Reparse File** button, located at the bottom of the dialog, to update the file variable.

- Note: After adding the file, you can double click on the file name to view the data or edit it.



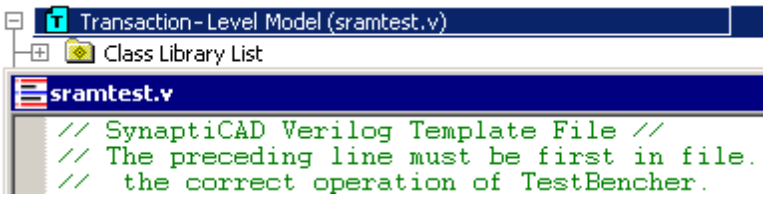
3) Generate the Test Bench:

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



Investigate the code that was generated:

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



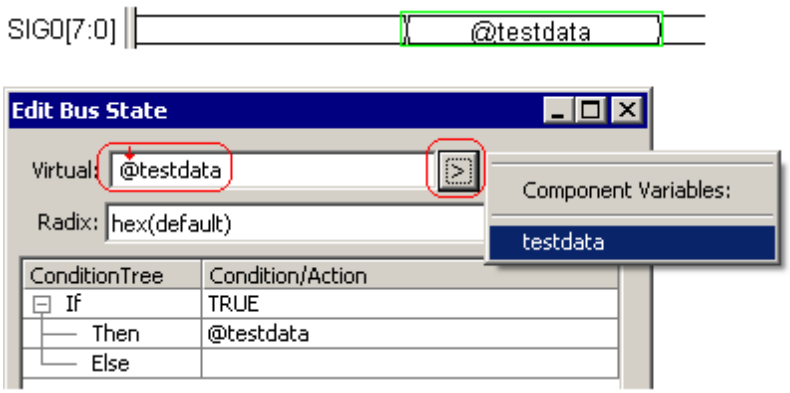
Copyright © 2014, SynaptiCAD Sales, Inc.

- Search for the variable that you attached to the file. The code generation should look something like the image shown.

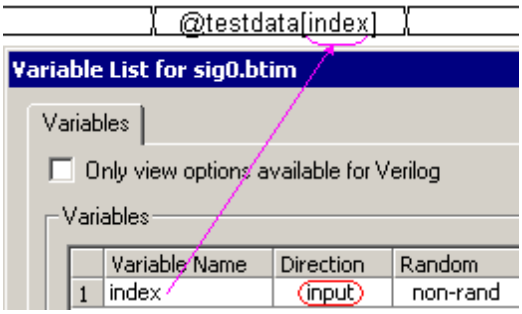
```
// $MiscFunctions
task InitializeVariables;
  begin
  testdata[0] = 8'h2E;
  testdata[1] = 8'hD7;
  testdata[2] = 8'h4B;
  testdata[3] = 8'hF4;
  testdata[4] = 8'h72;
  testdata[5] = 8'h7B;
  testdata[6] = 8'h83;
  testdata[7] = 8'h8E;
  testdata[8] = 8'hEB;
  testdata[9] = 8'h30;
  end
endtask
// End $MiscFunctions
```

4) Reference the Variable inside the Diagram

- Double click on a waveform segment to open the *Edit Bus State* dialog.
- Use the **>** button to find the file variable or type it into the **Virtual** edit box. Make sure to use the @ symbol to indicate that it is a project level variable.



- Since this is an array, you must add an index to the variable name. For example, the index can be a hard-coded value like "4" in **@testdata[4],** a variable that can be passed into the diagram each time it is called via the Apply call, or an index variable created by a Marker-generated for-loop.
- To make a variable that is an input to the timing diagram transaction, press the **View Variables** button on the diagram window and create a variable that has a direction of **input**. Then use this variable as the index to the array variable.



6.11 (VHDL) Initialize Array from File

In VHDL, project-level array variables can be initialized with data from a file during the test bench generation process. The data is read from the file during the test bench generation (not at simulation time) and written in the code of the top level test bench. To dynamically read from a file during simulation instead, use the technique described in Section 6.9 Input Data Dynamically from a File (VHDL) 100).

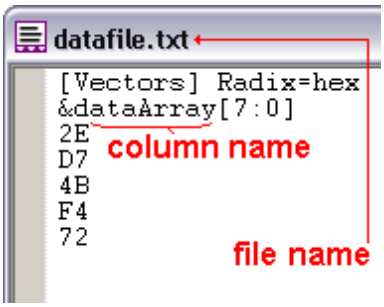
In VHDL, you will first add the data file to the **Initialized Variables** folder, then press the **Make TB**

Copyright © 2014, SynaptiCAD Sales, Inc.

button to generate the code. Inside a timing diagram, use **@*filename*_init.*columnname*[index]** notation to access the file variable, where *index* is the row number of the data to use.

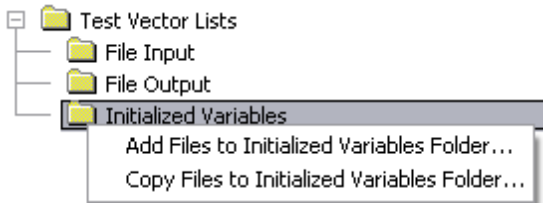
1) Create a data file:

- Create a data file in the *SynaptiCAD Test Vector File Format* as described in Section 6.7 SynaptiCAD Test Vector File Format 95.
- For VHDL, the file should contain **one or more** tab separated columns with data. Each column will represent one array field in a structure. Each field will be an array with the size of the arrays being set by the number of rows in the file.



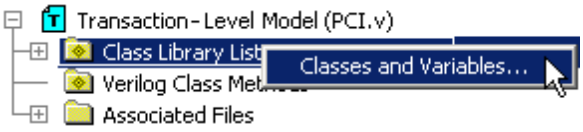
2) Add file to the Initialize File Folder to automatically create the class and variables:

- Under the **Test Vector Lists** folder, right click on **Initialized Variables** to open the context menu. The **Add...** menu adds the file without moving it, and the **Copy...** menu copies the file to the Project directory and then adds it.

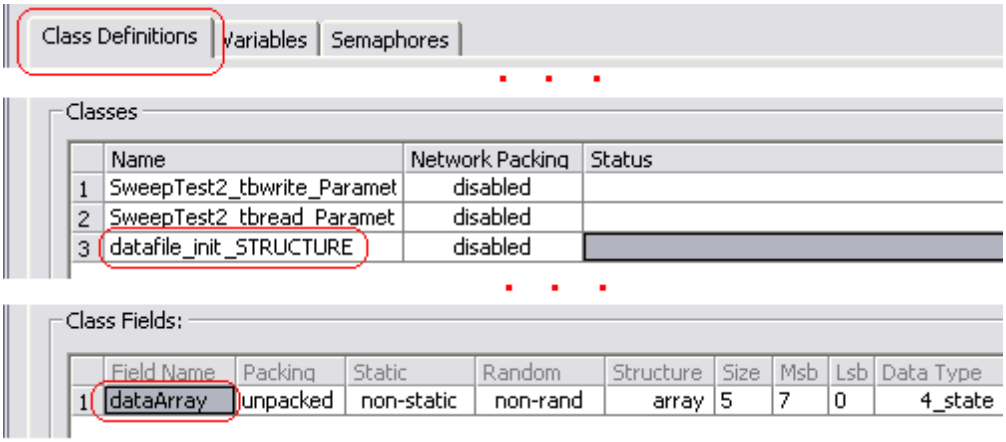


Investigate what got created when you added the file:

- Right click on the **Class Library List** folder and choose **Classes and Variables** to open the *Classes and Variables* dialog.

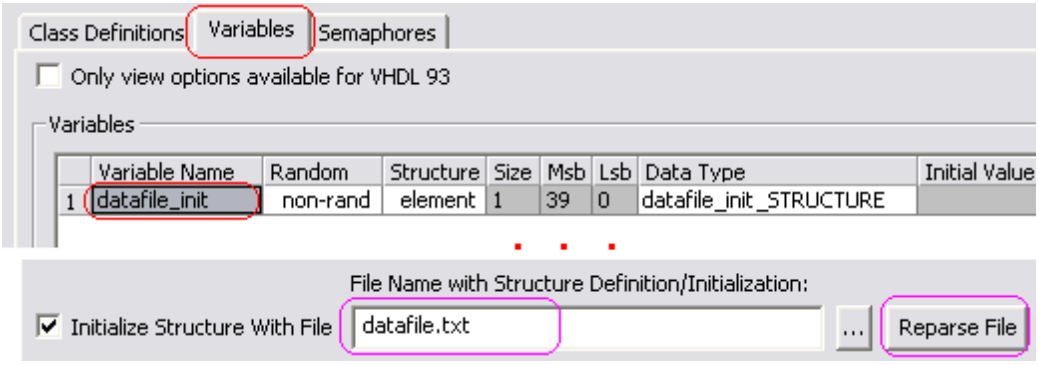


- Under the **Class Definitions** tab, notice there is a class called ***filename*_init_STRUCTURE**.When you select the class, notice that the class fields are named after the ***columnnames***.



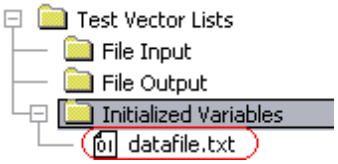
- Click on the **Variables** tab and notice that there is an project-level instance of the file class named after the ***filename***. Inside a timing diagram, you can access this variable using the **@*filename*_init.*columnname*[index]** notation, where *index* is the row number of the data to use.

Copyright © 2014, SynaptiCAD Sales, Inc.



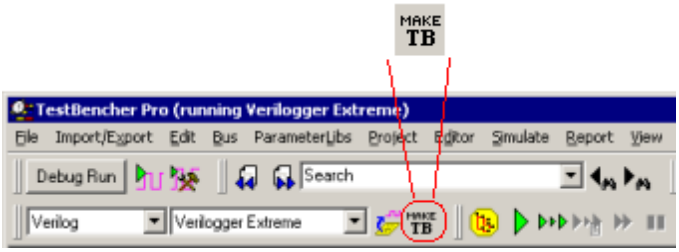
- If at some point the data file column names are changed, you can use the **Reparse File** button, located at the bottom of the dialog, to recreate the class and file column names.

- Note: After adding the file, you can double click on the file name to view the data or edit it.



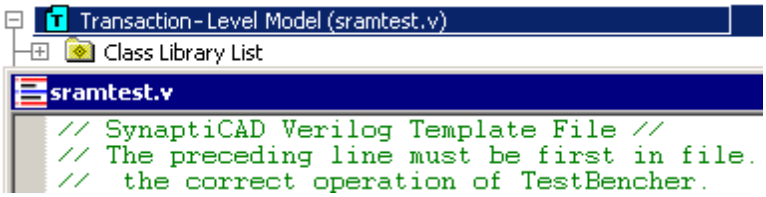
3) Generate the Test Bench:

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



Investigate the code that got generated:

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



- Search for one of the column names. The code generation should look something like the image shown.

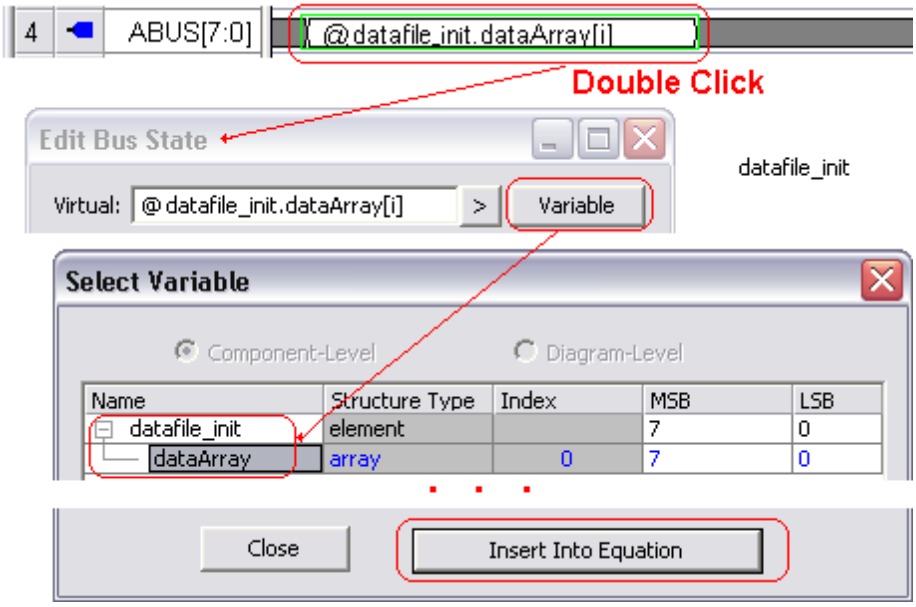
```
-- $MiscFunctions
procedure InitializeVariables(tb_ProjectID : inout natural) is
begin
 tb_ProjectVariables(tb_ProjectID).datafile_init.dataArray(0):=x"2E";
 tb_ProjectVariables(tb_ProjectID).datafile_init.dataArray(1):=x"D7";
 tb_ProjectVariables(tb_ProjectID).datafile_init.dataArray(2):=x"4B";
 tb_ProjectVariables(tb_ProjectID).datafile_init.dataArray(3):=x"F4";
 tb_ProjectVariables(tb_ProjectID).datafile_init.dataArray(4):=x"72";
end procedure;
-- End $MiscFunctions
```

file name column name

Copyright © 2014, SynaptiCAD Sales, Inc.

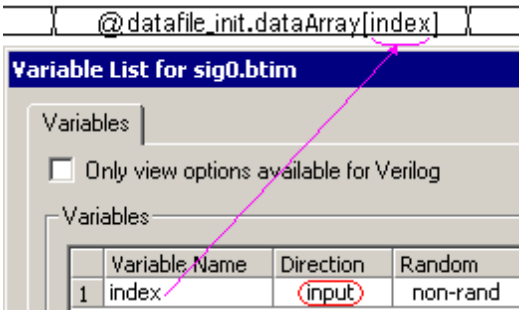
4) Reference the Variable.ColumnName inside a Diagram

- Double click on a waveform segment to open the *Edit Bus State* dialog, then press the **Variable** button to open the *Select Variable* dialog.



- In the *Select Variable* dialog, highlight the column name element and press the **Insert Into Equation** button to insert the file variable into the waveform.
- The array will come in with an index of 0, so you will probably want to replace the 0 with a variable name that can access the different elements of the data array.

- To make a variable that is an input to the timing diagram transaction, press the **View Variables** button on the diagram window and create a variable that has a direction of **input**.
- This variable will be an input in the Apply call to the timing diagram, so you can set the index each time the diagram is called.



6.12 Variables 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 6.1: Variables 82 and Section 6.2 Classes 86. The section defines the different properties. Certain property settings, such as 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.

Variable Properties:

- **Variable Name** or **Field Name** is used when referencing the variable or field of the class.
- **Direction** (Variable Only) determines whether or not the variable is an input to the diagram or

Copyright © 2014, SynaptiCAD Sales, Inc.

component. An input diagram-level variable will become an input to the transaction apply call, just like a state value. An input project-level variable will become an input to the project Transaction-Level Model, and each instance of the component can specify an initialization value for the input 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 6.6: Data Packing 94.
- For Verilog, the **Network Packing** property reverses the byte order of the class after 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, rand, and randc (random cyclic)*. The *non-rand* selection indicates that the variable is not randomized. If it is set to *rand or randc,* 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 Transaction-Level Model. The generation language may limit the options that are available for randomization. Section 6.5 Constrained Random Number Generation 92 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 listed in Section 6.13 Language Independent Types 109 (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, then it will be converted to the appropriate type for the generated language in the generated test bench.
- **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 project 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** is a single data item (like a single integer).

Copyright © 2014, SynaptiCAD Sales, Inc.

- **Array** is 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 6.9 Input Data Dynamically from a File (VHDL) 100). 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.

6.13 Language Independent Types

SynaptiCAD has defined a set of language independent types that is used by TestBencher's graphical interface 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. 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.

Description of SynaptiCAD Language Independent Types

SynaptiCAD 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
int	32	Signed integer represented by 32 bits

Copyright © 2014, SynaptiCAD Sales, Inc.

unsigned_int	32	Unsigned integer represented by 32 bits
real	64	Floating point 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 and the generated language native types. Cells that are grayed out represent items where no conversion is available between the language independent type and the native language types.

SynaptiCAD	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	N/A
4_state	reg	std_logic	tbvSmartSignal4StateT
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	
actel_current_delta	reg	std_logic	
actel_temperature	reg	std_logic	
actel_voltage	reg	std_logic	

Copyright © 2014, SynaptiCAD Sales, Inc.

actel_voltage_common	reg	std_logic	
actel_voltage_delta	reg	std_logic	

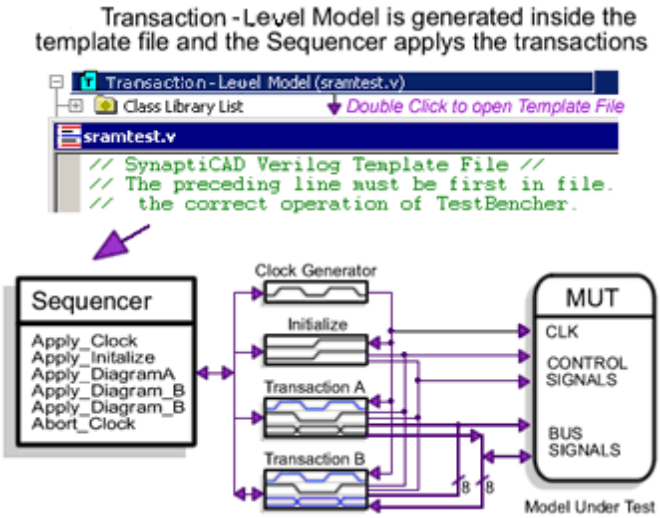
Note that not all language types are perfectly equivalent to the SynaptiCAD type. Variances are as follow:

- 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.

Copyright © 2014, SynaptiCAD Sales, Inc.

Chapter 7: Transaction-Level Model and Transaction Sequencer

The Transaction-Level Model is the top-level control file for the project. For top-level projects, this is the place where the transactors are instantiated and where the model under test is instantiated. For sub-projects, this is the place where any initialization class methods are called.



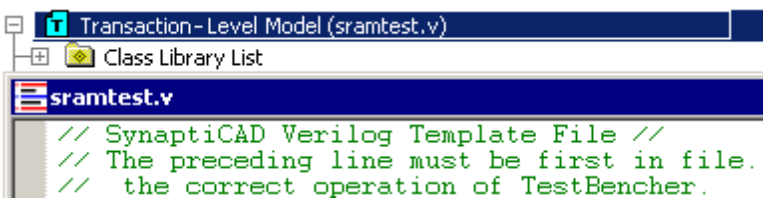
The template file also contains the **Transaction Sequencer** which is the area where you will specify how transactions should be applied to the model under test. Transactions can be called directly using the ***Insert Diagram Subroutine Calls*** dialog (Section 7.1: Template, Sequencer, and Transaction Calls 112), read from a file using the ***Transaction Manager*** queue (Section 7.3: Transaction Manager Overview 115), or randomly generated using the ***Transaction Generator*** (Section 7.6: VHDL: TM fill from Random Generator 121). Structural calls such as loops and conditional statements can be added to control test execution and generate data. 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. Apply calls are supported in all generation languages, but the transaction manager, generator, and monitor are limited to specific languages that support advanced programming statements.

7.1 Template, Sequencer, and Transaction Calls

The **template file** is a language-specific source code file where both the user and TestBencher insert source code for the Transaction-Level Model. TestBencher generates the Transaction-Level Model code and puts it into macros contained in the template file. The file also contains the **Transaction Sequencer** which is where transaction Apply and Post calls and most of the other user-written code for the test bench will be added by the user.

Open the Template file to edit the Transaction-Level Model and the Sequencer Process

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



Copyright © 2014, SynaptiCAD Sales, Inc.

Don't write code inside Macro Statements

Source code can be added anywhere in the template file, except for between two macro statements. The macro statements are the locations where TestBencher will place generated code. Each time the bus-functional model is generated, any code within the macro lines is overwritten. An example of a macro keyword pair is:

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

Writing Code for the Transaction Sequencer

The most common place to add source code is in the sequencer process. This process is the area where diagram calls are placed. You can also write source to define variables, call randomize functions on variables, and define loops and other control structures. You can also write code to place conditions on whether or not a diagram call is made based on information returned from previous transaction calls.

- Scroll down in the template file until you find the comment block that contains the words **Transaction Sequencer**. The comment will look something like the following (it changes depending on the generation language):

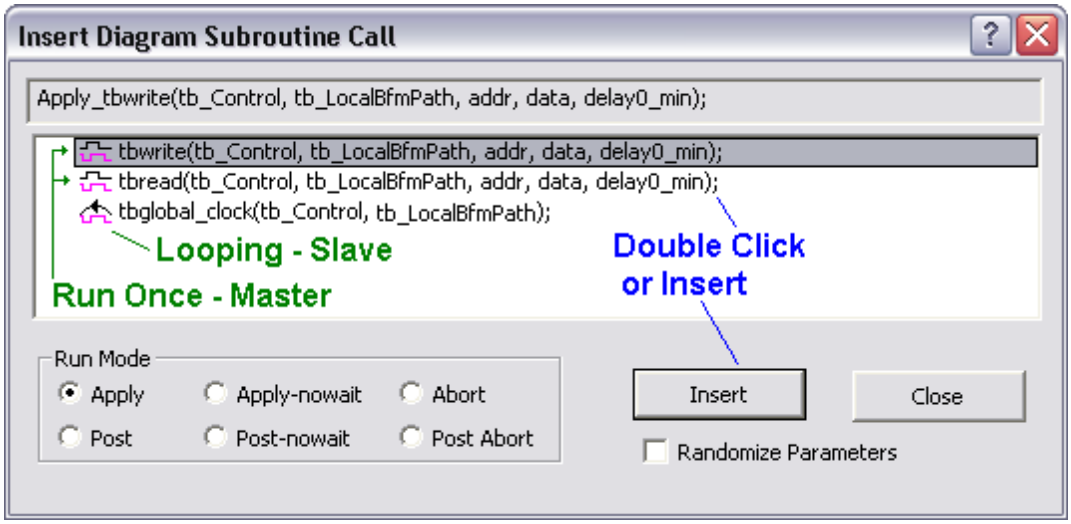
```
//*********************************************************
// Transaction Sequencer - After this comment, define how to
//    apply transactions to the model under test using:
```

Since the sequencer process is outside of any macro statements, code added here will be preserved during the code generation process.

Use the Insert Diagram Subroutine Call dialog to make Transaction Calls:

The fastest way to write Transaction Call statements is to use the *Insert Diagram Calls* dialog as described in Step 5: Define a Sequencer Process 17. This dialog will generate diagram apply calls so you do not need to memorize the function syntax. Chapter 11: Language Specific Details 170 has all of the language specific call details.

- To open the *Insert Diagram Subroutine Calls* dialog, put the cursor in the template file on the line to insert, and then right click and choose **Insert Diagram Calls** from the context menu.



- Double clicking on a diagram name inserts the normal Apply call for the type of transaction, or use the **Insert** button to apply with the radio buttons.

Copyright © 2014, SynaptiCAD Sales, Inc.

- If the transaction has parameters, you will have to edit the parameter variables in the inserted call.

Each timing diagram generates the following tasks to run or schedule transactions for that diagram:

- **Apply** runs a transaction immediately in a blocking mode (the statement after this Apply will not execute until this transaction completes).
- **Apply-nowait** runs a transaction immediately (this task returns immediately after triggering the transaction, so subsequent statements in the sequencer will execute before simulation time advances).
- **Post** schedules a transaction on a queue for future execution. See Section 7.4 VHDL: TM fill with Post Calls 117. No simulation time passes during the execution of the Post call itself, but the scheduled transaction will run in a blocking mode.
- **Post-nowait** schedules a transaction on a queue for future execution. No simulation time passes during the execution of the Post call itself. Unlike a regular **Post** call, the scheduled transaction will also be run in a non-blocking mode.
- **Abort** stops an executing transaction. Mainly used to stop slave transactions (e.g. clock generator) at the end of a test sequence.
- **Post Abort** schedules an **Abort** call on a queue. The specified transaction will be stopped when the Abort call is popped off it's queue.
- **Randomize Parameters** will randomize the input parameters for Apply and Post calls. See Section 7.7 Randomizing Transaction Inputs 124.

Master and Slave Transactors

The Master/Slave setting in the *Diagram Settings* dialog determines how many times a transaction executes (see Section 9.8 Diagram Settings Dialog 148). Master Transactors run once and stop. Master transactors display as an icon with waveforms in the *Insert Diagram* dialog. Slave transactors (e.g. a clock generator diagram) run in a looping mode. Slave transactors display as an icon with waveforms and a loop-back arrow to indicate they run repeatedly.

Example of Transaction Sequencer code

The following is a Verilog example of a sequencer process that performs a sweep test on an SRAM. The variable *delay0* 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 controls the timing of a waveform edge in the Write and Read transaction. Variables could have been defined for the transactions, but we chose to pass in values. The transaction apply calls were written using the *Insert Diagram Calls* dialog as described above. 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
```

Copyright © 2014, SynaptiCAD Sales, Inc.

Switching Languages and Template Files

Since the template file is language-specific, if you need to change the generation language for a project after the project has already been created, the template file must also be changed using the technique described in Section 7.8: Changing a Project Template File 126.

Creating new Template Files

The template file is a copy of a file that was specified in the *Project Wizard* when the project is first created (Section 9.9: Project Menu and New Project Wizard 151). Normally you will use the default TestBencher template file, but you can also develop your own templates (this is not generally necessary). When developing your own templates, it is important to copy all of the Macros so that TestBencher will know where to put the generated code.

7.2 Master and Slave Transactions

Many bus specifications differentiate between master and slave devices. A master device initiates transactions on a bus and a slave device performs an action in response to a transaction initiated by a master device. Master transactions are used to model transactions initiated by a master device, where the user makes an Apply call to initiate a single transaction. Slave transactions, by contrast, run continuously after an Apply call is made, so they are typically designed to block until a master transaction is initiated to the slave device, respond appropriately, then resume waiting for another master transaction.

For TestBencher, any timing diagram can be a Master or a Slave transactor by appropriately setting the **Diagram Execution** radio buttons in the *Diagram Settings* dialog (see Section 9.8: Diagram Setting Dialog 148 for more information about the Diagram Execution settings). TestBencher generates appropriate transaction calls for each timing diagram based on this setting. The following table describes the typical operations available for master and slave transactions:

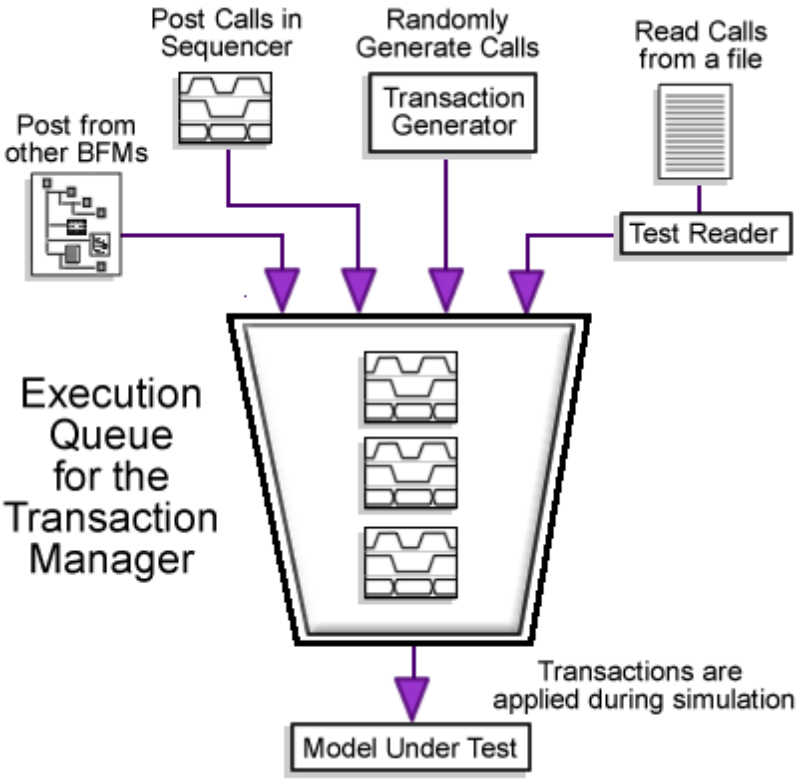
Desired Operation	Method call to initiate operation
Initiate a single master transaction	Apply
Execute pipelined master transactions	Consecutive Apply-nowait calls (pipeling controlled by pipeline markers within the diagrams)
Abort a master transaction	Abort
Start slave device or clock generator	Apply-looping_nowait
Stop slave device or clock generator	Abort

Section 7.1: Template, Sequencer, and Transaction Calls 112 has more information about the *Insert Diagram Subroutine Call* dialog. The **PCI example** in the **SynaptiCAD>Examples>TestBencher** 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.

7.3 Transaction Manager Overview

VHDL ONLY: In addition to immediately executing transactions via Apply calls in the template file's sequencer process, TestBencher can generate a transaction manager that maintains a queue of transactions scheduled for future execution. Transactions can be placed in this queue in several ways: using Post calls to add a specific transaction to the queue, using ApplyFile calls to read a list of transactions from a file, or generated randomly using a Transaction Generator. Transaction queues are currently only supported in VHDL.

Copyright © 2014, SynaptiCAD Sales, Inc.



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.

Hierarchical TLMs and Transaction Managers

If you create a hierarchical testbench with embedded TLMs (see Section 9.1 Sub-Projects 137), each project Transaction-Level Model can have it's own transaction manager. Any TLM sequencer can post a transaction to any other TLM's transaction queue. Most commonly, a top-level TLM model will generate test sequences and post them to it's child TLM transaction queues. All calls to transactors can be specified either as relative or fixed path, allowing any transactor to be referenced from anywhere in the project.

Filling up the Transaction Manager Queue:

After the Transaction Manager has been created, you can choose one of the following methods to fill the queue:

- Use **Post** calls in the sequencer process to add transactions to the queue. See Section 7.4: VHDL: TM fill with Post Calls 117 for details on call syntax.
- Use an **ApplyFile** call add a list of transactions stored in a file to the queue. See Section 7.5: VHDL: TM fill from File 120 section for the transaction list file format and call syntax.
- Use the Transaction Generator to **randomly apply transactions**. See Section 7.6: VHDL fill from Random Generator 121.

Transaction Manager Modes: Popping transactions from the Queue

The Transaction Manager can be run in different modes by calling a ***SetApplyCallMode*** function. By default, the mode is set to looping, in which case transactions are executed when they are available

Copyright © 2014, SynaptiCAD Sales, Inc.

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 the procedure **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.

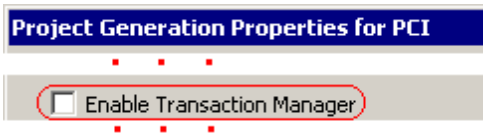
Disabling Transaction Manager Generation:

If you want to create the simplest possible test bench code and your test bench doesn't use transaction manager features (e.g. it doesn't post transactions to a queue), you can disable the generation of transaction manager code:

- Right click on the **Transaction-Level Model** folder and choose **Project Generation Properties** from the context menu to open the *Project Generation Properties* dialog.



- Uncheck the **Enable Transaction Manager** checkbox and press **OK** to close the dialog.



7.4 VHDL: TM fill with Post calls

One Method of filling the Transaction Manager Queue is to post diagram calls to the queue. This is similar to using a normal diagram apply calls, but the transaction is scheduled for execution in the future (when it gets popped from the queue), instead of at the present time.

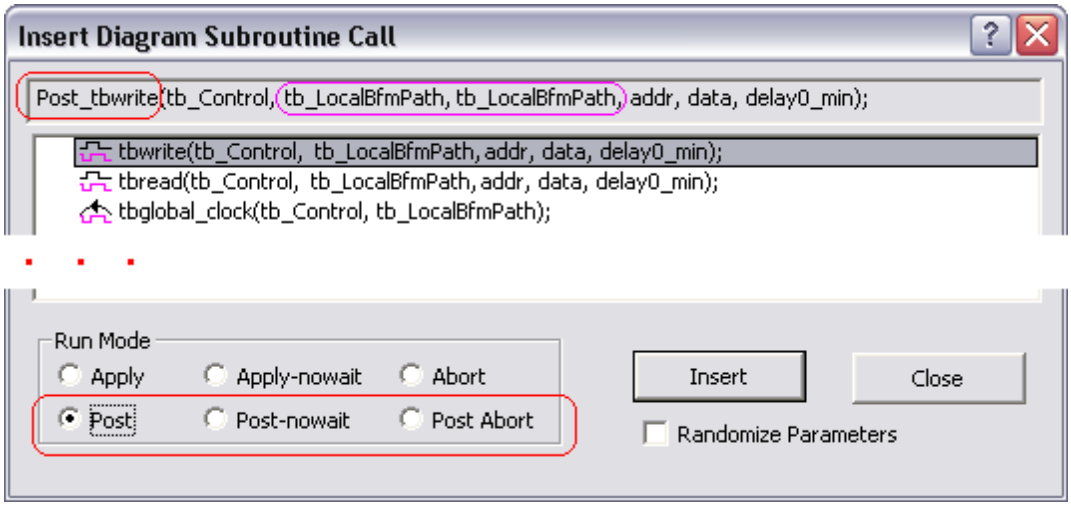
Post diagram calls require an extra parameter called **queuePath** which specifies the path to the transaction manager where the transaction will be scheduled for execution. Typically, queuePath will be set to **tb_LocalTLMPath,** which is a generic defined by each TLM that sequencers and transactor-level code can access to reference their containing TLM. Non-hierarchical TLMs will always use tb_LocalBfmPath.

TestBencher will only create the Post transaction calls after the Transaction Manager has been enabled and built as shown in section Section 7.3: Transaction Manager Overview 115. Procedures for posting transactions individually to the transaction manager (defined in tb_*projectName*_tasks.vhd).

Using a dialog to generate the Post calls:

- To open the *Insert Diagram Subroutine Calls* dialog, put the cursor in the template file on the line to insert, and then right click and choose **Insert Diagram Calls** from the context menu.

Copyright © 2014, SynaptiCAD Sales, Inc.



- Select one of the Post radio buttons and select a transaction to highlight it, then press the **Insert** button.

Master Transaction Manager Post Calls:

```
Post_diagramName ( signal tb_Control : inout tb_Control_Type;
                   queuePath : in string;
                   transactorBfmPath : in string;
                   <parameters> );
Post_diagramName_nowait ( signal tb_Control : inout tb_Control_Type;
                   queuePath : in string;
                   transactorBfmPath : in string;
                   <parameters> );
Post_Abort_diagramName ( signal tb_Control : inout tb_Control_Type;
                   queuePath : in string;
                   transactorBfmPath : in string );
```

Slave Transaction Manager Post Calls:

```
Post_diagramName_looping ( signal tb_Control : inout tb_Control_Type;
                   queuePath : in string;
                   transactorBfmPath : in string;
                   <parameters> );
Post_diagramName_looping_nowait ( signal tb_Control : inout tb_Control_Type;
                   queuePath : in string;
                   transactorBfmPath : in string;
                   <parameters> );
Post_Abort_diagramName ( signal tb_Control : inout tb_Control_Type;
                   queuePath : in string;
                   transactorBfmPath : in string );
```

Example Post calls

Below are some examples of posting transactions to transaction queues. To post a transaction, you must specify the TLM containing the queue (transactionManagerPath) and the TLM containing the transactor that will perform the transaction (transactorInstancePath). If the transactor is not defined in the local TLM that is making the Post call, you will need to specify the package containing the transactor's Post procedure (e.g. work.tb_master_tasks in the example below).

If you have designed a hierarchical test bench project, two situations will commonly arise where you will need to a use different value for queuePath:

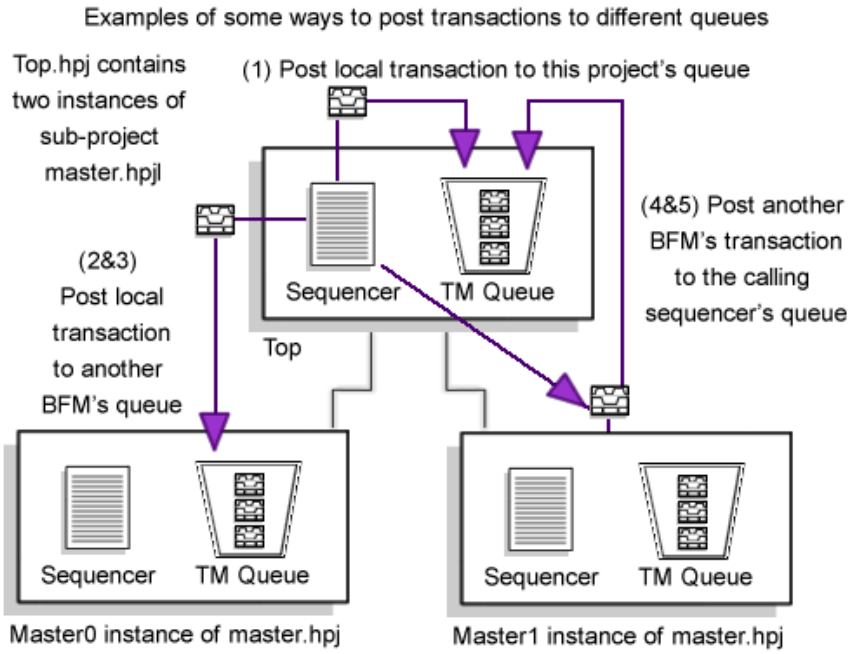
Copyright © 2014, SynaptiCAD Sales, Inc.

- If you need to post a call to a sub-TLM of the TLM containing the Post call, you should use a relative path to the sub TLM:

```
queuePath := tb_LocalBfmPath & ".sub0"
```

- If you need to call a TLM that is not a sub-TLM of the TLM containing the Post call, then you will have to use the full VHDL hierarchical name to post to that TLM's queue:

```
queuePath := "top.master0"
```



- **Example 1)** Post a transaction defined in this local TLM (i.e. local project) to the local transaction queue:

```
Post_write( tb_Control, tb_LocalBfmPath,
                        tb_LocalBfmPath, x"AB", x"EF");
```

- **Example 2)** Post transaction defined in this local TLM to the queue of a sub-TLM ("master0"):

```
Post_write( tb_Control, tb_LocalBfmPath & ".master0",
                        tb_LocalBfmPath, x"AB", x"EF");
```

- **Example 3)** Same as example 2, except using a full hierarchical name ("Top.master0"):

```
Post_write( tb_Control, "Top.master0",
                        tb_LocalBfmPath, x"AB", x"EF");
```

- **Example 4)** Post a transaction defined in a sub-TLM (sub-TLM is called "master1") to the local TLM's queue by specifying master1's path relative to the local TLM:

```
work.tb_master_tasks.Post_write( tb_Control, tb_LocalBfmPath,
                     tb_LocalBfmPath & ".master1", x"AB", x"EF");
```

- **Example 5)** Same as example 4, except using a full hierarchical path name to the transactor's TLM (in this example, the TLM Top contains the master1 TLM)

```
work.tb_master_tasks.Post_write( tb_Control, tb_LocalBfmPath,
                   "Top.master1", x"AB", x"EF");
```

Copyright © 2014, SynaptiCAD Sales, Inc.

7.5 VHDL: TM fill from a File

One method of filling a transaction manager queue is to read in the Post calls from a file. This method allows different test sequence files to be used in 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.

An input file can be created to specify the test sequence used by the Transaction Manager module.Each row contains a **transactor instance name**, a **transactor mode**, and the transaction's **input parameters**.

- **Transactor Instance Name** is the name of the transactor and the relative path from the top-level module to the TLM instance that contains the transactor. Some options are:
  - transactorName
  - subProjectInstanceName.transactorName
  - projectName.transactorName
  - projectName.subProjectInstanceName.transactorName

```
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
```

- **Transaction Mode** is an integer that specifies how the transaction should be executed. The different types of modes are shown below.
  - **0 => Apply** The transaction will block the transaction manager from executing any more transactions until the transaction completes.
  - **1 => Apply_nowait** The transaction will not block the transaction manager: the next transaction call in the queue will immediately execute.
  - **2 => Abort** Aborts the specified transactor if it is currently executing.
- **Input Parameters** TestBencher will read in the appropriate number and type of parameters based on the transaction type.

Once a test sequence file is created, the file can be passed to a transaction manager queue by calling either the **ApplyFile** or **ApplyFile_nowait** method. 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. The transaction manager will read a new line from the test sequence file whenever the transaction manager's queue becomes empty.

ApplyFile will not return control to the calling process until all transactions have been read from the file. By contrast, 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 a transaction manager.

Example Test Bench Using ApplyFile calls

There is an example project (VME.hpj) that demonstrates how to use ApplyFile calls in a hierarchal TLM in the **Synapticad/Examples/TestBencher/VHDL/VME** directory.

Procedures for posting transactions from files (defined in tb_control_types.vhd)

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

Copyright © 2014, SynaptiCAD Sales, Inc.

ApplyFile will not return to the calling process until all transactions have been read from the test sequence file and placed into the transaction manager's queue. If the file only contains concurrent (nowait) transactions, then this function will return in zero simulation time since the transaction manager will start each transaction without blocking for the previous one.

```
ApplyFile_nowait ( signal tb_Control : inout tb_Control_Type;
                   fileName : string;
                   queuePath : 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 online documentation for actual implementation.

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

Sets the mode of the transaction manager. By default the mode is set to looping, meaning transactions will be 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( queuePath : 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.

7.6 VHDL: TM fill from Random Generator

TestBencher Pro provides functions for randomly posting master transaction calls to a transaction manager's queue, based on the probabilities specified in a weightings table for that transaction manager. The input data for these randomly selected transactions is randomly generated when a transaction is popped from the queue and executed. The randomly generated input values are constrained by the constraint settings for each input variable. This feature is currently only supported in VHDL.

Post Random Transaction Types:

Calls to **PostRandomTransactionType** will randomly generate one of the master transaction types of the specified TLM and add this transaction to the TLM's transaction manager queue. In converting a directed test bench project into a constrained-random test bench (where transactions are randomly applied to the model under test), you will remove most of the apply calls to master transactors and replace them with PostRandomTransactionType calls. The apply calls to slave transactors will probably remain the same.

Below is an example of code that was put into a sequencer process to generate random transactions:

Copyright © 2014, SynaptiCAD Sales, Inc.

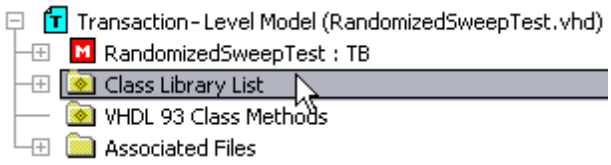
```
Apply_GClock_generator_looping_nowait(tb_Control, tb_LocalBfmPath);

for i in 0 to 5 loop
  PostRandomTransactionType(tb_Control, tb_LocalBfmPath);
end loop;

wait for 2000 ns;                                   Start and Stop
                                                      the Clock
Abort_GClock_generator(tb_Control, tb_LocalBfmPath);
```

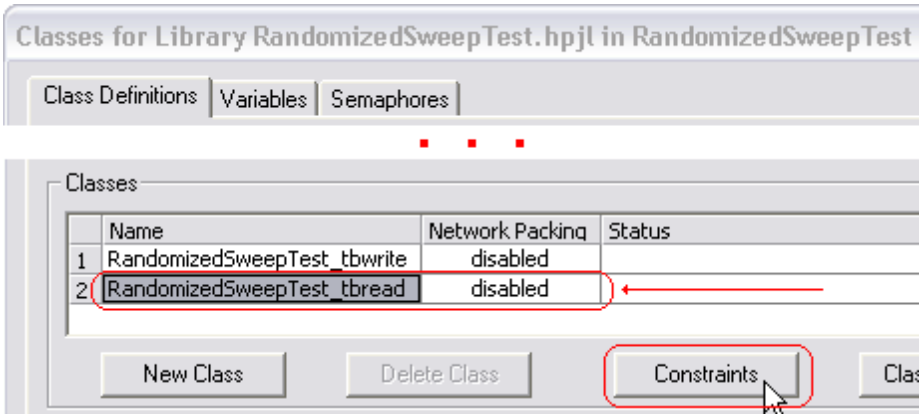
- Notice that the we started and stopped the clock using regular Apply calls. The clock generator is a slave transaction, so it is not eligible to be randomly generated by the Test Generator.
- The **PostRandomTransactionType** call puts one randomly chosen transaction into the Transaction Manager's Queue.
- There is a loop around the **PostRandomTransactionType** call so it will execute and put 5 transactions in the queue.
- There is also a **wait** call that will force the simulation to continue to run until all the transactions have occurred and the queue is empty (both the read and the write transactions in this example take 300ns to complete, so 5 of these transactions will complete in 1500ns).

Constrain the Random Input Data:

Each time a random transaction is executed, the input arguments for the transaction will be randomly generated, taking into account any constraint definitions for the transactor's input variables.

- Double click on the **Class Library List** under the **Transaction-Level Model** branch of the project to open the *Classes for Library* dialog.

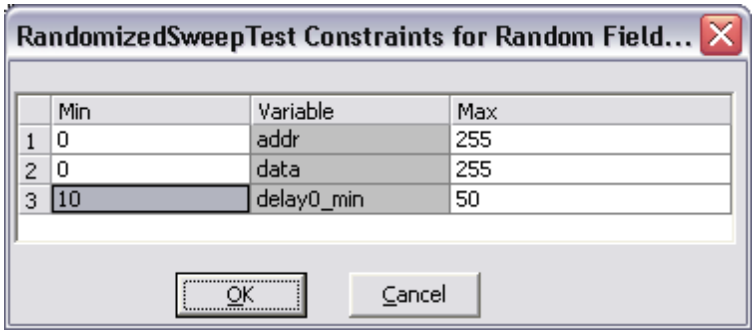




- Select a particular diagram in the **Class Definitions** tab and then press the **Constraints** button to view and edit the constraints. Each transactor diagram has its own set of input variables.

Copyright © 2014, SynaptiCAD Sales, Inc.

- Variables default to the entire legal range for their data types. Here we have edited the range for the delay values.

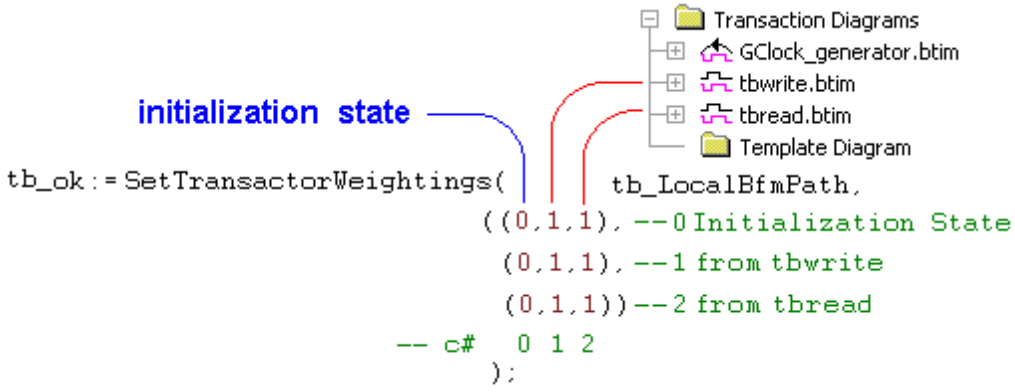


Setup up the Weightings Table:

A weightings table defines the relative probabilities for the next transaction type based on the type of the most recently posted transaction call. The default weightings table is set inside the **$InitializeDiagramTaskCall** macro at the top of the Sequencer Process. This default table gives a uniform weighting to all master transactions (each transactor has the same probability that it will execute after any other master transactor). The default table sets the probability of the initialization state to zero (the initialization state is not a transactor type, it just represents the initial state of the TLM before any transactor has been randomly generated).

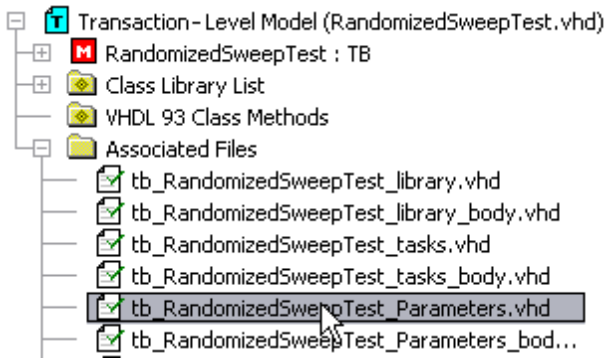
```
-- Sequencer process
process
 begin

-- ensure all diagrams initialize before applying them
-- $InitializeDiagramTaskCall
tb_initialize_RandomizedSweepTest(tb_ProjectID);
tb_ok := InitializeUniform(7, 7000);
tb_ok := SetTransactorWeightings(tb_LocalBfmPath,
                    RandomizedSweepTestUniformWeighting);
-- End $InitializeDiagramTaskCall
```

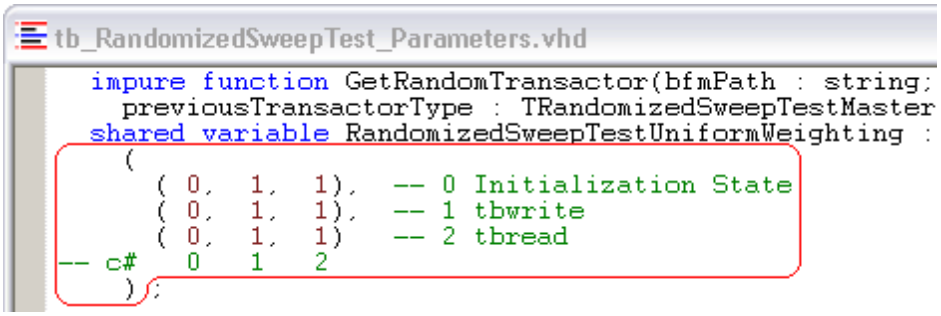
- To change the the weightings, copy the entire **SetTransactorWeightings** line from inside the above macro, and paste it below the macro. You can then edit this line to have your table values overwrite the default table during simulation.
- The ***ProjectName*UniformWeighting** term needs to be replaced with a state matrix, where the rows represent the last randomly generated master transactor type and the columns define the weighting for the next transactor type. The first row/column is reserved for the reset or starting state of the TLM. The order of the master transactions is the same as they appear in the Transaction Diagrams folder in the *Project* window.



Copyright © 2014, SynaptiCAD Sales, Inc.

- The table can be typed in or copied from the package file that defines the default weightings table. To copy the table, double click on the **tb_*ProjectName*_Parameters** file located in the **Associated Files** folder of the **Transaction-Level Model** to view the code.
- Search for **UniformWeighting** in the package file and copy/paste the table as an argument to a SetTransactorWeightings call in the sequencer.





- 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. The table will accept values from 0 to max integer. Here we made it just as likely that a read will follow a write or vise-a-versa by using 1 in all the spaces that do not involve the initialization state. Filling the last column with 2's would make it twice as likely to generate *tbreads* than *tbwrites*. The first column should always contain zeros, because it means that the test bench should not randomly enter the test bench initialization state.

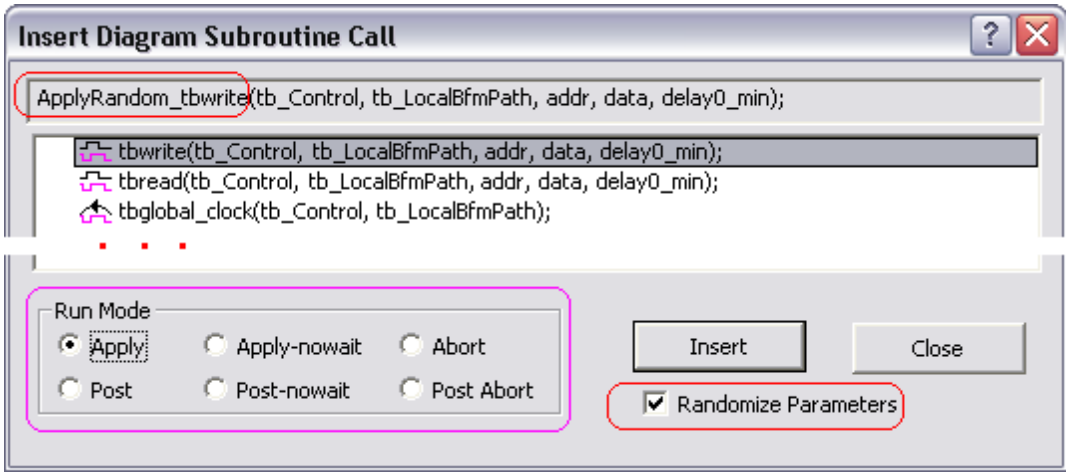
7.7 Randomizing Transaction Inputs

This topic describes how to automatically randomize data going into a transaction. If you want to randomly apply transactions to the model under test see Section 7.6: VHDL: TM fill from Random Generator 121. To call a specific transactor, but with randomized input data, use either ApplyRandom or PostRandom calls. You can define a subrange for the randomization of an input variable using the *Classes and Variables* dialog (by default, variables randomize over the entire legal value range for the variable's type).

Using Insert Diagram Call dialog to Write Transaction Calls with Random Data:

- To open the *Insert Diagram Subroutine Calls* dialog, put the cursor in the template file on the line to insert, and then right click and choose **Insert Diagram Calls** from the context menu.

Copyright © 2014, SynaptiCAD Sales, Inc.



- Select the **Randomize Parameter** check box to change all of the Apply and Post calls to Random data calls.
- Select one of the Apply or Post radio buttons and select a transaction to highlight it, then press the **Insert** button. Post calls are not supported in Verilog.

Constraining Random Data Generation:

By default each variable in the diagram will be randomized using the entire range of the type of the variable. To constrain the randomization or to stop randomizing a particular variable, use the *Classes and Variables* dialog as shown in Section 6.5 Random Number Generation and Constraints 92.

Master Transaction Manager Calls:

```
ApplyRandom_diagramName ( signal tb_Control : inout tb_Control_Type;
                transactorBfmPath : in string );
ApplyRandom_diagramName_nowait ( signal tb_Control : inout tb_Control_Type;
                transactorBfmPath : in string );
PostRandom_diagramName ( signal tb_Control : inout tb_Control_Type;
                queuePath : in string;
                transactorBfmPath : in string );
PostRandom_diagramName_nowait ( signal tb_Control : inout tb_Control_Type;
                queuePath : in string;
                transactorBfmPath : in string );
```

Slave Transaction Manager Apply Calls:

```
ApplyRandom_diagramName_looping ( signal tb_Control : inout tb_Control_Type;
                transactorBfmPath : in string );
ApplyRandom_diagramName_looping_nowait( signal tb_Control : inout tb_Control_Type
                                        transactorBfmPath : in string; );
PostRandom_diagramName_looping ( signal tb_Control : inout tb_Control_Type;
                queuePath : in string;
                transactorBfmPath : in string; );
PostRandom_diagramName_looping_nowait ( signal tb_Control : inout tb_Control_Type
                queuePath : in string;
                transactorBfmPath : in string; );
```

Copyright © 2014, SynaptiCAD Sales, Inc.

7.8 Changing a Project Template File

A top-level template file is automatically created for a project when the project is created and usually does not need to be changed. However, it is necessary to replace the template file for a project when the language of the project is changed, because the template file is written in the target generation language (see Section 9.6: Project Generation Properties dialog 144). The *Copy TestBench Template* dialog is used to copy a new template file into a project, and 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 Transaction-Level Model (including the 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 **Transaction-Level Model** folder of the *Project* window.

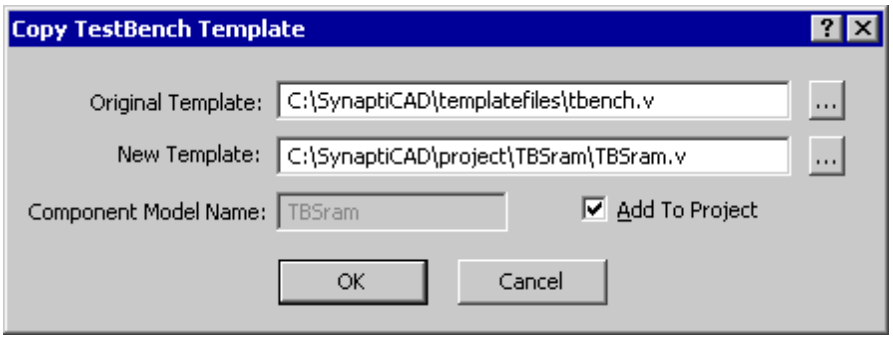
Alternate Recommended Method is to use two Projects and one Class Library:

The instructions is the rest of this section show how to switch in a new template file. This is easy to do if you are only doing it one time, however if you need to continually generate in two different languages then this can get a little confusing. So it is recommended that you instead use two projects if you will need to maintain a test bench in two languages.

Create separate projects with the language-dependent template files in different directories, then include the same transactions and class libraries into the second project. This way, if you alter the language-independent parts of the first project, they will automatically update the second project.

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

- Choose **Project > Copy TestBencher Template File** 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 TestBencher:
  - **TestBuilder** use *tbuidOnlyTBench.cpp* as the primary and *tbuildOnlyTBench.h* will be included as a secondary template file.
  - **Verilog** use *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.

Copyright © 2014, SynaptiCAD Sales, Inc.

- **Verilog with TestBuilder Enabled** use *tbuildTBench.cpp* as the primary, with *TbuildTBench.h*, *tbuildMain.cpp,* and tbench.v included as secondary template files.
- **VHDL** use 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 Enabled** use *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.

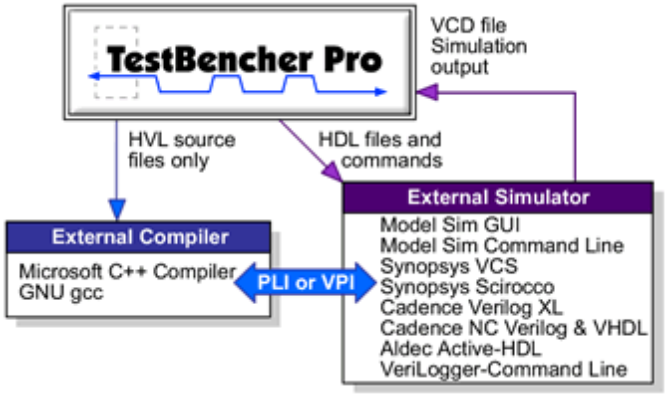
To switch in a different template file:

- 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.

Copyright © 2014, SynaptiCAD Sales, Inc.

Chapter 8: Simulating a Test Bench

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



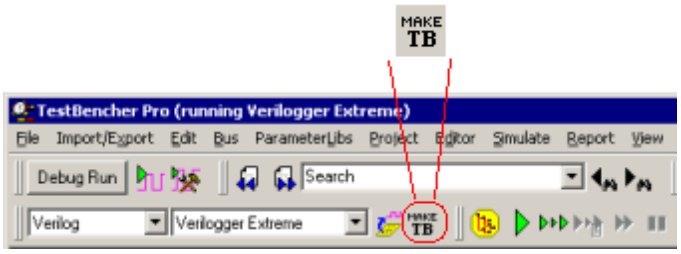
8.1 Simulate the Test Bench and Design

Test benches created by TestBencher can be simulated standalone in standard simulators, without requiring a TestBencher license to perform the simulation. But during development of a test bench, it's often useful to use TestBencher to launch and control your chosen HDL simulator. TestBencher can remotely control most popular 3rd-party VHDL and Verilog simulators and C++ compilers. The settings for configuring the launch of external simulators and tools are controlled using the *Simulator and Compiler Settings* dialog described in Step 7: Setup External Simulators 21.

To build and simulate a bus-functional model:

Both third-party simulators and SynaptiCAD's simulators are controlled through the simulation button bar. Below is a brief description of how to simulate a test bench using the simulation button bar. See the BugHunter Manual for complete details on HDL simulation and debugging features.

- Before simulating, it is a good idea to press the **Generate Test Bench** button to make sure your most recent changes have been incorporated into the generated code (See Step 6: Generate the Test Bench 19).



- If this is the first time you are using TestBencher/BugHunter to simulate with your current simulator, then follow the steps in Step 7: Setup External Simulators 21 and make sure to follow the directions for the **Compile Syncad Libraries** button to compile libraries using your new simulator.

- Click the yellow **Compile Model and Test Bench** button. This builds (parses) the project using the tools specified in the *Project Simulation Properties* and *Simulator/ Compiler Settings* dialogs (Section 8.2: Project Simulation Properties dialog 129). For mixed language projects like TestBuilder, several tools will be needed to compile different parts of the project. For example, TestBuilder projects require a C++

Copyright © 2014, SynaptiCAD Sales, Inc.

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, before going on to the next step.
- Run the Simulation. The **Run/Resume** button compiles the files (if there have been changes since the last build) and then runs a simulation until it is stopped by a breakpoint, the Pause button, the End Simulation button, or the end of the simulation is reached. The Run/Resume button also continues a simulation when it is currently paused. The **<F5>** key and the **Simulate > Run** menu perform the same function. Simulations can also be run in single-step mode (green buttons with multiple small arrows) or run for a specified amount of simulation time (green button with an hourglass above it)





- During Simulation the status of the simulation is displayed in the bottom right hand corner of the TestBencher Pro main window. When the simulation is complete, **Simulation Good** will be displayed.



- When a simulation is paused or completes, 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 TestBencher to build the project, they will be automatically added to the project, and contained in the *Compiled Library Files* folder.

Using TestBencher to start the ModelSim Graphical Interface

In addition to being able to launch the command line ModelSim simulators, TestBencher can also launch the graphical version of ModelSim. Once a design is loaded into ModelSim, three windows will be opened automatically - the Structure, Signals, and Wave windows. By default, TestBencher will also add all of the top-level signals from the component entity to the Wave window. These signals will allow you to monitor the MUT and the transactions during simulation from inside the ModelSim interface.

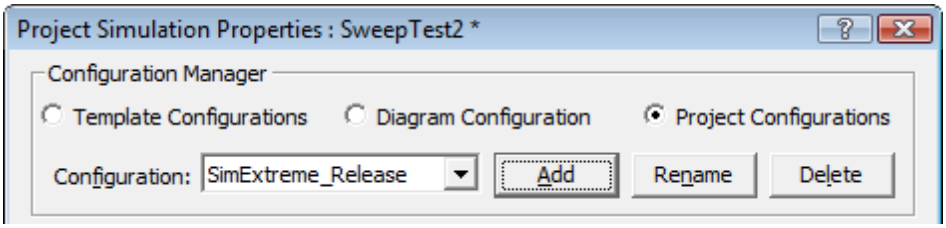
8.2 Project Simulation Properties Dialog

The *Project Simulation Properties* dialog controls how a project is compiled and run, and determines the simulator run time options and which simulator to use for projects and diagrams. If you are moving projects back and forth between different machines and simulators, you may wish to create a**configuration** template for each machine. Also you may wish to have different configuration for different debug and release setups. The basic operation of this dialog is covered in Step 7: Setup External Simulators 21. This section defines each of the controls in more detail .

Copyright © 2014, SynaptiCAD Sales, Inc.

To open the Project Simulation Properties dialog:

- Press the **Project Simulation Properties** button or choose the **Project > Project Simulation Properties** menu to open the dialog.



Global versus Project Settings

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

Configurations:

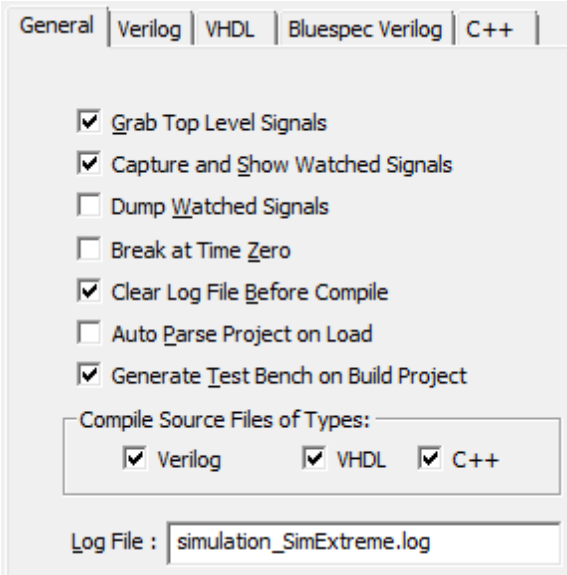
If you are moving projects to different machines or if you want to have different project settings for debugging and releasing a project you may want to create a new configuration to store the different settings. You can also create different configurations for different simulators which need different command line options. Configurations are also useful if you have projects containing source files in multiple languages and you occasionally only want to compile files of a particular language type (for example, when you're translating from VHDL to Verilog and only want to compile the translated files).

The ***Debug* Configuration** holds the default settings. If you need to create a new configuration:

- Press the **Add** button to open the *Add New Configuration* dialog and you specify a name and the default configuration to copy 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 is currently active (e.g. which settings will be used for compiling and running simulations).

Copyright © 2014, SynaptiCAD Sales, Inc.

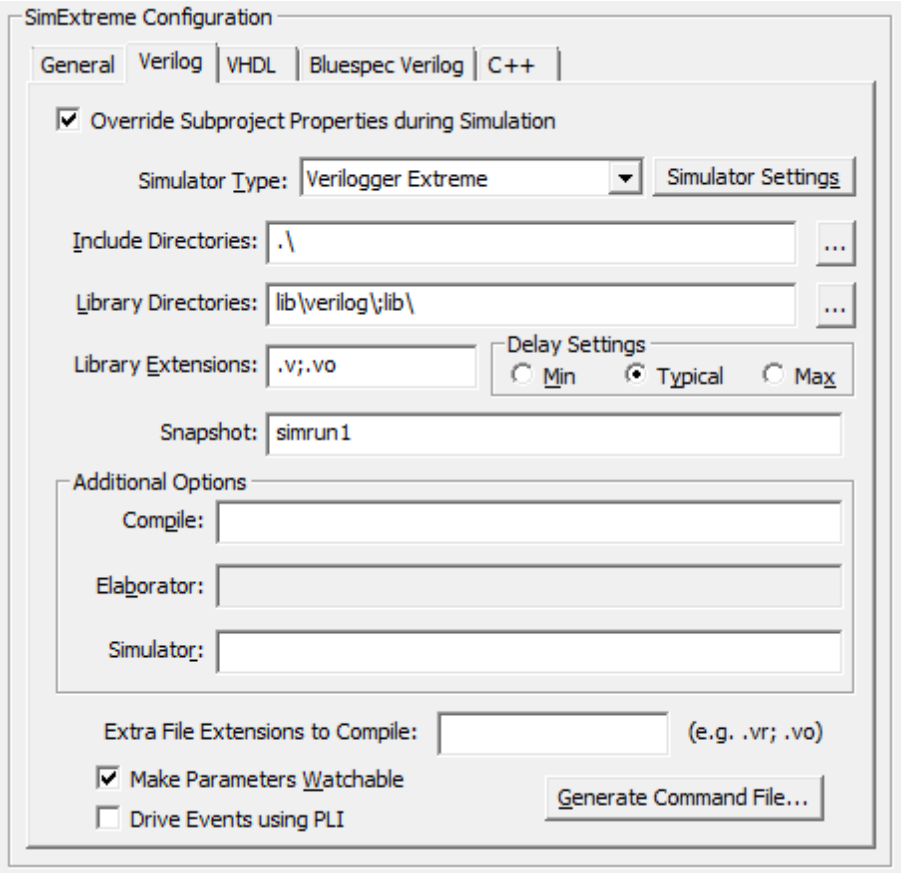
The General tab contains simulation options that are standard across all of the simulators.



- **Grab Top Level Signals** turns on the automatic monitoring of ports or internal signals in the top-level module.
- **Capture and Show Watched Signals** enables the display of waveform results from a simulation run.
- **Dump Watched Signals** generates a dump file for any watched signals in the diagram. The generated file named ***diagramName*.VCD**.
- **Break at Time Zero** is the equivalent of setting a breakpoint at time zero. This starts the simulator and allows you to enter commands into the console window that will be executed during simulation.
- **Clear Log File Before Compile** clears the simulation log just prior to a new compilation being performed. 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.
- **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.
- **Compile Source Files of Type** section specifies which kind of files to compile. This is especially important for V2V translation projects, and projects with mixed language code. Consider using this along with the above *Configurations* setting, by having a VHDL configuration and a Verilog configuration for doing different simulations.
- The **Log File** specifies the name of the log file that receives all the simulation results and information. By default TestBencher uses *simulation.log*.

Copyright © 2014, SynaptiCAD Sales, Inc.

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



- **Simulator Type** determines the simulation tool used when compiling source files for a given language.
- **Simulator Settings** button opens the *Simulator / Compiler Settings* dialog where you can edit the path to the simulation tool.

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

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

- **Library Directories** lists the path and directories where the program searches for library files. TestBencher 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
  ```

- **Lib Extensions** 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 determine 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.
- The **Snapshot** box accepts a text string that will become the name of the executable that is built (overriding the normal name of the simulation build).
- **Compile**, **Elaborator**, and **Simulator** option edit boxes allow you to set additional command

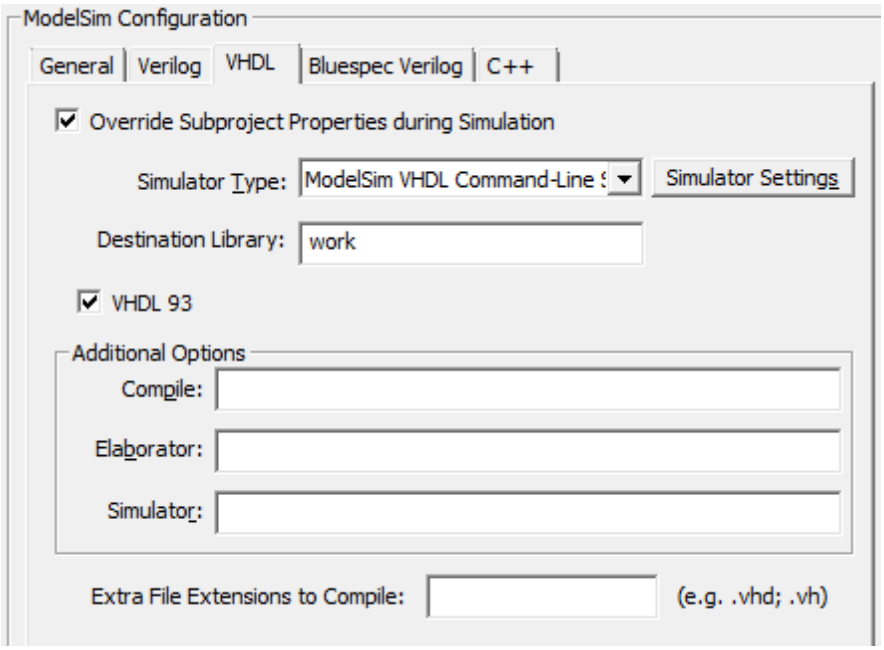
Copyright © 2014, SynaptiCAD Sales, Inc.

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 **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 **Drive Events using PLI** checkbox changes the way stimulus from the StimulusAndResults diagram is sent to the simulator. By default, stimulus waveforms cause Verilog stimulus code to be compiled into the simulation (see Section 3.2: Drawing Waveforms for Stimulus Generation). But when this option is checked, stimulus is directly injected into the simulation at runtime via a PLI application that reads from the btim timing diagram file. This allows you to change the stimulus by editing the btim file without requiring a recompile of your simulation. The disadvantage of this approach is that you cannot single step through the stimulus, since it's injected via PLI.
- The **Extra File Extensions** to Compile adds more types of files that will be treated as Verilog files.
- 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 TestBencher GUI.

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

- **Simulator Type** determines the simulator.



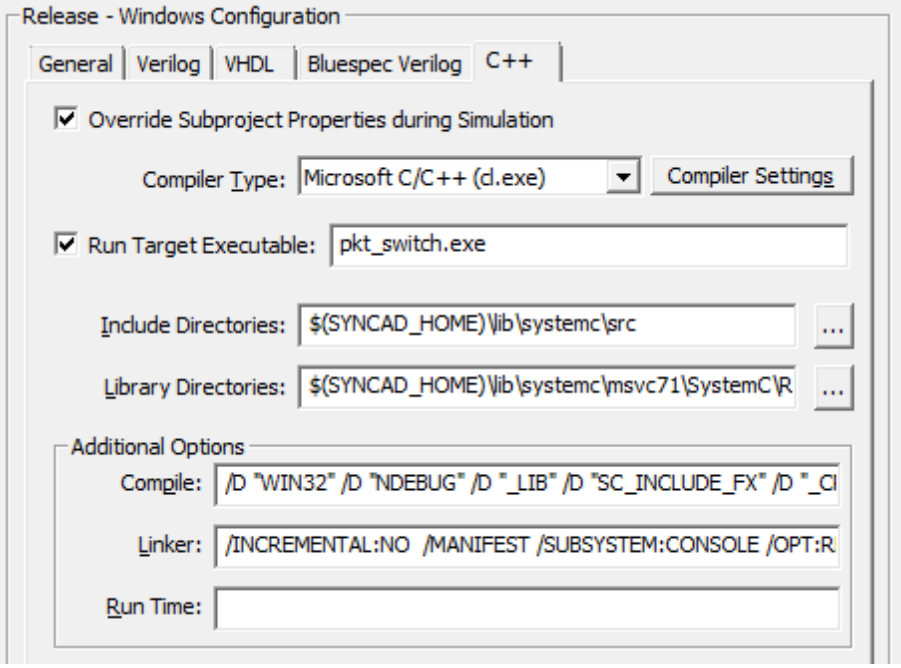
- The **Simulator Settings** button opens the *Simulator / Compiler Settings* dialog where you can edit the simulator paths.
  - The **Override Subproject Properties during Simulation** makes the subprojects use the properties of the top-level project during simulation (and ignore their own private properties). This only affects TestBencher projects that use hierarchical sub-projects.
  - The **Destination Library** controls the logical library that sources files will be compiled into when a build is performed.

Copyright © 2014, SynaptiCAD Sales, Inc.

- **VHDL 93** specifies that that 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 **Extra File Extensions to Compile** adds more types of files that will be treated as VHDL files.

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

- **Compiler Type** determines the C++ compiler.
- **Compiler Settings** opens the *Simulator / Compiler Settings* dialog where you can review and edit the compiler paths.



- The **Override Subproject Properties during Simulation** makes the subprojects use the properties of the top-level project during simulation (and ignore their own private properties). This only affects TestBencher projects that use hierarchical sub-projects.
- The **Run Target Executable** sets the name for the executable program that will be built/run from the C++ source code in the project.
- The **Include Directories** specifies the include directories that will be searched for header files when files are included into a source file.
- The **Library Directories** specifies the directories to search for library files during the build process.
- 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.

8.3 TestBencher Simulation Modes

TestBencher supports two methods for generating test benches: system level TLM 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 BugHunter Pro on-line help. The system-level features are covered in this manual.

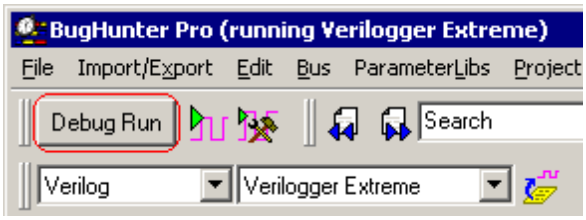
To support these methods of test bench generation, TestBencher has two simulation settings, **Auto**

Copyright © 2014, SynaptiCAD Sales, Inc.

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.

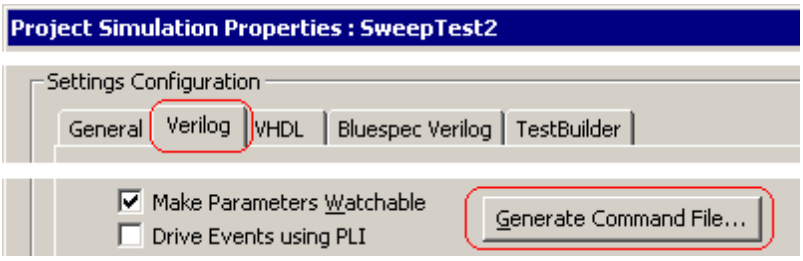


8.4 Generate Command Files for Third Party Simulators

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 Step 7: Setup External Simulators 21 and Step 8: Simulate the Test Bench 24.

Generate the command file:

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



- In the **Verilog** tab, press the **Generate Command File** button to open a file dialog.
- 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.

Use the command file:

To use this file with a third party simulator, you will probably 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:

```
simx –f test.vc
```

8.5 Parsing Package Files into Libraries

The interface to your model under test may require definitions for a package defined in a different library. For example, your device may have a PCI port and require constants defined in a library that

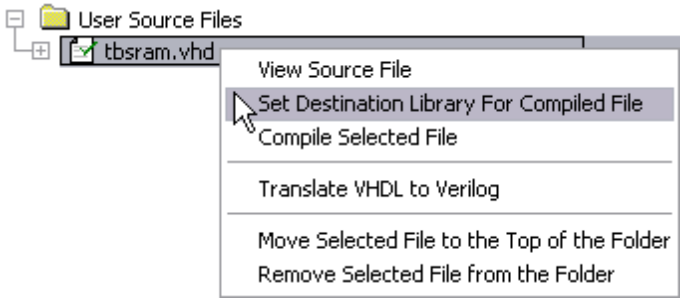
Copyright © 2014, SynaptiCAD Sales, Inc.

defines the bus sizes for the PCI signals. TestBencher needs to parse the files containing these packages in order to know the values of these constants when it parses your model under test, so these package files must also be included in the list of user source files. In addition, you need to to tell TestBencher the destination library for these packages:

Parsing Package files into Libraries

By default, the source files under the **User Source File folder** (in the Project Window) are compiled to the standard 'work' library in VHDL. Sometimes, however, these source files may need to be compiled to a different library. You can override the default work library destination for a source file by specifying the new library using a right click context menu.

- In the User Source Files folder, right click on the source file and choose **Set Destination Library for Compiled File** from the context menu. This opens a dialog where you can set the name of the logical library that this file will be compiled to when Builds are performed.



Compiling Files to Libraries in 3rd party Simulators

The above steps are sufficient to allow TestBencher to parse a model under test that uses packages from other libraries, but additional steps may be required to tell a 3rd party simulator where to look for the libraries on disk. These steps vary, depending on the simulator used.

For Cadence Incisive (ncsim):

- Open the **cds_project.lib** file located in the project directory and add the logical-library-to-physical-directory mapping.

```
DEFINE lib1 ./lib1_ncsim
```

- For example, the above would map a logical library named lib1 to a directory called lib1_ncsim. **Note:** If a directory called lib1_ncsim doesn't already exist, you will also need to create this directory from the shell.

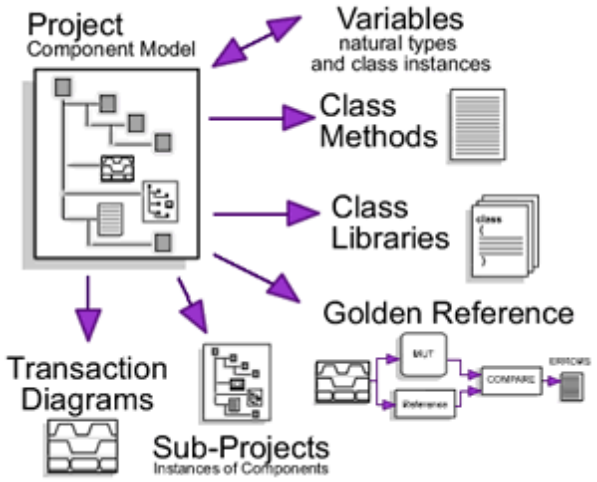
For ModelSim and Aldec:

- No further steps required as TestBencher automatically runs the required vlib commands.

Copyright © 2014, SynaptiCAD Sales, Inc.

Chapter 9: Advanced Project Features and Properties

This chapter covers advanced project level features including sub-projects, golden reference models, and properties that effect the code generation. Sub-projects are TestBencher projects that are contained within other projects, allowing complex test benches to be developed and verified in an incremental manner. Sub-projects also provide an easy interface for multiple port testing.



TestBencher 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.

Right-mouse click on any element in the project tree to reach the context sensitive menu that provides a list of actions that can be performed for that element. Many elements have *Properties* dialogs that can be reached through the context menus and Project menu functions.

9.1 Sub-Projects

TestBencher Pro supports hierarchical TLM 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.

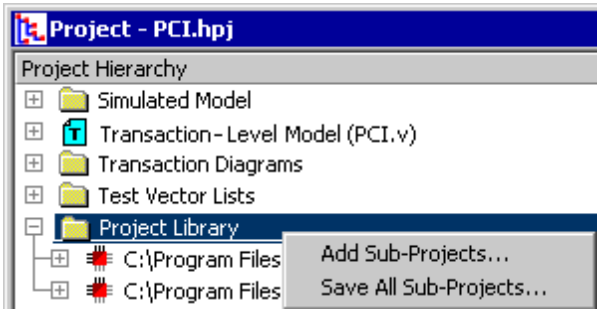
General Sub-Project Design Flow:

- Design and debug each TestBencher project that will become a subcomponent of a larger project.
- In the larger project, add the sub-project to the **Project Library** folder so that the larger project has access to the sub-projects files (see the instructions below).
- Set the default signal mappings to use when the sub-component is instantiated (see the instructions below).
- Instantiate the sub-component inside the larger project and set the port mapping. You may create more than one instance of the sub-component (See Section 9.2 Component Instances of Sub-Projects 139).

Copyright © 2014, SynaptiCAD Sales, Inc.

Add a sub-project to the Project Library folder:

- Right-click on the **Project Library** folder and choose **Add Sub-Projects** menu. 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.

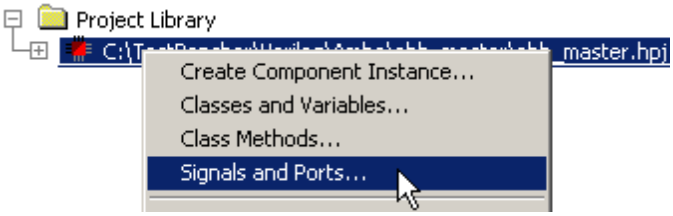


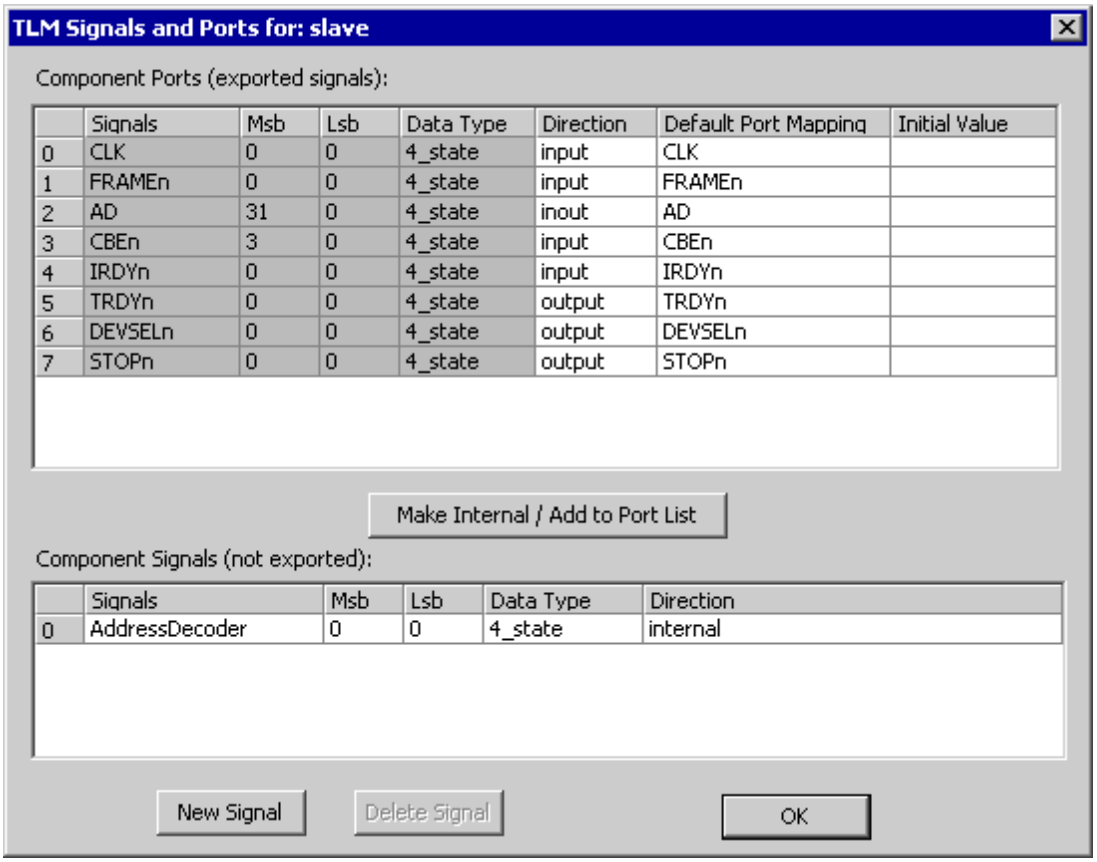
The sub-project can be modified while the owning project is open, by expanding the sub-project tree. The sub-project will remain an independent project that can be opened and edited alone, as well.

Setup default mappings

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

- 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.





	Signals	Msb	Lsb	Data Type	Direction	Default 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	
3	CBEn	3	0	4_state	input	CBEn	
4	IRDYn	0	0	4_state	input	IRDYn	
5	TRDYn	0	0	4_state	output	TRDYn	
6	DEVSELn	0	0	4_state	output	DEVSELn	
7	STOPn	0	0	4_state	output	STOPn	

	Signals	Msb	Lsb	Data Type	Direction
0	AddressDecoder	0	0	4_state	internal

Copyright © 2014, SynaptiCAD Sales, Inc.

- Create ports for the sub-project by selecting signals in the *Component Signals (not exported)* 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.

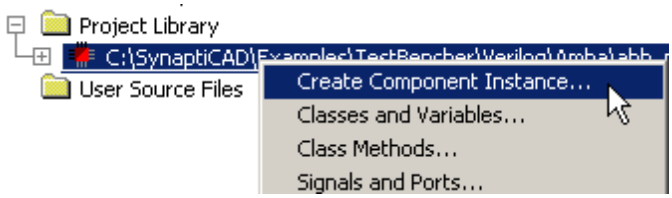
For more information on the *Signals and Ports* dialog see Section 9.5 Signals and Ports for Components 142.

9.2 Component Instances of Sub-Projects

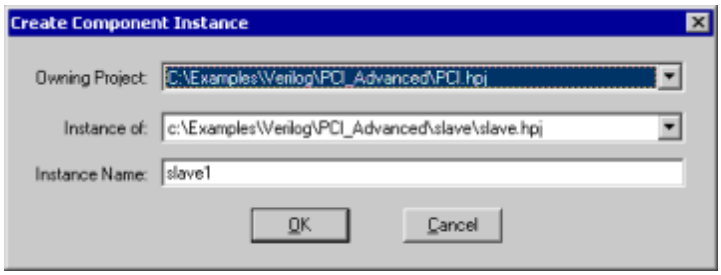
Projects listed in the **Project Library** folder can be instantiated and used by the containing project. When a sub-project is instantiated, TestBencher will generate a component instantiation in the top-level template within the *$TBSubComponentInstantiation* macro and make all of the transaction diagrams 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 7.1: Template, Sequencer, and Transaction Calls 112.

Create a Component Instance of the sub-project:

- Right-click on the sub-project node (the red chip icon) 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.
- Press **OK** to close the dialog and add the component instance (green chip icon) to **Transaction-Level Model** folder.

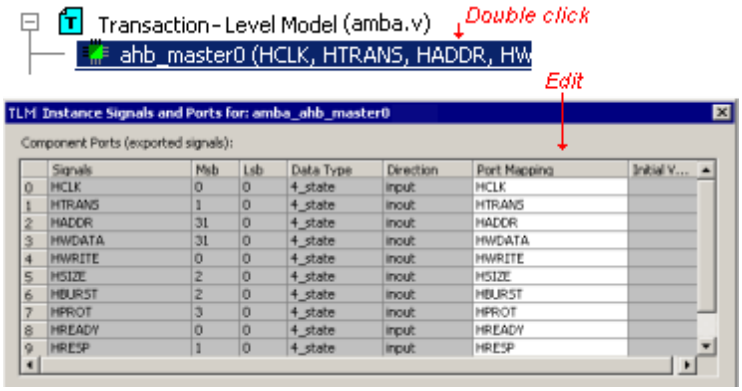
Edit the port mappings of the new component:

A new component instance uses the default port mappings that were set using the instructions in the Section 9.1 Sub-Projects 137. If these are correct you can skip this step. If there are multiple instances of a sub-project, then some of the port mappings will need to be edited so that this

Copyright © 2014, SynaptiCAD Sales, Inc.

particular instance hooks up to the correct signals in the owning project.

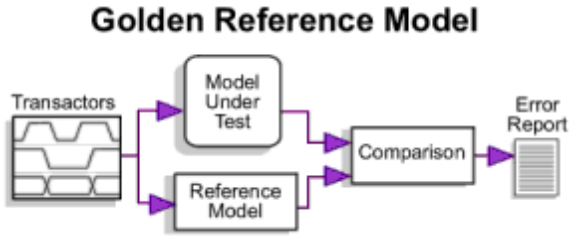
- In the ***Transaction-Level Model*** folder, double-click on the ***name of the component instance*** (green chip icon) to open the *TLM Instance Signals and Ports* dialog.
- Double-click on a cell in the **Port Mapping** column and either type in a name or choose a signal name from the drop-down list. The signals in the list are the signals in the current project or the ports of the MUT.



For more information about the *Component Signals and Ports* dialog, see Section 9.5: Signals and Ports for Components 142.

9.3 Golden Reference Models

TestBencher 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 the model under test 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.



TestBencher 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. TestBencher uses the TestBuilder library to generate 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 model under test.

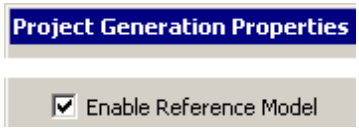
Copyright © 2014, SynaptiCAD Sales, Inc.

To enable Reference Model Generation:

- Right click on the **Transaction-Level Model** folder and choose **Project Generation Properties** from the context menu to open the *Project Generation Properties* dialog.



- Check the **Enable Reference Model** checkbox and click **OK** to close the dialog.



- During the next code generation TestBencher will generate either a C++ or Verilog reference module depending on the generation language. Step 6: Generate the TestBench 19 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.
- Step 7: Setup External Simulators 21 describes how to setup the C++ compiler so the TestBencher 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.

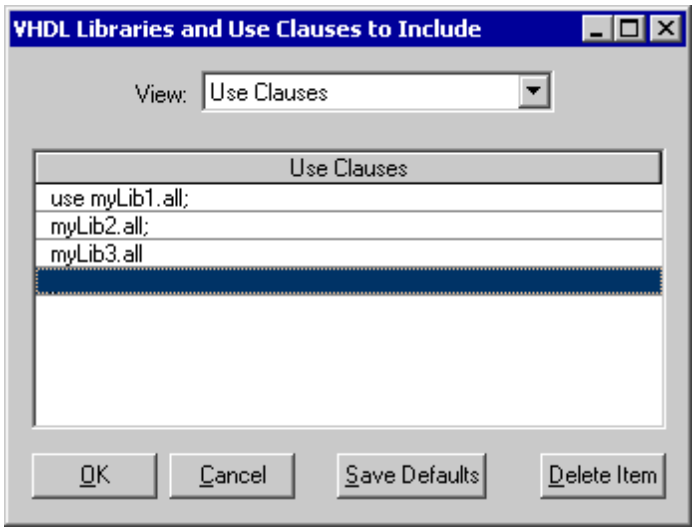
9.4 Libraries and Use Clauses dialog (VHDL)

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. 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.

Copyright © 2014, SynaptiCAD Sales, Inc.

VHDL Library and Use Clause dialog controls for VHDL support:

- Choose the **Options > VHDL Libraries and Use Clauses** menu to open the dialog. Changes made in this dialog will be applied to all diagrams that are exported to VHDL.
- From the **View** control select either **Use Clauses** or **VHDL Libraries** and enter the information onto a blank row.



- If the 'USE' is omitted from a use clause, or the 'LIBRARY' from a library statement, WaveFormer will automatically add 'USE' or 'LIBRARY' before the clause in the source file for the diagram. If necessary, WaveFormer will also automatically add semicolons to the end of library includes and use clauses
- 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
```

Exporting a single 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 for all diagrams.

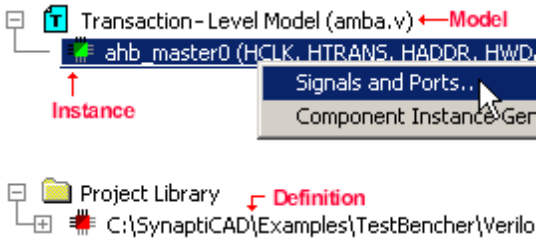
9.5 Signals and Ports for Components dialog

The *Signals and Ports* dialog is used to view and edit the ports and internal signals of Transaction-Level Models and definitions as well as the port mappings of Component Instances. TestBencher 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.

Copyright © 2014, SynaptiCAD Sales, Inc.

To open the Signals and Ports dialog:

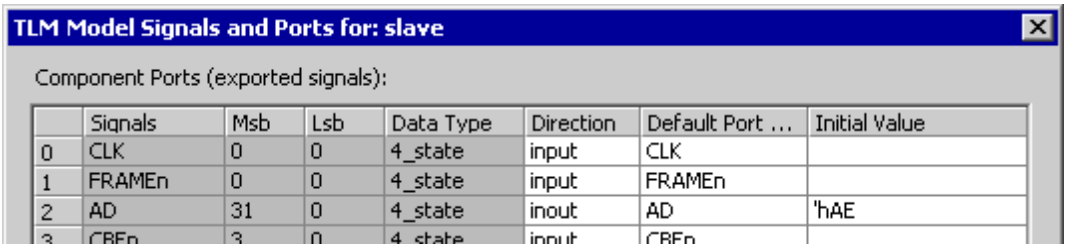
- Right click the item to be edited (a Transaction-Level Model, Component Definition or a Component Instance) and select **Signals and Ports** from the context menu. This will open the *Signals and Ports* dialog.



While this dialog is open, you can select any Component or Component Instance in the *Project* window and edit the signal and port information associated with that object.

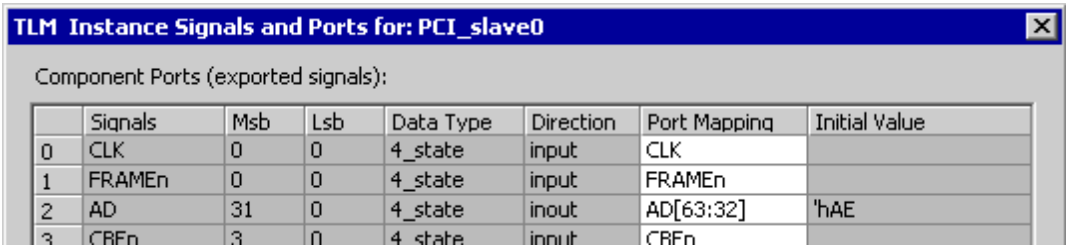
Editing signals and ports with the dialog:

- 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. These fields are fixed and cannot be edited by the dialog.
- 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.



	Signals	Msb	Lsb	Data Type	Direction	Default Port ...	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	'hAE
3	CBEn	3	0	4_state	input	CBEn	

- Component Instance internal signals cannot 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	CBEn	3	0	4_state	input	CBEn	

- New component level signals can be added when editing signals and ports for the Transaction-Level 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 cannot 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. TestBencher will

Copyright © 2014, SynaptiCAD Sales, Inc.

automatically create a new signal in the owning component for this connection.

- A **bit slice** can be specified for each **Port Mapping**. TestBencher 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, TestBencher 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, TestBencher 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**.

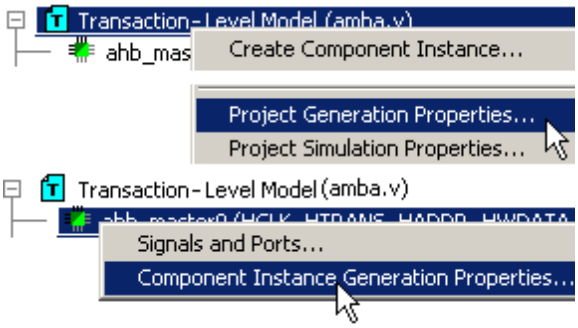
The Transaction-Level Model has both exported signals and internal signals. By default, all signals that are generated by TestBencher are internal, but the Component Signals and Ports dialog allows these signals to be exported by simply changing the direction of the signal.

9.6 Project Generation Properties dialog

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. The controls in this dialog change depending on if you are editing a Component Instance or the actual Transaction-Level Model.

Use context menus to open Project Generation Properties dialog:

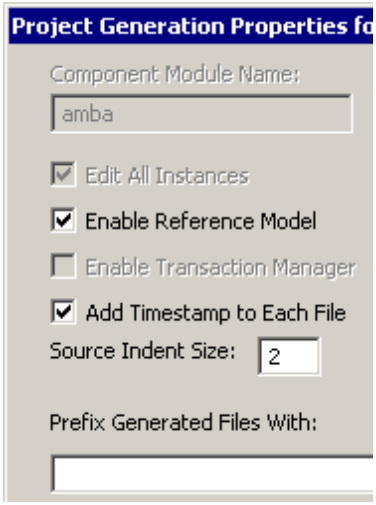
- For project level properties, right-click on the **Transaction-Level Model** folder and choose **Project Generation Properties** from the context menu
- For sub-project level properties, right click on a component instance (green icon) and choose **Component Instance Generation Properties** from the drop-down list box.



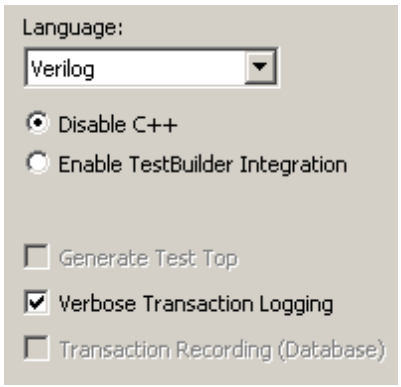
Properties of a top-level project:

The following properties can be edited at the Transaction-Level Model level:

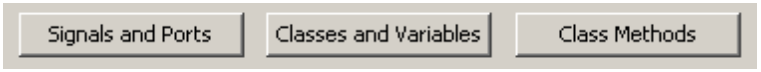
Copyright © 2014, SynaptiCAD Sales, Inc.



- **Enable Reference Model** enables the generation of a golden reference model in C++, VHDL, or Verilog. This feature is covered in Section 9.3: Golden Reference Models 140.
- **Enable Transaction Manager** enables the generation of the transaction manager code. This feature is covered in Section 7.3: Transaction Manager Overview 115.
- **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.
- **Source Indent Size** specifies the number of spaces that are used for indenting blocks of code.
- **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.



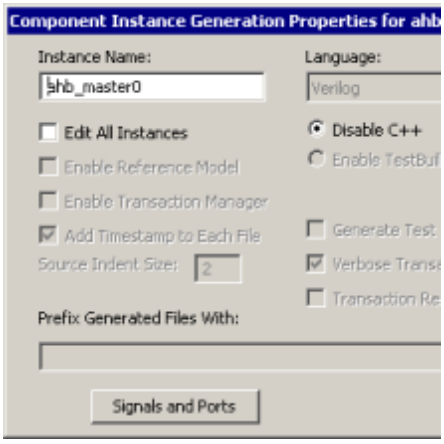
- **Language** and **Enable TestBuilder Integration** control the generation language for this component. If the language for the project is changed, then the project template file must also be changed. Section 7.8: Changing a Project Template File 126 discusses changing the project template file.
- **Verbose Transaction Logging** reports to the simulation log file when transactions start and stop. Also if the transaction is applied in looping mode it reports when each loop starts.
- **Transaction Recording** enables SDI transaction recording calls. This records every transaction that is run during simulation along with applied parameters and 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 **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 6: Classes and Variables 82 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 6.3: Class Methods 87 for more information.

Copyright © 2014, SynaptiCAD Sales, Inc.

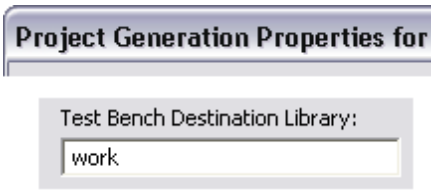
Properties of a Component Instance of a sub-project:



- **Instance Name** 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 **Transaction-Level Model** folder of either the containing project or in a sub-project's **Transaction-Level 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.
- **Disable C++** indicates whether TestBuilder is available or not.

Destination Library for TestBencher Pro Generated files:

By default, TestBencher Pro compiles the generated testbench files to the standard 'work' library in VHDL. Sometimes, however, these source files may need to be compiled to a different library. You can override the default work library destination for all generated files via a setting in the Project Generation Properties dialog.



- Choose **Project > Project Generation Properties** from the main menu, to open a dialog of the same name.
- Type in a new destination library name into the **Test Bench Destination Library** box of the *Project Generation Properties* dialog.

Note: User-written source files are also compiled into the work directory by default. The destination library for each user-written source file can be set individually by right-clicking on the source file in the project window and selecting **Set Destination Library For Compiled File** from the context menu.

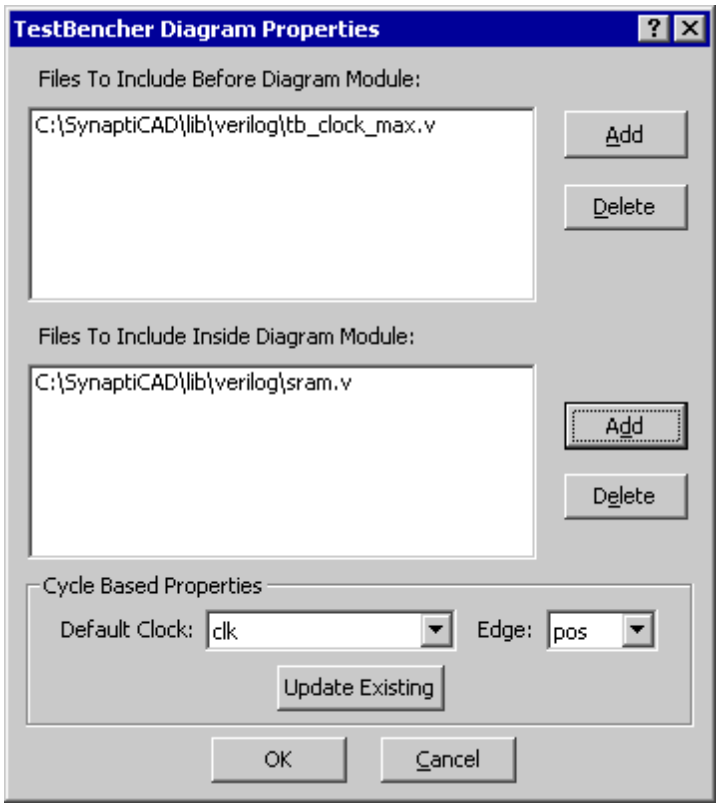
9.7 Diagram Properties dialog

The *Diagram Properties* dialog sets properties that that are ***saved in the timing diagram BTIM file*** and affect all projects that use the diagram. These include the cycle based settings and the include files lists. Changing these properties can break or dramatically change the way the diagram works during simulation. Other diagram settings that are saved in the project HPJ file and only effect the generation of the code (not the operation of the diagram) are edited through the *Diagram Settings* dialog as discussed next in Section 9.8: Diagram Settings Dialog (TestBencher only) 148.

Copyright © 2014, SynaptiCAD Sales, Inc.

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 **TestBencher Diagram Properties** from the context menu. This will open the *TestBencher 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 6.3: Class Methods 87. 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.

Copyright © 2014, SynaptiCAD Sales, Inc.

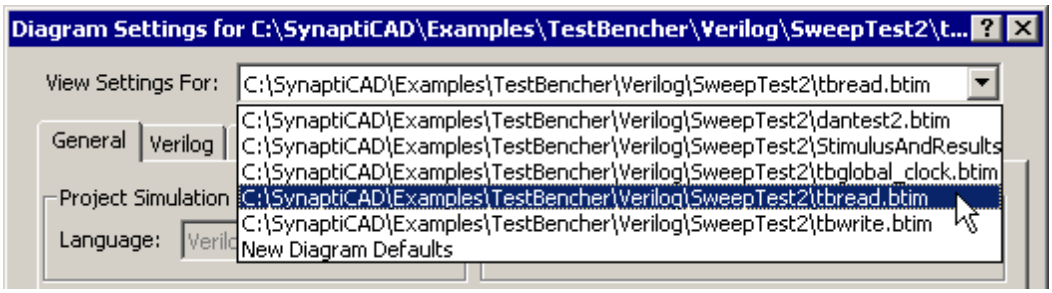
- The **Update Existing** button is used to update all signals, samples, delays and anything with a clocking signal defined to the currently selected **Clock** and **Edge/Level**.

9.8 Diagram Settings dialog (TestBencher only)

In TestBencher Pro, the *Diagram Settings* dialog sets diagram properties that are **saved in the HPJ project file**. These only affect a specific diagram, but since the settings are stored in the project, they can be different across different projects. 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. 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 BugHunter-style projects or for Reactive Test Bench Generation. Properties that affect the basic operation of the timing diagram and are saved in the BTIM file, such as cycle-based settings and the include file list, are edited using the *Diagram Properties* dialog as discussed in Section 9.7: Diagram Properties 146.

To open the Diagram Settings dialog:

- Either 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.



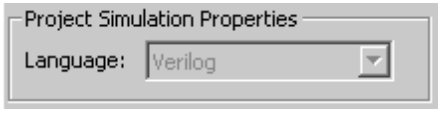
- The **View Settings For** box allows you to quickly view the settings for different timing diagrams in the project. The **New Diagram Defaults** sets the options for new diagrams that are created for the project.

The 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.

Project Simulation Properties

- **Language** controls the generation language for timing diagram. The language choices depend on what languages have been licensed for use with TestBencher. TestBencher can generate Verilog and VHDL testbenches.



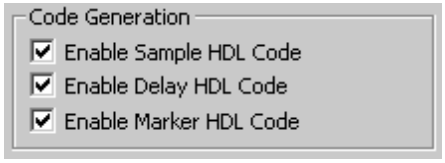
Copyright © 2014, SynaptiCAD Sales, Inc.

Signal HDL



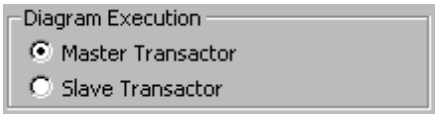
- **Direction** sets the default signal direction for new signals. For more information on signal directions, see 2.3 Waveform Colors and Bi-Directional Signals 35.
- **Type** determines the default SynaptiCAD signal type for new signals. These language independent types are covered in the Timing Diagram Editor manual Section 4.4 Export VHDL and Verilog test benches.

Code Generation



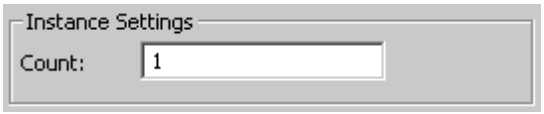
- The three check boxes in the *General* tab allow you to toggle whether HDL code generation is enabled for samples, markers and delays.

Diagram Execution



- These settings control the default execution process for the current diagram (see Section 7.2: Master and Slave Transactions 115).
- 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.

Instance Settings



- The Instance Settings are used to specify information for Pipelining Transactions (see Section 5.8 Pipeline Markers 78). The **Count** determines the number of instances of a transaction. These instances will be automatically instantiated in the test bench.

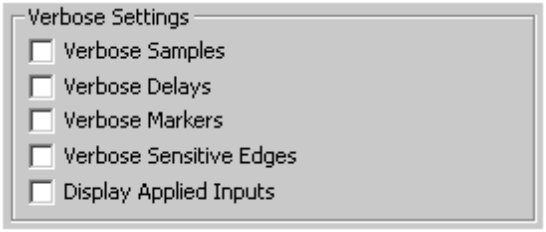
Sample Parameter



- These controls defines the default ***if condition*** and subsequent ***actions*** for new samples. For more information, see Chapter 3: Transaction Samples 49.

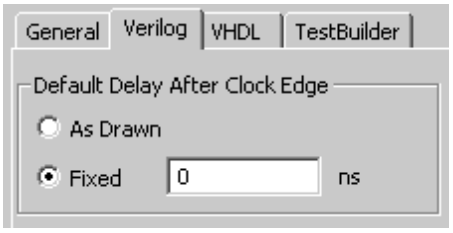
Copyright © 2014, SynaptiCAD Sales, Inc.

Verbose Settings control debugging output:



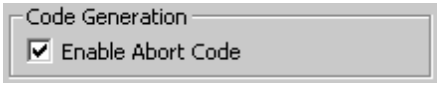
- **Verbose Samples** display when a sample has been triggered to run, when its window has been entered, when it triggers another sample or delay, and when it causes the diagram to restart.
- **Verbose Delays** display when a delay starts, when a conditional test fails (causing the delay to be disabled), and when a delay makes a signal assignment.
- **Verbose Markers** display their name, type, and the simulation time. Markers that trigger loops or simulation run time events also display when they start and what they are doing.
- **Verbose Sensitive Edges** display when the sequence verification starts, when an edge is detected on a sensitive signal, and when an unexpected edge occurs.
- **Display Applied Inputs** displays the values of input parameters that are passed into calls for the selected transactor.

Cycle Based Settings (Language specific)



- This setting affects code that is generated for signals that are clocked . 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 relative to the clocking edge.
- **Early Versions of TestBencher Pro:** 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.

Copyright © 2014, SynaptiCAD Sales, Inc.

Time-out Settings (Language-Specific)

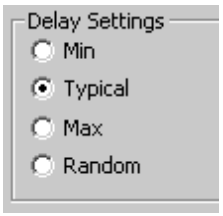
- A transaction **timeout** will aid in the prevention of an endless wait condition occurring in a timing transaction.



- 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)

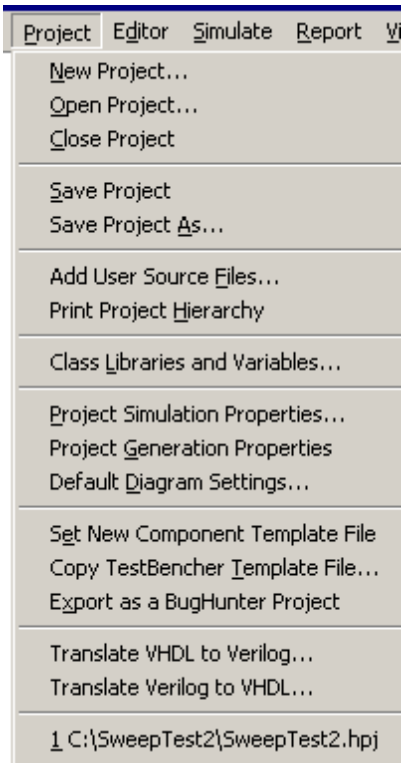
- The **delay settings** determine how delays will be computed during diagram code generation. This setting determines which delay value is used in min:typ:max expressions. The **random** setting allows a random value to be computed from within the range of the min and max specified for the delay.



9.9 Project Menu and New Project Wizard dialog

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

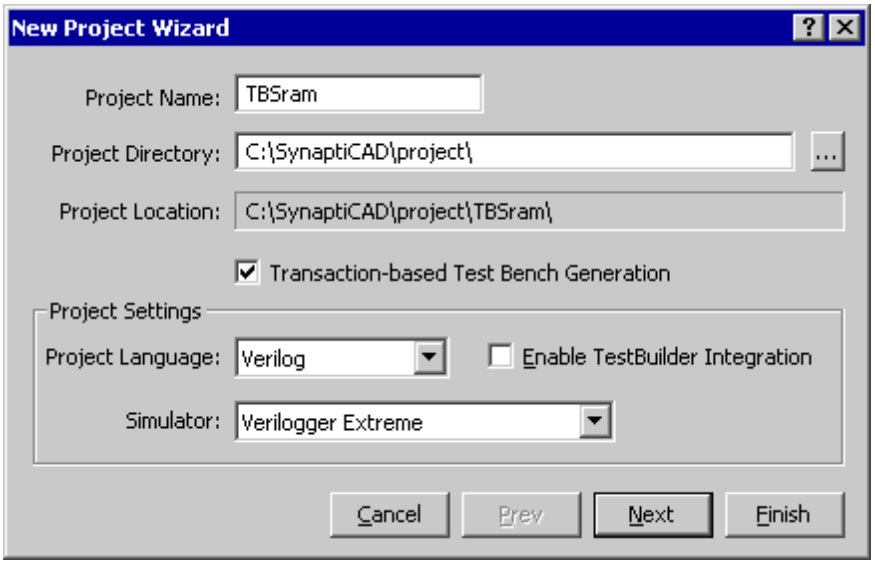
- Select the **Project** menu option and choose one of the project submenus: **New Project**, **Open Project**, **Save Project**, or **Close Project**.
- Notice at the bottom of the menu there is a list of recently opened projects.
- This menu also opens all of the different Properties dialogs that control how the test bench is generated and the tools used to simulate it.
- The **Export as BugHunter Project** allows a TestBencher Project to be exported as a BugHunter project for simulation by those that do not have a TestBencher license. There is also a similar menu in BugHunter that will convert a project to a TestBencher Project.



Create a Project using the New Project Wizard dialog:

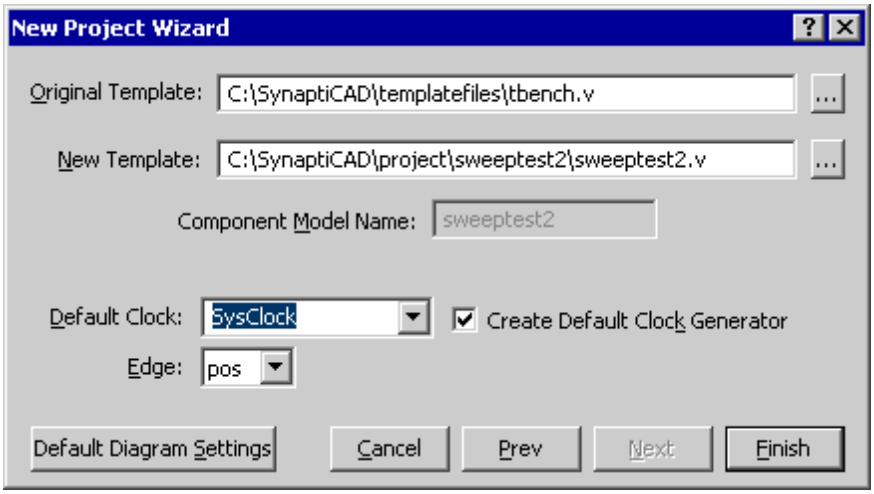
- Choose the **Project > New Project** menu to open the *New Project Wizard* dialog which will step you through the process of creating a new project.

Copyright © 2014, SynaptiCAD Sales, Inc.



- **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.
- **Transaction-based Test Bench Generation** check box enables multi-transaction, bus-functional model generation (i.e. a "TestBencher" test bench). If you uncheck this TestBencher will generate a BugHunter-style project, where only the StimulusAndResults diagram generates a test bench.
- **Project Language** controls the test bench generation language. The available language selections are based on your TestBencher License file.
- **Simulator** specifies the simulator that will be used for this project. The default settings are stored in the dialog reached by choosing the **Options > Simulator/Compiler Settings** menu.
- **Enable TestBuilder Integration** enables C++ code generation for Verilog projects. This adds many advanced language features, such as constrained random number generation.
- Click the **Next** button to view pane two of the *New Project Wizard*:

Copyright © 2014, SynaptiCAD Sales, Inc.

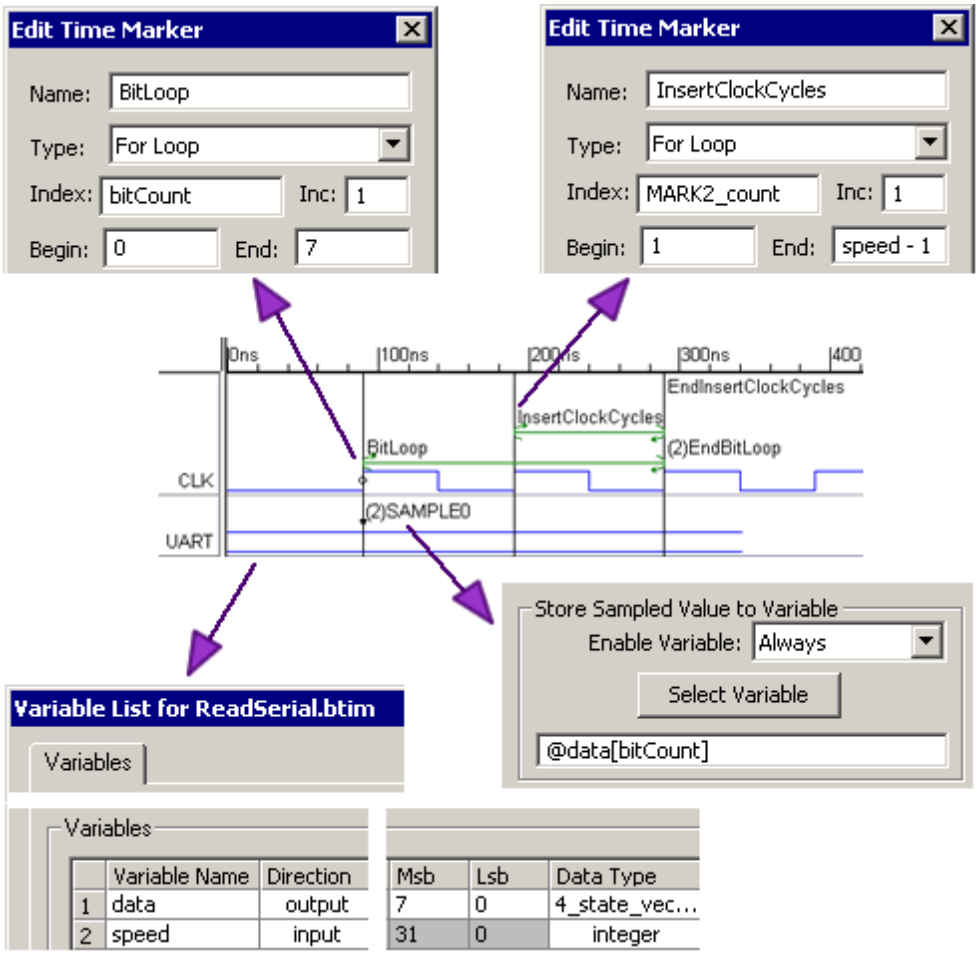


- The **Template** edit boxes specify the original and new (copied) template file names. The new template file will serve as the top-level source file for the test bench. This file is where the Transaction-Level Model is generated. A default original template file name is provided, but you can create your own template files. This feature rarely needs to be used (Section 7.8: Changing a Project Template File 126 has information on editing and changing the template file).
- The **Default Clock** 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 9.8: Diagram Settings dialog 148.

Copyright © 2014, SynaptiCAD Sales, Inc.

Chapter 10: Test Bench Techniques and Examples

This chapter shows some examples of how we have used TestBencher and Reactive Test Bench Generation to solve different types of test bench problems. It also compares and contrasts different ways to loop over a section of stimulus and how to support external models. Read through the examples in the chapter to get an idea of the different ways to use the graphical constructs to solve testing problems.



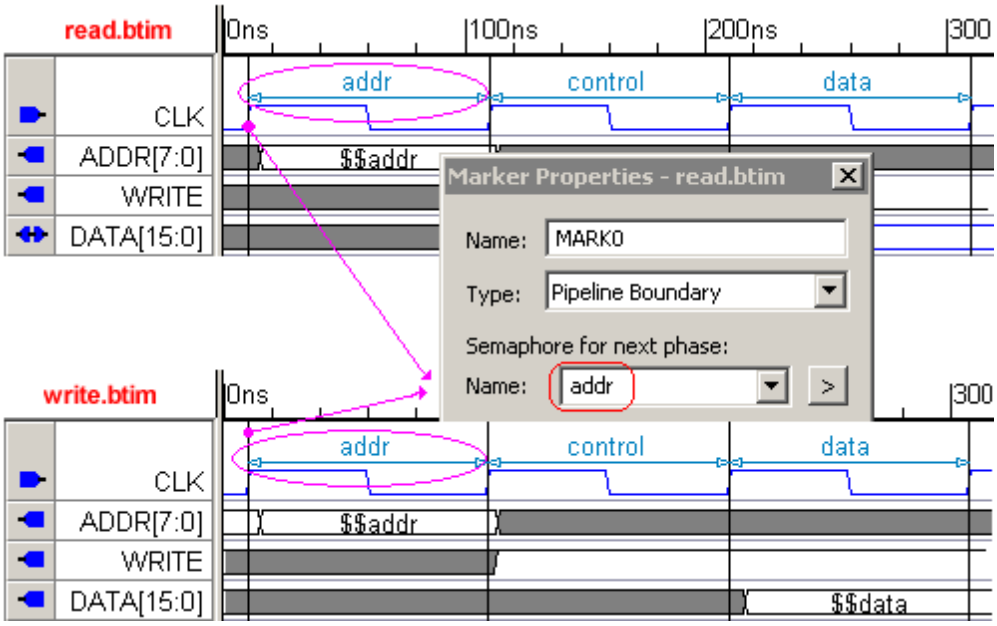
10.1 Pipeline Transactions

TestBencher Pro supports Pipeline modeling using markers and semaphore variables. When you enter a pipeline marker in one timing diagram, its semaphore variable becomes a project level variable that can be used in other timing diagrams in the same project. When a transaction enters a pipeline stage, it decrements the associated semaphore so that any other transaction that attempts to enter that particular pipeline stage will block until the original transaction exits the stage. Section 5.4: Pipeline Boundary Markers 78 describes how to insert pipeline stages into a timing diagram.

Setup the Pipeline Stages in the Timing Diagrams

The **Pipeline** project in the **SynaptiCAD>Examples>TestBencher** directory shows an example of pipelined read and write timing diagrams.

Copyright © 2014, SynaptiCAD Sales, Inc.



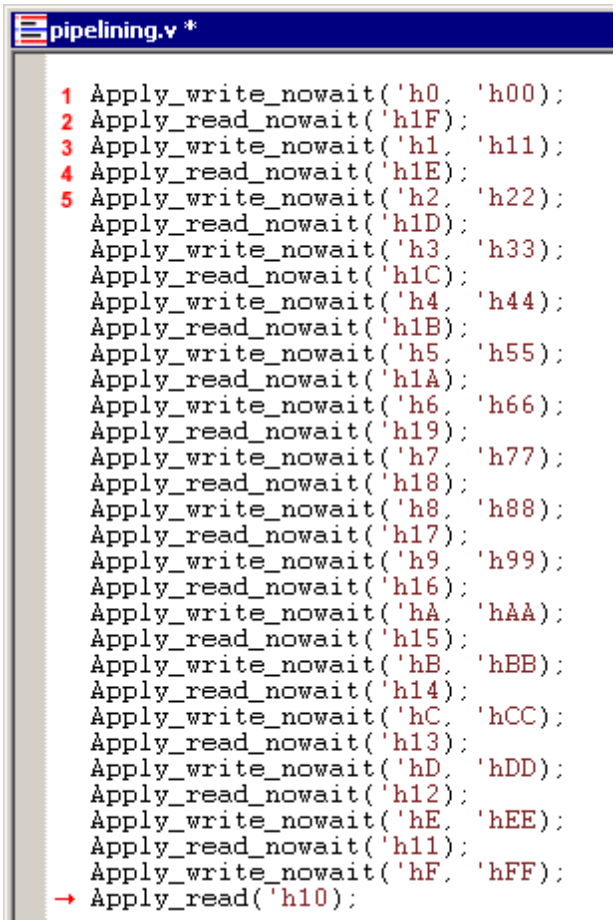
Both the read and the write diagram have three pipeline stages defined by markers with the same semaphore names: **addr**, **control**, and **data**. The pipeline stages are automatically displayed using a blue arrow line and the semaphore name centered between the entry and exit to the stage.

Use Apply-No-Wait transaction calls in the Sequencer Process

In most of the examples in the manual, we call master transactions using a regular Apply call, which blocks subsequent transaction calls until the current transaction is completed. However, when using pipelines, you will want the transactions to run currently and block only at the pipeline markers since the transaction durations overlap in time. To do this, just check the **Apply No Wait** radio button in the *Insert Diagram Calls* dialog (see Step 5: Define Sequencer Process 171).

Copyright © 2014, SynaptiCAD Sales, Inc.

- Since all of the transactions calls are **Apply_nowait**, all of the transactions will start at the same time and queue up in the order they were called.
- Line1: the first **write** call will execute and grab the **addr** semaphore, block all the other transactions from entering the pipeline stage.
- Line 2: when the first **write** transaction moves into the **control** section, it releases **addr** and the next transaction (the read in line 2) grabs **addr** and blocks all others.
- When the first **write** moves into the **data** pipe, it will release **control** and allow the next transaction to enter the **control** stage. This process will continue until all of the transactions have passed through all three stages of the pipeline.
- Note that the final apply call does not need to be a NoWait call. In fact, if it's the last apply call in your sequencer process, you should probably use the "blocking" call version of the Apply, so that the sequencer is blocked from finishing until the last transaction completes. For example, note the Apply_read instead of Apply_read_nowait at the end of the example sequencer.



Semaphore Implementation of Pipeline stages

A semaphore is created for each stage of a pipeline. The semaphore is set with an initial count of 1, effectively making it a mutex (a mutually exclusive semaphore that enables only one transaction at a time to enter the pipeline stage).

When a transaction attempts to enter a pipeline stage, it makes a semaphore **Wait** call to see if it can enter the stage by acquiring the mutex. If it fails, it will wait until the transaction currently executing that stage completes and releases the mutex. When the transaction finally does enter the stage, it will make a **Post** call to release the mutex for the previous stage it was in (assuming this isn't the first stage in the pipeline). Below is pseudocode for the semaphore calls made in the read and write diagrams discussed above:

```
addr.Wait() //attempt to grab the addr mutex,
            //and block until we get it
perform addr stage
control.Wait() //wait until we can enter the control stage
addr.Post() //only allow other diagrams to enter addr stage
```

Copyright © 2014, SynaptiCAD Sales, Inc.

```
            //after we have left it
perform control stage
data.Wait()
control.Post()
perform data stage
data.Post() //last stage in the pipeline, so we don't
            //need to wait for a new stage before releasing
```

Note how the wait for the next sempahore must happen before the post of the previous semaphore. This detail is automated when using Pipeline Boundary markers. For each semaphore needed, a tbfifosemaphore module is instantiated in the top-level component model. That allows the semaphore to be shared between different diagrams in the same project. Currently there is no wait to share a semaphore across multiple projects.

Verilog Specific Details:

The definition of the tbfifosemaphore module can be found in lib\verilog\tbfifosemaphore.v file. This module's purpose is to preserve the order in which diagrams are waiting for a given semaphore. This module defines 3 procedures to access the semaphore: GetPidIfNecessary, WaitComplete, and Post.

Here is an example of what is generated for a Wait semaphore call.

```
//Waite for Semaphore: pipelining.pipeline_phase_control
tb_pid = pipelining.pipeline_phase_control.GetPidIfNecessary(0);
if (tb_pid != 0)
  begin
    while(pipelining.pipeline_phase_control.resume_pid != tb_pid)
       begin
          @(pipelining.pipeline_phase_control.resume_pid);
       end
  end
pipelining.pipeline_phase_control.WaitComplete(tb_pid);
```

GetPidIfNecessary will return 0 if the semaphore's count is > 0, which means that the transaction doesn't have to wait for the semaphore to be posted (i.e. incremented). If GetPidIfNecessary returns a non-zero number then thta number represents the resume_pid to wait for. When a tbfifosemaphore is posted it will set the resume_pid to the next waiter in line (if there is one). Once the wait is complete, WaitComplete is called on the tbfifosemaphore which will decrement the count. When the transaction wants to release the semaphore, it simples calls Post which will increment the semaphore and set the appropriate resume_pid if necessary.

10.2 Waiting for Signal Transitions or Conditions

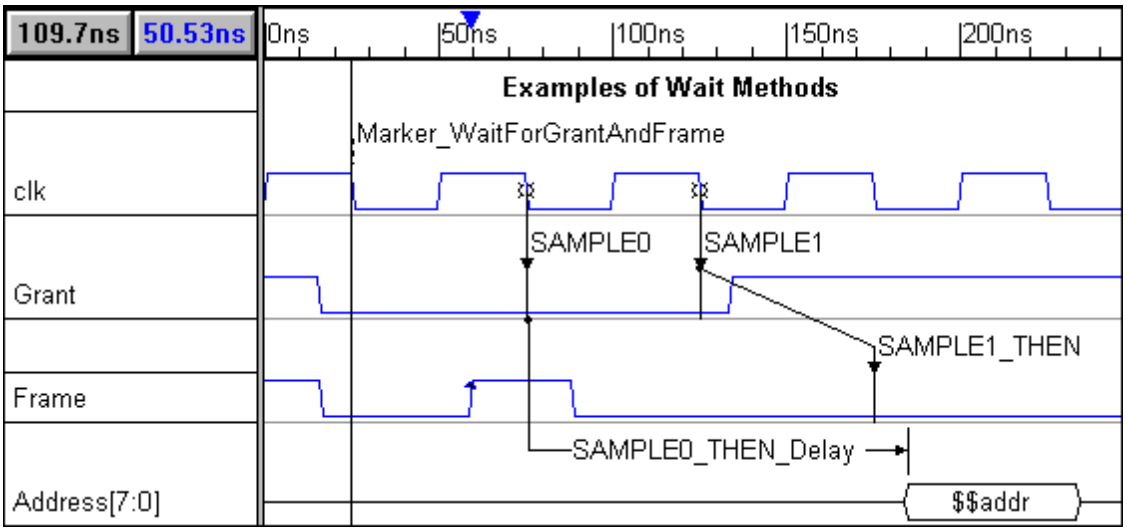
In TestBencher and Reactive Test Bench Generation, either **samples, markers,** or **sensitive edges** can be use to make a transaction wait for an event or series of events before continuing to execute. Below is a chart of the different methods of waiting and the recommended usage for each method.

Wait On	How Long	Construct	Help/Settings
One event	Block indefinitely or until diagram times out	**Sensitive Edge**	Section 2.6: Sensitive Edges 40
One or more conditions (e.g., signal states)	Block indefinitely until all conditions are true	**Wait Until Marker**	Section 5.4: Wait Until Marker 74
		Samples	Section 3.3: Samples that Block Clock Domains 54 to make a sample block, check **Blocking** and

Copyright © 2014, SynaptiCAD Sales, Inc.

			uncheck **Full Expect** (so it triggers when the condition is true) in the *HDL Code Generation* dialog.
One event or condition	Block until time out	Sample with window	Section 3.1: Adding a Sample 49 **min** != **max** **multiplier** == 1
One event or condition	Block until time out	Sample with Multiplier	Section 3.2: Sample Checking and Actions 51 **min** == **max** **multiplier** > 1
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	Section 3.4: Samples Triggering a delayed transition or another Sample 56

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 edges on the **Frame** signal are sensitive, and are drawn with the little blue arrow on the rising edges. This will cause the diagram to wait for the rising 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.
- **Sample0** 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

Copyright © 2014, SynaptiCAD Sales, Inc.

successive clock cycles. They are defined using the same settings as Sample0 in the previous example except the multiplier is set to 1.

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. This is because the Sample will execute and then immediately pass or fail depending on the condition (if the sample was windowed, it potentially blocks the clock domain until the sample's timeout time, but without a window, the "timeout" occurs immediately).

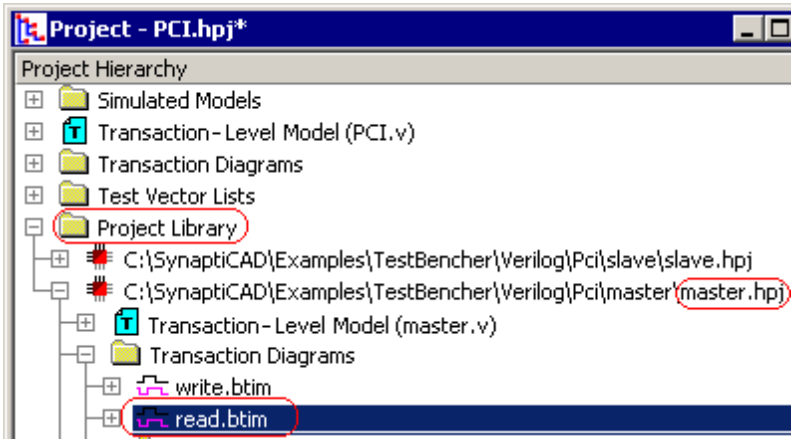
10.3 Burst Mode Transactions

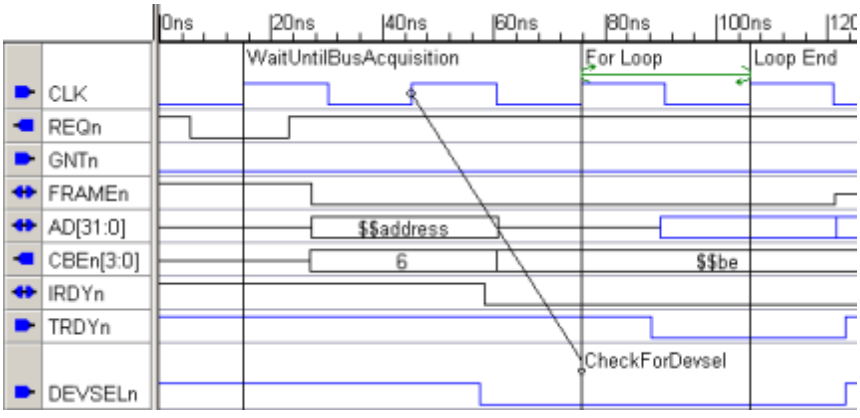
Burst Mode Transactions usually process one or more cycles depending on information received at simulation time. TestBencher can model burst mode transactions using a combination of marker loops, parameter variables, and samples. The basic design for a burst mode transaction is as follows:

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.

The **PCI example** in the **SynaptiCAD>Examples>TestBencher** directory has examples of both master and slave burst mode read transactions.

- After opening the PCI example for your language, drill down into the **Project Library** to find the **read cycle** of the **Master** sub-project.
- Below is an example of a simple burst mode master read cycle.





Copyright © 2014, SynaptiCAD Sales, Inc.

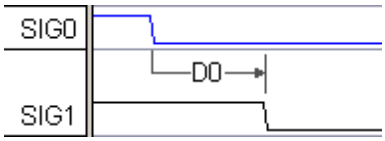
- The marker, **WaitUntilBusAcquisition**, will cause the transaction to wait until the condition *(GNTn === 0 && FRAMEn === 1 && IRDYn === 1)* is true (Section 5.4: Wait Until Markers 74).
- The sample, **CheckForDevsel**, checks the *DEVSELn* signal and if it is active low then the transaction proceeds normally. If it fails, the transaction aborts (Chapter 3: Transaction Samples 49).
- 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 6.1: Variables 82). The For Loop uses the *dataCycles* variable to determine how many read cycles to perform (Section 5.5: Loop Markers 75).
- Two state variables, *address* and *be*, are also transaction inputs (2.4 Waveform States with Variables and Conditions 36).

10.4 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. The **SweepTest2** project in the **SynaptiCAD>Examples>TestBencher** directory shows how an edge can be moved. Make sure to study both the timing diagrams and the Sequencer Code that calls the diagrams to figure out how the code works.

Add a delay that forces the edge that will be moved:

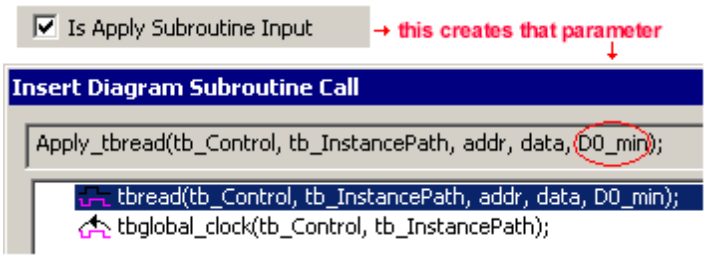
A **Delay** can be used to move and conditionally trigger signal edges. Each delay creates two variables named ***delayName*_min** and ***delayName*_max**, which specify either a fixed time or the number of clocking signal edges between two signal transitions. Section 4.1: Delays 61 describes how to add a delay, which is usually added between a blue input signal and the black output signal that will be forced.



When a Delay has both a min and a max time value, the actual value used at run time is determined by the **delay setting** in the *Diagram Settings* dialog (see Section 9.8 Diagram Settings dialog 148).The delay time can be set to be the minimum value, the maximum value, the average of the min and the max, or a randomly changing value between the min and the max value.

The delay value can be passed into the timing diagram Apply call

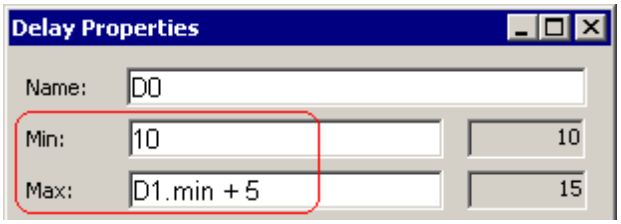
- If the delay value will be passed into the timing diagram, check the **Is Apply Subroutine Input** box in the *Delay Properties* dialog, to add the delay variable to the timing diagram apply call.



Copyright © 2014, SynaptiCAD Sales, Inc.

The delay value can be calculated within the timing diagram during simulation

- If the delay value will be calculated within the timing diagram during simulation, enter the **min** or **max** times or time formulas like **D1.min + 5**.



Time formulas can be based on either diagram level variables, variables from other delays, setups, holds or samples, or project level variables that are passed into the diagram. See Chapter 6: Classes and Variables 82 and also the **Timing Diagram Editor's Manual** Section 2.5: Time Formulas for Clocks and Parameters for information on the syntax of formulas.

Add a Setup, Hold, or Sample to Monitor the transaction behavior as the edge is moved:

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 4.2: Setups and Holds 65).
- Continuous setups monitor all of the edge transitions during a transaction 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 4.3: Creating Continuous Setups and Holds 67.
- A windowed, full-expect sample can be used monitor the MUT response. This type of sample checks to make sure that the MUT's signals either transition or stay stable within a window of time (Chapter 3: Transaction Samples 49).

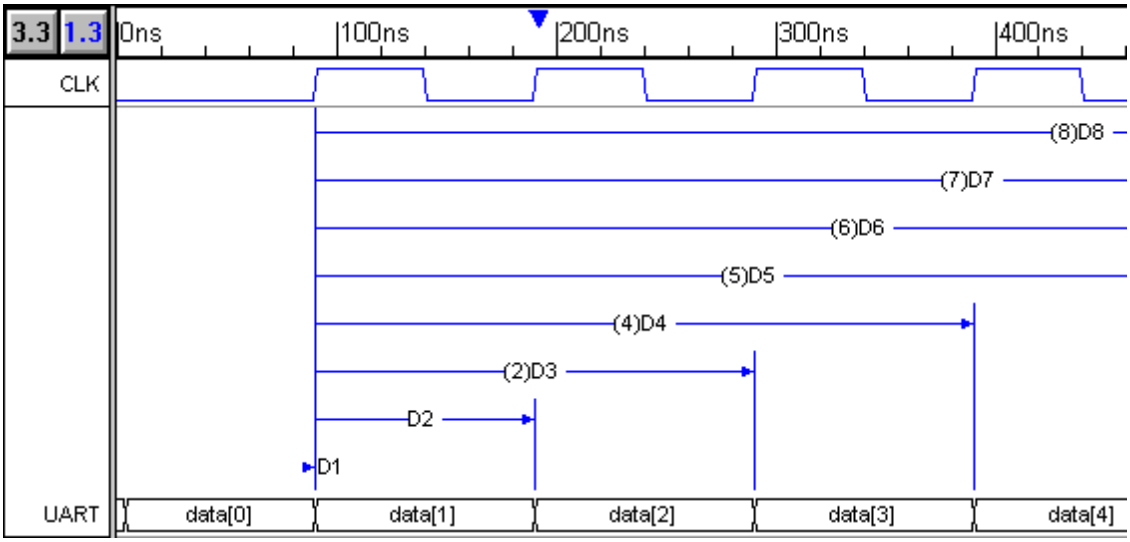
10.5 Reading and Writing Serial Data

To read and write serial data, you can 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 **SynaptiCAD>Examples>TestBencher** directory is an excellent example of manipulating serial data.

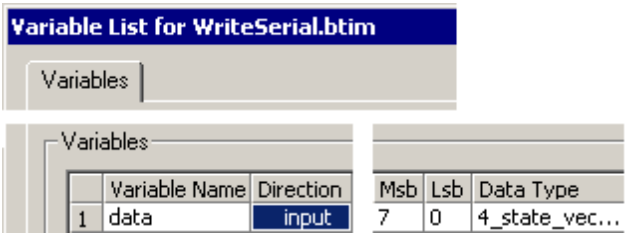
Setup Write Diagram to write out one bit at a time:

Below is the UART **write** timing diagram that takes a byte it receives from the top-level project and serializes the data to the MUT.

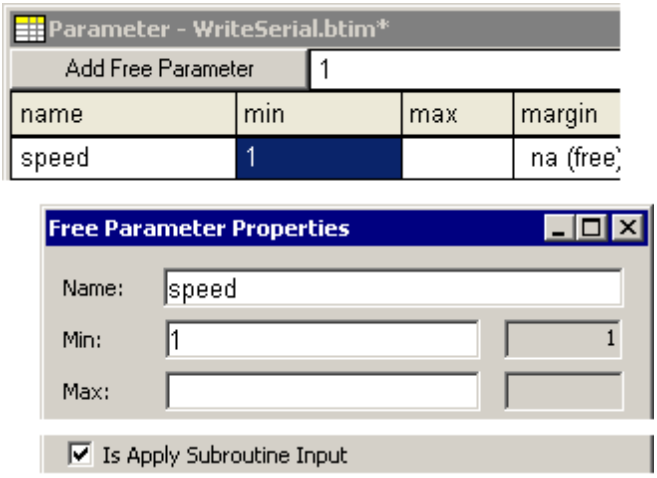
Copyright © 2014, SynaptiCAD Sales, Inc.



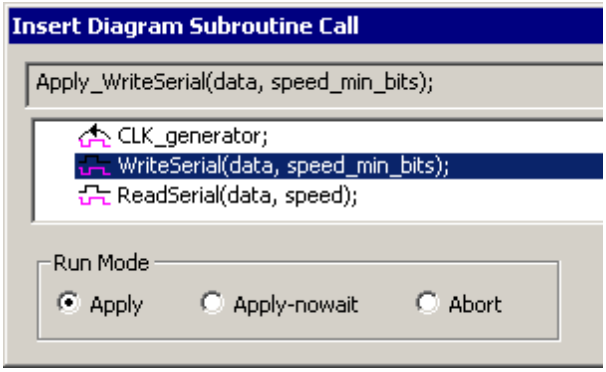
- The diagram-level variable, **data**, is an input to the transaction. It will hold the byte data to be serialized. Press the **View Variables** button on the diagram window to open the *Variable List* dialog



- A free parameter variable called **speed** determines the number of clock cycles needed to write the bit of data. It was added by pressing the **Add Free Parameters** button in the Parameter Window.



- Making **input** the direction of **data** variable, causes **data** to become an input parameter to the diagram apply call.
- Checking **Is Apply Subroutine Input** in the **speed** parameter's Properties dialog causes **Speed** to become an input parameter



Copyright © 2014, SynaptiCAD Sales, Inc.

- 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** which controls how many clock cycles that they should delay.

Parameter - WriteSerial.btim*

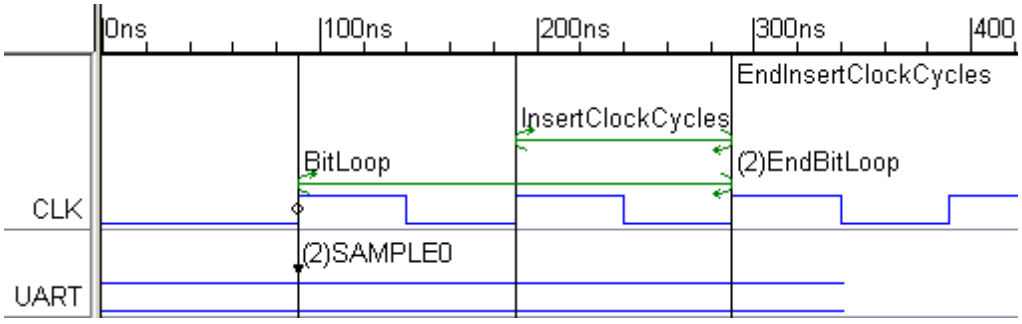
Add Free Parameter

name	min	max	margin
speed	1		na (free)
D0	0		na (free)
D1	speed-1.0		na (delay
D2	speed*2.0-1.0		na (delay
D3	speed*3.0-1.0		na (delay
D4	speed*4.0-1.0		na (delay
D5	speed*5.0-1.0		na (delay
D6	speed*6.0-1.0		na (delay
D7	speed*7.0-1.0		na (delay
D8	speed*8.0-1.0		na (delay

ReadSerial.btim | WriteSerial.btim

The Read Diagram re-packs the serial bits back into a byte:

The read diagram will read in the serial bits and repack them back in to a byte variable that will be exported as an output to the timing diagram. Below are two examples on how to do this. The first uses a series of loops and a signal sample to make the diagram.



- This diagram has two variables, **data** and **speed** that are both parameters to the timing diagram apply call. Press the **View Variables** button on the diagram window to open the *Variable List* dialog.

Variable List for ReadSerial.btim

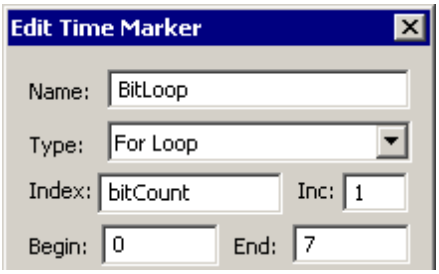
Variables

Variables

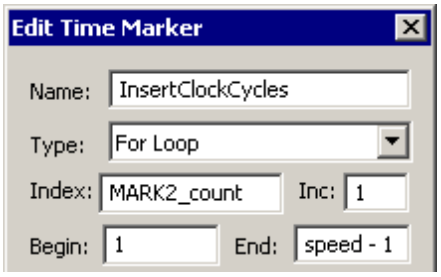
	Variable Name	Direction	Msb	Lsb	Data Type
1	data	output	7	0	4_state_vec...
2	speed	input	31	0	integer

Copyright © 2014, SynaptiCAD Sales, Inc.

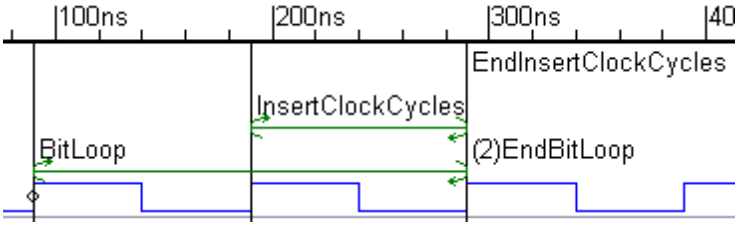
- The BitLoop marker begins a for-loop that counts over the number of bits in the data stream.



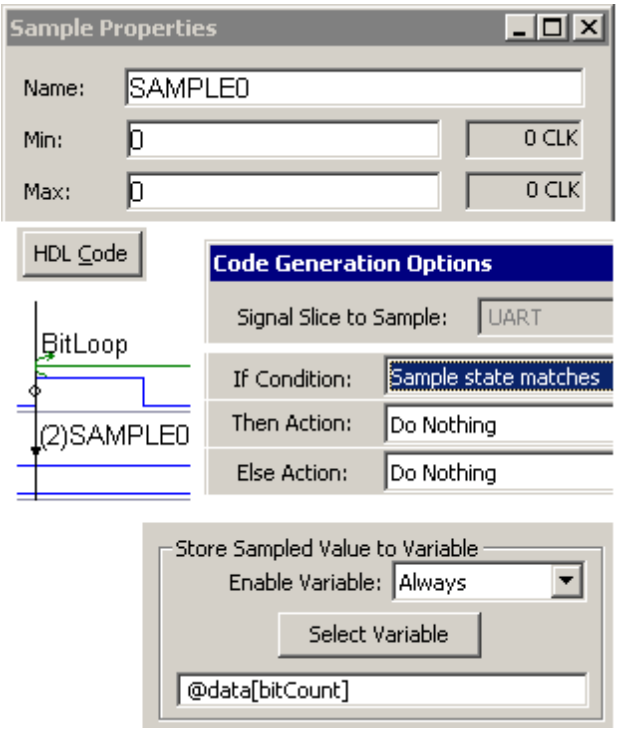
- The InsertClock Cycles marker begins a for-loop that will add in clock cycles based on the **speed** variable. The **speed** variable is passed into the write cycle apply subroutine call.



- Notice that the **(2)** beside **EndBitLoop** indicates that it triggers after the **End Insert Clock Cycles** marker. The triggering order is set by double clicking on the edge. The Bit loop surrounds the InsertClockCycles loop.

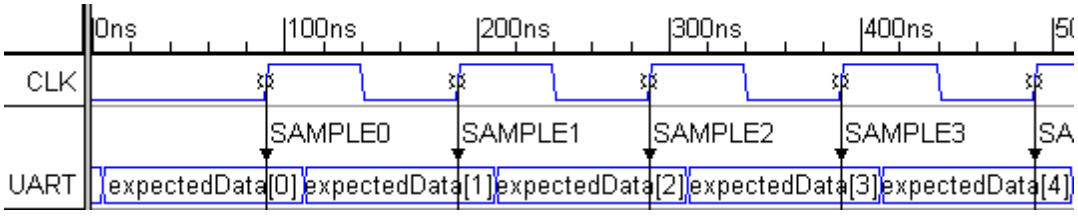


- Sample0 is triggered from the same edge as BitLoop so that both are in the same clock domain. The **(2)** beside Sample0 indicated that it is inside t he BitLoop (triggers after).
- Each time that BitLoop executes, Sample0 will sample the value on the UART bus. Then only thing that Sample0 does is write the bit to the **data** variable. The **bitCount** variable is the index variable created by the **BitLoop** marker.



Another way to design the read UART is to fully draw out the timing diagram.

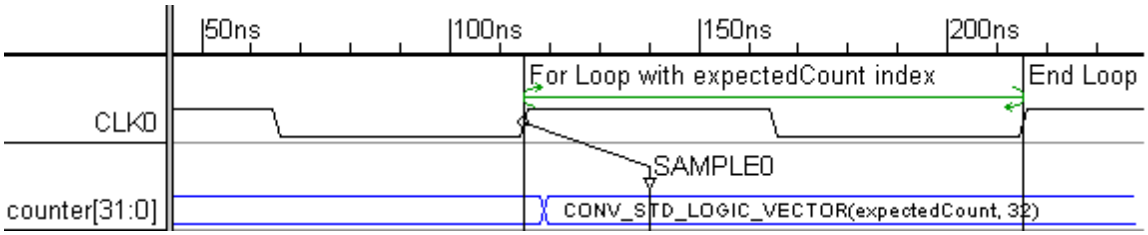
Copyright © 2014, SynaptiCAD Sales, Inc.



- There is an output variable called ***data*** to hold the data coming back from the MUT
- There is an an input variable called ***expectedData*** that is 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 widow, 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.

10.6 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:

- **Counter Loop:** two loop markers surround a clock 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.

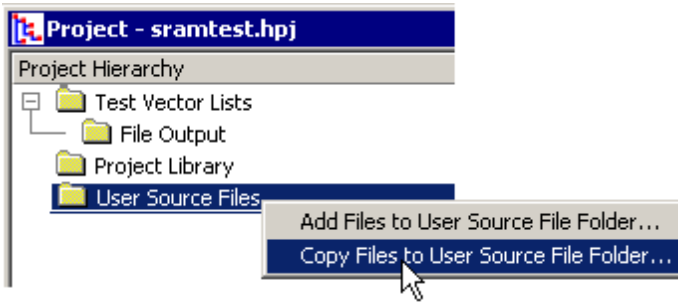
Copyright © 2014, SynaptiCAD Sales, Inc.

10.7 Different ways to bring in External Models

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

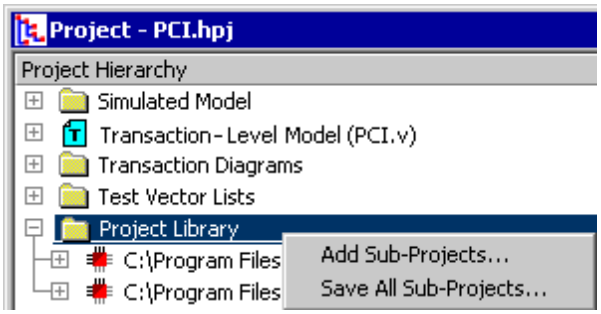
VHDL and Verilog Models

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 Step 2: Add the MUT to the Project 12 and Step 3: Extract Ports into Template Diagram 14.



Other TestBencher TLMs

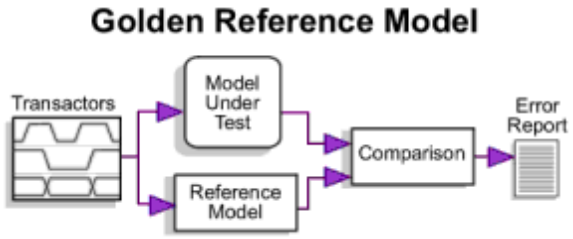
Other TestBencher TLMs are first added to the Project Library folder and then instantiated using the context menu functions. This process is detailed in Section 9.1: Sub-Projects 137 and Section 9.2: Component Instances of Sub-Projects 139. The TLMs should be of the same generation language as the main TestBencher project.



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 9.3: Golden Reference Models 140.

Copyright © 2014, SynaptiCAD Sales, Inc.

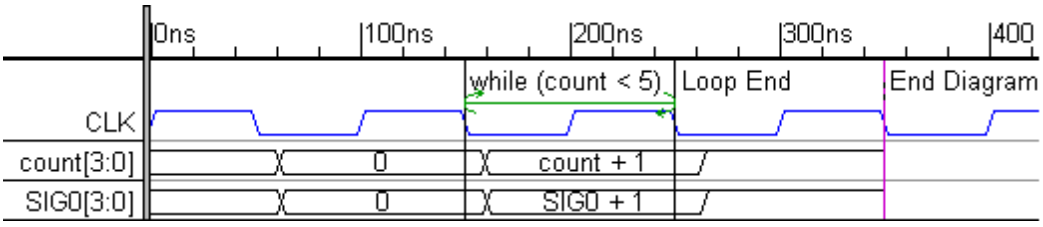


10.8 Different ways to Loop

There are three different ways to loop.

1) Use Markers to loop within a diagram

Markers can be used inside a timing diagram to create a loop around a specific clock domain (like all objects and waveforms triggered off the negative edge of a clock). Section 5.5: Loop Markers 75 has more information on the creating the loops and setting the clock domains.



2) Use code at the Sequencer Lever to repetitively call a transaction

Inside the Sequencer Process of the Transaction-Level Model, you can repetitively call a transactor by writing loop code in the generation language. Below is code excerpt from the **Sweeptest2.hpj** examples located in the **SynaptiCAD\Examples\TestBencher** directories. In these examples, each time the loop executes it calls a write then a read transaction using a different delay value to see when the circuit starts to fail. See Section 7.1: Template, Sequencer, and Transaction Calls 112 for more information.

Verilog

```
// Sequencer Process
for (delay0 = 32.0; delay0 > 15.0; delay0 = delay0 - 5.0)
  begin
    Apply_tbwrite( 'h55 , 'hee , $realtobits(delay0) );
    Apply_tbread( 'h55 , 'hee , $realtobits(delay0) );
  end
```

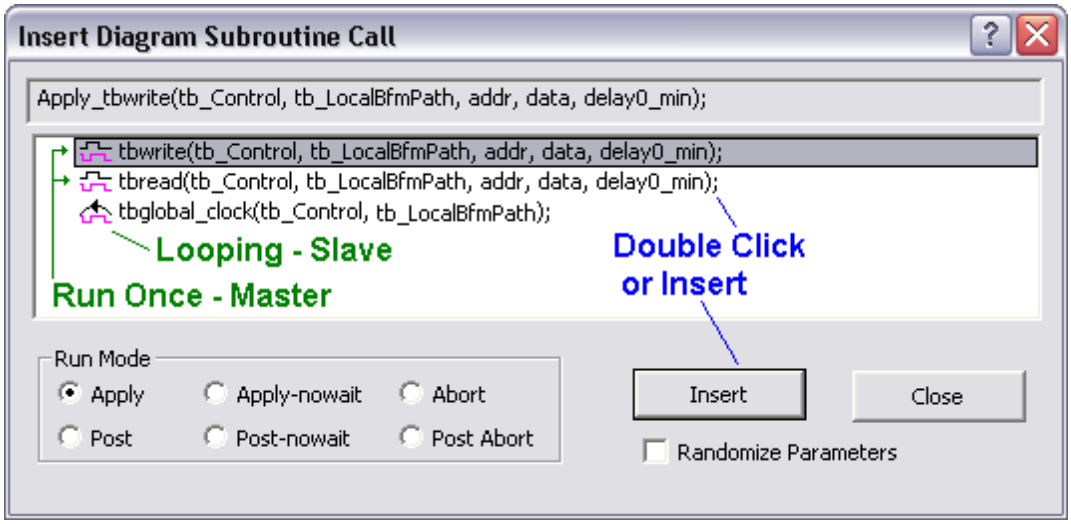
VHDL

```
-- Sequencer Process
for i in 0 to 5 loop
  delay0 := real (32 - (i * 5));
  Apply_tbwrite( tb_Control, tb_instancePath, x"F0", x"AE", delay0);
  Apply_tbread( tb_Control, tb_instancePath, x"F0",  x"AE", delay0);
end loop;
```

3) Apply the transaction in Looping Mode and make an abort call.

Inside the Sequencer Process of the Transaction-Level Model, you will make calls to apply the transactions. A transaction can be applied in a looping mode that will make it continually run. Section 7.1: Template, Sequencer, and Transaction Calls 112 has more information on the different types of apply calls. Looping transactions need to be aborted after the testing is complete.

Copyright © 2014, SynaptiCAD Sales, Inc.



Verilog

```
// Sequencer Process
Apply_tbglobal_clock_looping_nowait;
    ...
Abort_tbglobal_clock;
```

10.9 Scoreboarding

Scoreboarding is the process of tracking which transactions have been applied to the model under test (MUT). One way to do scoreboarding is to create a "slave diagram" that you launch near the start of the test bench that reacts to the same read/write protocol as the model under test. This diagram would therefore run in parallel with your MUT and would snoop on the bus to save off all the writes written to the memory to either an array or an associate array. For small address spaces (e.g. 64k or less) an array can be used, but for larger address spaces it is better to use an associative array.

Simple Test Benches

In a simple testbench, you would create the array in the "class library" for the project and have the slave diagram save data to this memory using a sample to capture the address and data values at the appropriate points during the recognition of a write cycle (any diagram in a project can read/write the project level variables).

Complicated Test Benches

In a more complicated system with several memories being tested at once, you would probably want to make a separate project file (*.hpj file) to act as the "golden reference"/"executable specification" ram model. This would create an entire "ram" component that could be instantiated multiple times in an upper level project file.

An Asynchronous Example:

There is an example of this, in the **Examples > TestBencher > VHDL > VME** directory, where an upper level project, **VME.hpj**, contains two different types of subcomponents (**VME_slave.hpj** and **VME_master.hpj**). To view this example, open up **VME.hpj**, then drill into the **VME_slave.hpj** file under the "Project Library" folder. If you look into the ***Variable List*** under ***Component Model*** for VME_slave, you'll see there's a variable called **memory** defined here. It's a simple 65k element array of 4 state values. It's written to by the slave's write diagram whenever the write diagram detects a write bus transaction.

The **write** diagram is the key piece of this example, as it shows one way to create a slave that "recognizes" a write transaction on an asynchronous bus like VME using "sensitive edges". For VME,

Copyright © 2014, SynaptiCAD Sales, Inc.

the address strobe (**AS**) and the data strobe were made sensitive edges to indicate the diagram should "wait" until these events occur. To make the rising or falling edges of a signal sensitive, double click on the signal name and click the appropriate check box at the bottom of the *Signal Properties* dialog. Any drawn edges of the sensitive type will then generate an appropriate "wait" statement in the VHDL code at either the relative time or during that relative clock cycle (depending on whether the signal is "unclocked" or "clocked"). Section 2.6 Sensitive Edges - a Blocking Construct 40 covers sensitive edges and has links to other methods for waiting for events in a diagram.

After the address strobe occurs (the wait condition is met), the sample triggered by the address strobe checks the address to verify it's within the range of the SRAM, otherwise the diagram ignores the write cycle (by "restarting" and waiting for another address strobe).

The other important sample is the one triggered when the data strobe falls. This one verifies that it's actually a write cycle (if it's not the diagram is "restarted" and waits for another address strobe).

A Synchronous Example

The PCI example, located in the **Examples > TestBencher > VHDL > PCI** directory, is very similar to the VME example, except it demonstrates monitoring a synchronous bus rather than an asynchronous one. For fastest simulation, the write diagram in the slave of this example waits for a falling edge on FRAME signal (which indicates a cycle of some sort is beginning), then samples the address on the next clock cycle after the frame to verify we have an address with the address space of the device.

The PCI write example also demonstrates how to capture the data from a "burst write" using a loop. To gain an understanding of how it works, it is best to double click on each marker and also on the samples in the diagram and press the **HDL Code** button on the *Sample Properties* dialog. The **SampleData** sample does the actual capture of the data being written to the bus, and it saves it out to an array of 32 bit words:

```
@sram.data[CONV_INTEGER(UNSIGNED(DecodeAddress[4:0]))]
```

Note it's using the **DecodeAddress** it captured in the **DecodeAddress** sample to write to the proper location in the array.

Copyright © 2014, SynaptiCAD Sales, Inc.

Chapter 11: Language Specific Details and Test Bench Architecture

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 and the apply calls within it change from language to 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.

Language Specific Apply Calls

Verilog
```
// Apply_test(addr, data);
   Apply_test(addr, data);
```
TestBuilder
```
// Tvm.Apply_test.DefaultArgs: addr, data
   Tvm.Apply_test.run( /*DefaultArgs*/ );
```
VHDL
```
-- Apply_test(tb_Control, tb_LocalBfmPath, addr, data);
   Apply_test(tb_Control, tb_LocalBfmPath, addr, data);
```

11.1 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 Transaction-Level Model, TLM, 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 9.3 Golden Reference Models 140) then a file named *projectName*_emulator_skeleton.v is generated and placed in the Associated Files folder for the Transaction-Level 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.

Copyright © 2014, SynaptiCAD Sales, Inc.

- 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 Transaction-Level 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 Transaction-Level 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 example, 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.

Third Party Simulators

When you launch a third party simulator from the command line, you may need to specify that there are two top-level modules created by TestBencher. The first is **tbench** which contains all of the transaction code. The other module is **tbsyslog** which contains the code for logging errors and warnings. If you are launching the simulator from within the GUI environment, then this will automatically be handled. Also SynaptiCAD's command line simulators are smart enough to detect that there are two top level modules and set the design up automatically.

11.2 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 Transaction-Level 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

Copyright © 2014, SynaptiCAD Sales, Inc.

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.
- The following files can be accessed from the project tree under **Transaction-Level Model > Associated Files**.
  - **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.
  - **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 Transaction-Level 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 Transaction-Level 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 Transaction-Level Model of the top-level project during test bench generation.
- There also some package files (collectively know as the syncad library) 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.
- TestBencher 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 is a global variable, **tb_control**, that is used to control the triggering of transactors and this additional parameter must be passed to Apply and Post calls..

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 7.1: Template, Sequencer, and Transaction Calls 112). However you may type in the apply calls by hand. To apply a diagram that is

Copyright © 2014, SynaptiCAD Sales, Inc.

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 Transaction-Level 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 automatically for you.

Each timing diagram generates several apply call procedures:

Master Transaction Apply Calls:

```
Apply_diagramName ( signal tb_Control : inout tb_Control_Type;
                    transactorBfmPath : in string;
                    <parameters> );
Apply_diagramName_nowait ( signal tb_Control : inout tb_Control_Type;
                           transactorBfmPath : in string;
                           <parameters> );
Abort_diagramName ( signal tb_Control : inout tb_Control_Type;
                    transactorBfmPath : in string);
```

Slave Transaction Apply Calls:

```
Apply_diagramName_looping ( signal tb_Control : inout tb_Control_Type;
                            transactorBfmPath : in string;
                            <parameters> );
Apply_diagramName_looping_nowait( signal tb_Control : inout tb_Control_Type;
                                  transactorBfmPath : in string;
                                  <parameters> );
Abort_diagramName ( signal tb_Control : inout tb_Control_Type;
                    transactorBfmPath : in string);
```

Master Transaction Manager Post Calls:

```
Post_diagramName ( signal tb_Control : inout tb_Control_Type;
                   queuePath : in string;
                   transactorBfmPath : in string;
                   <parameters> );
Post_diagramName_nowait ( signal tb_Control : inout tb_Control_Type;
                   queuePath : in string;
                   transactorBfmPath : in string;
                   <parameters> );
Post_Abort_diagramName ( signal tb_Control : inout tb_Control_Type;
                         queuePath : in string;
                         transactorBfmPath : in string );
```

Slave Transaction Manager Post Calls:

```
Post_diagramName_looping ( signal tb_Control : inout tb_Control_Type;
                   queuePath : in string;
                   transactorBfmPath : in string;
                   <parameters> );
Post_diagramName_looping_nowait ( signal tb_Control : inout tb_Control_Type;
                   queuePath : in string;
                   transactorBfmPath : in string;
                   <parameters> );
Post_Abort_diagramName ( signal tb_Control : inout tb_Control_Type;
                         queuePath : in string;
                         transactorBfmPath : in string );
```

Copyright © 2014, SynaptiCAD Sales, Inc.

Call to Randomly Post a Master Transaction to the Transaction Manager Queue:

```
PostRandomTransactionType (signal tb_Control : inout tb_Control_Type;
                           transactorBfmPath : in string);
```

Calls to Read Calls from a File:

See Section 7.5 VHDL: TM fill from File 120 for syntax on the file.

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.

transactorBfmPath - the hierarchical path to the project which contains the transactor you wish to apply. You can type in the full instance path or use the generic **tb_LocalBfmPath** that contains the project instance path for transactors relative to the current entity.

queuePath - path to project that contains the transaction manager queue that you want to post the apply call to. Typically, this will be **tb_LocalBfmPath** which is a generic available in all project components and diagrams that specifies the full instance path to the current scope. So, tb_LocalBfmPath 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 a list of transactions.

<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_LocalBfmPath, 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_LocalBfmPath & "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 7.6: TM fill from Random Generator 121.

- 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_LocalBfmPath, ((0, 5, 1),
                                                   (0, 1, 1),
                                                   (0, 1, 1)));
```

- The **SetTransactorWeightings** function registers the weightings matrix. You can change the random weightings at any time during the simulation.
- The **PostRandomTransactorType** function randomly creates a transaction call and hands it

Copyright © 2014, SynaptiCAD Sales, Inc.

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
  PostRandomTransactorType(tb_Control, tb_LocalBfmPath);
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.

11.3 TestBuilder

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.

Files needed for simulation:

- ***projectName.*cpp/h** - contains top-level module for the project. This is the Transaction-Level model file (the expanded template file).
- For pure TestBuilder projects: one timing transaction file and one 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.

Copyright © 2014, SynaptiCAD Sales, Inc.

- **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 Transaction-Level 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 9.3: Golden Reference Models 140) then a file named *projectName*_emulator_skeleton.cpp is generated and placed in the Associated Files folder for the Transaction-Level 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

When TestBuilder Integration is enabled, the parameters (e.g., Applied Parameters and State Variables) for the Apply calls are placed into an **Arguments** class. This class is then used to pass data to and from the apply calls. If no arguments are specified for an apply call then the DefaultArgs member of the apply call class will be used. The DefaultArgs can be accessed and set before running a transaction in this case.

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 *Transaction-Level 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 92 for more information on constraints. To randomize a variable or class instance:

- Double-click on the **Transaction-Level 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

Copyright © 2014, SynaptiCAD Sales, Inc.

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 Transaction-Level 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` `( tbvTaskT* 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 tbvTaskT object manually. Each transaction has three corresponding classes defined that derives from tbvTaskT and are named TApply_projectName_diagramName, TApply_projectName_diagramName_looping, TAbort_projectName_diagramName.

Copyright © 2014, SynaptiCAD Sales, Inc.

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();
  }
```

Copyright © 2014, SynaptiCAD Sales, Inc.

Appendix A: Editor Commands

Listed below are the keyboard and mouse commands supported by the editor windows. TestBencher also supports complete editing and debugging integration with the popular XEmacs text editor. This is covered in the BugHunter Manual under Section 4.6: XEmacs Integration and other External Editors.

Jump To, Searching, and Printing:

Key	Purpose
Ctrl+G	Jump to line#
Ctrl+F	Search and Replace in the selected file
F3	Search Again
Ctrl+Shift+F	Search an through an entire directory, use the **Editor > Find in Files...** menu. The results are displayed in the *Report* window **Grep** tab.
F4	Print from window
Shift+F4	Print options

Copy and Undo Commands:

Key	Purpose
Ctrl+X	Cut
Ctrl+C	Copy
Ctrl+V	Paste
Ctrl+Z	Undo
Ctrl+Y	Redo

Selection Commands:

Key	Purpose
Ctrl+A	Selects all of the text in the document
Shift+Left	Selects text one character at a time to the left
Shift+Right	Selects text one character at a time to the right
Shift+Down	Selects one line of text down
Shift+Up	Selects one line of text up
Shift+End	Selects text to the end of the line
Shift+Home	Selects text to the beginning of the line

Copyright © 2014, SynaptiCAD Sales, Inc.

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

Moving and Editing Commands:

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

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

Copyright © 2014, SynaptiCAD Sales, Inc.

Appendix B: Supported Simulators

TestBencher works with the simulators that are listed below. See Step 7: Setup External Simulators 21 to setup the program to work with the external simulator.

Windows simulators and languages:

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 waveforms when ModelSim VHDL XE/PE 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

Solaris UNIX simulators and languages:

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.

Copyright © 2014, SynaptiCAD Sales, Inc.

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	Hierarchically 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
Operator Name	**Equations**	**States**	**Autogen**	**Syncad**	**Example**	**Definition**
Bit Slice	yes	yes	yes	: or to or dow nto	a[3:0]	Used w ith Selection operator to select a range of bits
Operator Name	**Equations**	**States**	**Autogen**	**Syncad**	**Example**	**Definition**
Class	no	yes	no	@	@class1	Specifies that the name follow ing @ is an instance of a user defined class 82
State Variable	no	yes	no	$$	$$addr	Specifies that the name follow ing $$ is a State Variable 82
Macro Definition	no	yes	no	`	`base_address	Refers to a defined macro
Operator Name	**Equations**	**States**	**Autogen**	**Syncad**	**Example**	**Definition**
Logical Negation	yes	yes	yes	! (any)	!a	Converts a nonzero operand into 0 and a

Copyright © 2014, SynaptiCAD Sales, Inc.

						zero operand into 1
Bit-wise Negation	yes	yes	yes	~ (any)	~a	Performs bit-wise negation of operand
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
Operator Name	**Equations**	**States**	**Autogen**	**Syncad**	**Example**	**Definition**
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	% (any)	a % b	Performs division operation and returns the remainder
Operator Name	**Equations**	**States**	**Autogen**	**Syncad**	**Example**	**Definition**
Addition	yes	yes	yes	+	a + b	Adds left and right operands
Subtraction	yes	yes	yes	-	a - b	Subtracts right operand from left operand
Operator Name	**Equations**	**States**	**Autogen**	**Syncad**	**Example**	**Definition**
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

Copyright © 2014, SynaptiCAD Sales, Inc.

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

Copyright © 2014, SynaptiCAD Sales, Inc.

Negative String Matching	no	yes	no	!~	a !~ b	See e Language Reference manual
Operator Name	**Equations**	**States**	**Autogen**	**Syncad**	**Example**	**Definition**
Bit-w ise AND	yes	yes	yes	&	a & b	Performs bit-w ise AND of tw o operands
Reduction AND	yes	yes	yes	&	&a	Performs bit-w ise AND of one operand and results in a single bit
Reduction NAND	yes	yes	yes	~&	~&a	Performs bit-w ise NAND of one operand and results in a single bit
Operator Name	**Equations**	**States**	**Autogen**	**Syncad**	**Example**	**Definition**
Bit-w ise XOR	yes	yes	yes	^	a ^ b	Performs bit-w ise XOR of tw o operands
Reduction XOR	yes	yes	yes	^	^a	Performs bit-w ise XOR of one operand and results in a single bit
Bit-w ise XNOR	yes	yes	yes	^~ or ~^	a ^~ b	Performs bit-w ise XNOR of tw o operands
Reduction XNOR	yes	yes	yes	^~ or ~^	^~a	Performs bit-w ise XNOR of one operand and results in a single bit
Operator Name	**Equations**	**States**	**Autogen**	**Syncad**	**Example**	**Definition**
Bit-w ise OR	yes	yes	yes	|	a | b	Performs bit-w ise OR of tw o operands
Reduction OR	yes	yes	yes	|	|a	Performs bit-w ise OR of one operand and results in a single bit
Bit-w ise NOR	yes	yes	yes	nor (any)	a |~ b	Performs bit-w ise NOR of tw o operands
Reduction NOR	yes	yes	yes	~|	~|a	Performs bit-w ise NOR of one operand and results in a single bit
Bit-w ise NAND	yes	yes	yes	nand	a nand b	Performs bit-w ise NAND of tw o operands
VHDL Bit-w ise XNOR	yes	yes	yes	xnor	a xnor b	Performs bit-w ise XNOR of tw o operands
VHDL Bit-w ise XOR	yes	yes	yes	xor	a xor b	Performs bit-w ise XOR of tw o operands
VHDL Bit-w ise OR	yes	yes	yes	or	a or b	Performs bit-w ise OR of tw o operands
VHDL Bit-w ise AND	yes	yes	yes	and	a and b	Performs bit-w ise AND of tw o operands

Copyright © 2014, SynaptiCAD Sales, Inc.

Operator Name	Equations	States	Autogen	Syncad	Example	Definition
Logical AND	yes	yes	yes	&&	a && b	If both operands are non-zero then result is 1, otherwise result is 0
Operator Name	**Equations**	**States**	**Autogen**	**Syncad**	**Example**	**Definition**
Logical OR	yes	yes	yes	||	a || b	If either operand is non-zero then result is 1, otherwise result is 0
Operator Name	**Equations**	**States**	**Autogen**	**Syncad**	**Example**	**Definition**
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
Operator Name	**Equations**	**States**	**Autogen**	**Syncad**	**Example**	**Definition**
Ternary	yes	yes	yes	? : (any)	(condition) ?a:b	If condition is TRUE then result is a, otherwise result is b.
Operator Name	**Equations**	**States**	**Autogen**	**Syncad**	**Example**	**Definition**
Delay	yes	no	no	delay	SIG0 delay 5	Delays left operand by time specified by right operand

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 shown in the Syncad column. 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. In the language specific columns in the table below, cells the 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
Bit Replication	{{ }}	{{ }}	n/a	n/a	n/a	n/a

Copyright © 2014, SynaptiCAD Sales, Inc.

Selection	[ ]	[ ]	[ ]	[ ]	[ ]	( )
Hierarchy	.	.	n/a	. or ~/ /	.	.
List Slice	..	n/a	n/a	..	n/a	n/a
List Concatenation	{ ; }	n/a	n/a	{ ; }	n/a	n/a
Operator Name	**Syncad**	**Verilog**	**VHDL**	**e**	**OpenVera**	**TestBuilder**
Bit Slice	: or to or dow nto	:	to or downto	:	:	, (comma)
Operator Name	**Syncad**	**Verilog**	**VHDL**	**e**	**OpenVera**	**TestBuilder**
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	`		
Operator Name	**Syncad**	**Verilog**	**VHDL**	**e**	**OpenVera**	**TestBuilder**
Logical Negation	! (any)	!	not	!	!	!
Bit-w ise 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	-	-	-	-	-	-
Operator Name	**Syncad**	**Verilog**	**VHDL**	**e**	**OpenVera**	**TestBuilder**
Multiply	*	*	*	*	*	*
Divide	/	/	/	/	/	/
Modulo	mod	n/a	mod	n/a	n/a	n/a
Remainder	% (any)	%	rem	%	%	%
Operator Name	**Syncad**	**Verilog**	**VHDL**	**e**	**OpenVera**	**TestBuilder**
Addition	+	+	+	+	+	+
Subtraction	-	-	-	-	-	-
Operator Name	**Syncad**	**Verilog**	**VHDL**	**e**	**OpenVera**	**TestBuilder**

Copyright © 2014, SynaptiCAD Sales, Inc.

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
Operator Name	**Syncad**	**Verilog**	**VHDL**	**e**	**OpenVera**	**TestBuilder**
Less Than	<	<	<	<	<	<
Less Than or Equal	<=	<=	<=	<=	<=	<=
Greater Than	>	>	>	>	>	>
Greater Than or Equal	>=	>=	>=	>=	>=	>=
Operator Name	**Syncad**	**Verilog**	**VHDL**	**e**	**OpenVera**	**TestBuilder**
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
Operator Name	**Syncad**	**Verilog**	**VHDL**	**e**	**OpenVera**	**TestBuilder**
Equality	==	==	=	==	==	==
Inequality	any	!=	/=	!=	!=	!=
Case Equality	any	===	n/a	===	===	n/a
Case Inequality	any	!==	n/a	!==	!==	n/a
Wild Equality	=?=	n/a	n/a	n/a	=?=	n/a
Wild Inequality	!?=	n/a	n/a	n/a	!?=	n/a
Operator Name	**Syncad**	**Verilog**	**VHDL**	**e**	**OpenVera**	**TestBuilder**
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

Copyright © 2014, SynaptiCAD Sales, Inc.

Operator Name	Syncad	Verilog	VHDL	e	OpenVera	TestBuilder
Bit-wise AND	&	&	n/a	&	&	&
Reduction AND	&	&	n/a	n/a	&	.reductionAnd()
Reduction NAND	~&	~&	n/a	n/a	~&	.reductionNand()
Operator Name	**Syncad**	**Verilog**	**VHDL**	**e**	**OpenVera**	**TestBuilder**
Bit-wise XOR	^	^	n/a	^	^	^
Reduction XOR	^	^	n/a	n/a	^	.reductionXor()
Bit-wise XNOR	^~ or ~^	^~ or ~^	n/a	generates: ~(x ^ y)	^~ or ~^	generates: ~(x ^ y)
Reduction XNOR	^~ or ~^	^~ or ~^	n/a	n/a	^~ or ~^	.reductionXnor()
Operator Name	**Syncad**	**Verilog**	**VHDL**	**e**	**OpenVera**	**TestBuilder**
Bit-wise OR	|	|	n/a	|	|	|
Reduction OR	|	|	n/a	n/a	|	.reductionOr()
Bit-wise NOR	nor (any)	generates: ~(x | y)	nor	generates: ~(x | y)	|~	generates: ~(x | y)
Reduction NOR	~|	~|	n/a	n/a	~|	.reductionNor()
Bit-wise NAND	nand	generates: ~(x & y)	nand	generates: ~(x & y)	&~	generates: ~(x & y)
VHDL Bit-wise XNOR	xnor	generates: x ~^ y	xnor	n/a	n/a	n/a
VHDL Bit-wise XOR	xor	generates: x ^ y	xor	n/a	n/a	n/a
VHDL Bit-wise OR	or	generates: x | y	or	n/a	n/a	n/a
VHDL Bit-wise AND	and	generates: x & y	and	n/a	n/a	n/a
Operator Name	**Syncad**	**Verilog**	**VHDL**	**e**	**OpenVera**	**TestBuilder**

Copyright © 2014, SynaptiCAD Sales, Inc.

Logical AND	&&	&&	n/a	&&	&&	&&
Operator Name	**Syncad**	**Verilog**	**VHDL**	**e**	**OpenVera**	**TestBuilder**
Logical OR	||	||	n/a	||	||	||
Operator Name	**Syncad**	**Verilog**	**VHDL**	**e**	**OpenVera**	**TestBuilder**
Boolean Implication	=>	generates: (!x) || y	generates: (not x) or y	=>	generates: (!x) || y	generates: (!x) || y
Boolean Event Check	now @	n/a	n/a	now @	n/a	n/a
Operator Name	**Syncad**	**Verilog**	**VHDL**	**e**	**OpenVera**	**TestBuilder**
Ternary	? : (any)	? :	when else	? :	? :	? :
Operator Name	**Syncad**	**Verilog**	**VHDL**	**e**	**OpenVera**	**TestBuilder**
Delay	delay	#	transport or after	n/a	n/a	n/a

Copyright © 2014, SynaptiCAD Sales, Inc.

Index

- 2 -

2_state 109
2_state_vector 109

- 4 -

4_state 109
4_state_vector 109

- A -

Abort Transaction 112
Absolute Markers 70
Absolute Samples
  add to diagram 49
  definition 49
Advanced Register and Latch Controls dialog 67
Apply Call
  no-wait 112
  wait 112
Apply Statements 17, 171, 175
  adding to template file 17
  in TestBuilder 175
  in Verilog 170
  in VHDL 171
ApplyFile 120
ApplyFile_nowait 120
ApplyNextTransaction 120
array 107
  Verilog Initialize from file 102
  VHDL Initialize from file 104
Arrays 82
associative array 107
Associative Arrays 82
Attach to Edge 70
Attach to Time 70
Auto Parse Project on Load 129
Auto Run 134
Auto-Adjust Display Label Position Markers 70

- B -

Bi-Directional Signals 35
Big Endian 94
Bit Order 94
Bit Slice 142
Bit Stream Packing 94
Blocking Construct
  Example 157
  Samples 54
  Sensitive Edges 40
  Wait Until Marker 74
bool 109
Boolean Conidtion For Delay 61
Boolean Equations 39
Break at Time Zero 129
BugHunter Project
  converting to TestBencher 151
Burst Mode Transactions 159
Buses
  adding to diagrams 42
byte 109
Byte Order 94

- C -

C++ 140
C++ compiler 21
Capture and Show Watched Signals 129
Class Libraries
  Overview 82
Class Library List 91
Class Methods
  adding to a class 86
  Class Level methods 87
  Diagram Level methods 87
  Project Level methods 87
Classes
  defining fields for 100
  Overview 82
Classes and Variables Dialog
  constrained random generation 92
  defining constraints for 86
  defining fields for 86
  Definition of Types and properties 107
  View Variables 82

Copyright © 2014, SynaptiCAD Sales, Inc.

Clear Log File Before Compile 129
Clock Domain
  Blocking Samples 54
  Overview 44
  Sensitive Edges 44
  Waveform Code Generation 44
Clocks
  adding to diagrams 42
Code Generarion
  Enable Sample, Delay, Marker 148
Code Generation Options Dialog
  Blocking Samples 54
Colors of Waveforms 35
Command File - Generating 135
Compile the Active Project 24
Component Instance Generation Properties menu 144
Component Model
  Class Methods 87
  default port mapping 139
  definition of sub-project 137
  instance of sub-project 139
  port mappings 142
  Sequencer Process 112
Concurrent Apply 112
Conditional State Transitions 56, 160
Constrain Input Data 121
Constrained Random Number Generation 92
Constraints
  defining for a class 86
Continuous Setups and Holds 67
Count Clock Edges 61
Create Component Instance 139
Critical Regions
  defining 78
Cycle Based Properties 146

- D -

Data Packing 170
  in Verilog 170
Debug Run 134
Default Clock 151
Default Delay After Clock Edge
  As Drawn 148
  Fixed 148
Define from Template button 86, 100
Delayed State Transitions 56
Delays
  code generation 61
  conditional 56, 61
  creating 61
  Delay Code Generation min, max, typical 148
  Enable Code Generation for all 148
  Overview 61
  specifying the order of 40
  triggered by samples 56
  Verbose for all 148
Design Flow 8
Diagram Properties Dialog
  cycle based properties 146
  default clocking domain 146
  including Library Files 146
Diagram Settings dialog
  Code Generation 148
  Default Delay After Clock Edge 148
  default settings 151
  Display Applied Inputs 148
  during project creation 151
  Enable Abort Code 148
  Instance Setting 148
  Master Transactor 148
  Project Simulation Properties 148
  Signal HDL Direction and Type 148
  Slave Transactor 148
  TimeOut 148
  Verbose Samples, Delays, Markers, Sensitive Edges 148
Diagram Variables 82
  in waveforms 36
Diagrams 16
  creating 16
  template diagram 16
  view generated source code 16
Direction 35
Direction of classes or variables 107
Direction Waveform Segment 35
Display Applied Inputs 148
Display Signal States Markers 70
Documentation Markers 80
Drawing Waveforms 33
Driven Flags 35
Driving Events 32
Dump Watched Signals 129

Copyright © 2014, SynaptiCAD Sales, Inc.

- E -

Edge Properties Dialog
Trigger Order 40
Edit Bus State Dialog
conditional states 36
Driven flag 35
Variables 36
Editor
keyboard shortcuts 179
Editor Commands 179
element 107
Enable Abort Code 148
Enable Code Generation
Delays 61
Setups and Holds 65
Enable Delay HDL Code Generation 148
Enable HDL Code Generation
for Samples 49
Samples 51
Enable Marker HDL Code Generation 148
Enable Reference Model
how-to 140
Enable Sample HDL Code Generation 148
Enable TestBuilder Integration 151
Enable Variable
Sample 59
End Diagram Markers 73
End Loop Marker 75
End Transaction 112
event 109
Events 32
Examples
Overview 154
Exit Loop When 75
Export as BugHunter Project 151
External Program Integration 181
External Program Settings 24
External Simulators 24
Extract Ports from MUT 14

- F -

Falling Edge Sensitive 40
Field Name 107
Fields
defining from file 100
File data initialize Array (Verilog) 102
File data initialize Array (VHDL) 104
File Input 107
File Output 107
File Structure Types 107
fixed_len_string 109
For Loop Markers 75
Full Expect 51
functions 77

- G -

Generate Test Bench 19
Generate Test Bench on Build Project 129
Generated Code 16
controlling 16
Generating Constrained Random Data 175
in TestBuilder 175
Golden Reference Model 140
Grab Top Level Singals 129

- H -

HDL Code Button 49
HDL Code Generation button
Delays 61
Setups and Holds 65
HDL Code Markers 77
Holds
continuous 67
creating 65
Enable Code Generation for all 148
outward arrows 65
Overview 61
properties 65
specifying the order of 40
Verbose for all 148

- I -

If-Then-Else Sample Code 51
Include Directories 129
Including Library Files 146
Initial value 107
Initialization Diagram 32
Initialize Array from file (Verilog) 102

Copyright © 2014, SynaptiCAD Sales, Inc.

Initialize Array from file (VHDL) 104
Inout Signals 35
Insert Diagram Calls 17
Insert into Equation 59
Instance Count for Pipelines 78
Instance Setting 148
int 109
Is Apply Subroutine Input
for Samples 49
Is Apply Subroutine Input control
Delays 61
Setups and Holds 65

- K -

Keyboard Shortcuts in Editor Windows 179

- L -

Language
details specific per 170
Libraries (VHDL) 141
Library Directories 129
Library Extensions 129
Library Files
class libraries 82, 91
including in diagrams 146
Little Endian 94
Loop Until Markers 75
Looping Markers 75
LSB 107

- M -

Make Parameters Watchable 129
Make TB button 19
Margin 65
Markers
absolute 70
attach to edge 70
attach to time 70
created 70
Documentation type 80
Enable Code Generation for all 148
End diagram type 73
HDL Code type 77
looping types 75
overview 70
Pause Simulation (Verilog) type 74
Pipeline type 78
relative 70
semaphore type 78
specifying the order of 40
Time Break type 80
Verbose 148
Wait Until Type 74
Master 112
Master Transactions 115
Master Transactor
Diagram Setting 148
Model Under Test (also see MUT) 12
ModelSim 21
MSB 107
Multiplier 51
MUT 12
adding to the project 12
extracting ports 14

- N -

NC Verilog 21
Network Packing 94, 107
New Diagram Defaults 148
New Project Wizard dialog 151

- O -

Outward Arrows 65

- P -

Packing 94, 107, 170, 171
in Verilog 170
in VHDL 171
Parameters
Delays 61
Holds 61
samples 49
Setups 61
specifying the order of 40
temporal expressions 61
Pause Simulation (Verilog) Markers 74
Pause Simulation (VHDL work around) 74
Pipeline

Copyright © 2014, SynaptiCAD Sales, Inc.

Pipeline
Instance Setting 148
Pipeline Boundary Markers 78
Point Samples
add to diagram 49
definition 49
Port Mappings
component instances 142
default for component definitions 142
Post Calls 117
VHDL 170
Post Semaphore Marker 78
PostRandomTransactionType 121
Project 12
adding files to 12
chaning template files 126
Chapter overview 137
Class Library List 91
Component Instance Generation Properties menu 144
Component Model 139
creating 151
Default Diagram Settings 151
directory 151
Enable TestBuilder Integration 151
Language 151
Project Generation Properties menu 144
Quick steps overview 8
saving 151
Project Generation Properties dialog
Add Timestamp to Each File 144
Enable Reference Model 144
Instance Name 144
Prefix Generated Files With 144
Source Indent Size 144
Transaction Recording 144
Verbose Transaction Logging 144
Project Generation Properties menu/dlg 144
Project Library
definitions of components 137
instances of components 139
Project Simulation Properties 148
Project Simulation Properties dialog
Auto Parse Project on Load 129
Break at Time Zero 129
Capture and Show Watched Signals 129
Clear Log File Before Compile 129
Dump Watched Signals 129
Generate Test Bench on Build Project 129
Grab Top Level Singals 129
Include Directories 129
Library Directories 129
Make Parameters Watchable 129
Project Variables 82, 170
accessing in Verilog 170

- Q -

queue 107
queuePath 171
Queues 82

- R -

Rand 92
Randc 92
Random Data 124
Random Transactions 121, 175
Randomization 107, 175
in TestBuilder 175
Randomize function 92
Reactive Test Bench
Overview 25
Step 1: Create Project and Extract Ports 25
Step 2: Draw Test Bench 28
Step 3: Generate the Reactive Test Bench 30
real 109
Reference Model 140
Relative Markers 70
Relative Samples
add to diagram 49
definition 49
Repeat Loops Markers 75
Rising Edge Sensitive 40
Run Simulation 24
RunRandomTransaction 175

- S -

Sample
Enable Code Generation 148
Verbose 148
Sample Properties Dialog
Enable HDL Code Generation 49
HDL Code Button 49

Copyright © 2014, SynaptiCAD Sales, Inc.

Sample Properties Dialog
Is Apply Subroutine Input 49
Samples
Actions 56
adding 49
Blocking 54
defined 49
Enable Variable 59
Full Expect 51
If Condition 51
Multiplier 51
point 49
relative 49
Sample Properties dialog 49
Sample Variable and Flag 58
self-testing code 51
specifying the order of 40
State Matches 51
Store Sampled Value as a Subroutine Output 58
Then/Else Action 51
triggering delayed state transitions 56
triggering other samples 56
scoreboarding 168
Select Packing Options dialog 94
Select Variable Dialog 59
Self-Testing Code
Samples 49
Semaphores
Markers 78
Sensitive Edges
Code Generation 44
Defining 40
Verbose 148
Sequence Recognition
ordering events 40
Sequencer Process 17
Overview 112
used to sequence transactions 17
Sequencing Transactions 115
Serial Data 161
SetApplyCallMode 120
SetTransactorWeightings 121, 175
Setups
continuous 67
creating 65
Enable Code Generation for all 148
outward arrows 65
Overview 61
properties 65
specifying the order of 40
Verbose for all 148
Signal HDL - Direction and Type 148
Signal Properties dialog
Sensitive Edges 40
Signals
adding to diagrams 42
advanced register 67
clocked 67
direction 42
Direction Inout 35
drawing 33
driving events on 32
properties 67
Signals and Ports
of Component Model 137
Signals and Ports dialog
menus to open 142
Port Mappings 142
signed_logic 109
Simulated Signals 39
Simulation 24, 128, 135
project properties 129
test bench 128
with third party simulators 135, 181
Simulation Button Bar 128
Simulation Files 170, 175
needed for TestBuilder 175
needed for Verilog 170
Simulation Mode
Auto Run 134
Debug Run 134
Simulator and Compiler Settings dialog 21
Size 107
Slave 112
Slave Transactions 115
Slave Transactor
Diagram Setting 148
Source Code 16
for Class Methods 87
generated for diagrams 16
Spacer
adding to diagrams 42
State Matches - Samples 51
State Transitions 160
conditional 160

Copyright © 2014, SynaptiCAD Sales, Inc.

State Transitions 160
waiting for 157
State Variables
in waveforms 36
Static 107
std_logic 109
std_logic_vector 109
std_ulogic 109
std_ulogic_vector 109
Store Sampled Value as a Subroutine Output 58
Structure Type 107
Sub-Project
Design Flow 137
instances of components 139
Supported Simulators 181
Sweep Test 160

- T -

tasks 77
tb_control VHDL variable 171
tb_LocalBfmPath VHDL variable 171
tbench 170
tbsyslog 170
Template Diagram 14, 16
Template Files 17
adding Apply calls 17
adding diagram Apply calls 112
changing for project 126
modifying 17
Sequencer Process 112
Temporal Expressions
expressing with Parameters 61
Test Bench
compiling 128
generating 19, 128
simulating 128
Test Reader 115
Test Vector Files
defining class fields with 100
format 100
TestBencher Project 8
design flow 8
TestBuilder
enabling integration 151
Third-Party Simulators 24, 135
time 109
Time 0 32
Time Break Markers 80
TimeOut 148
Timing Diagrams (see Diagrams) 16
Transaction
Timeout 148
Transaction Diagrams 17
overview 32
sequencing 17
Transaction Generator 121, 175
Transaction Manager 115, 171, 175
file format 115
in TestBuilder 175
in VHDL 171
run modes 115
Transactor Overview 32
transactorBfmPath 171
Triggering Order of Parameters and Markers 40

- U -

UART 161
unsigned_int 109
unsigned_logic 109
Use Clauses (VHDL) 141
User Defined Type 86

- V -

Variable
Definition of Types and properties 107
Variables 170
creating 82
Diagram Variables 36, 82
editing 82
exporting signal states to a file 59
Overview 82
Project - accessing in Verilog 170
Project Variables 82
random with constraints 92
State Variables 36
storing samlped values in 59
Verbose Samples, Delays, Markers, Sensitive Edges 148
Verbose Transaction Logging 144
Verilog 170
accessing project variables 170
VeriLogger Project

Copyright © 2014, SynaptiCAD Sales, Inc.

VeriLogger Project
converting to TestBencher 151
VHDL 171
Libraries 141
Post Calls 170
Use Clauses 141
View Source Code 16
generated for diagrams 16
View Variables 82

- W -

Wait Semaphore Marker 78
Wait Until Marker 74
Waiting For State Transitions 157
Waveforms 32
driving with variables 36
waveperl.log 19
Weightings Table 121, 175
While Loop Markers 75
Window Samples
add to diagram 49
definition 49

Copyright © 2014, SynaptiCAD Sales, Inc.