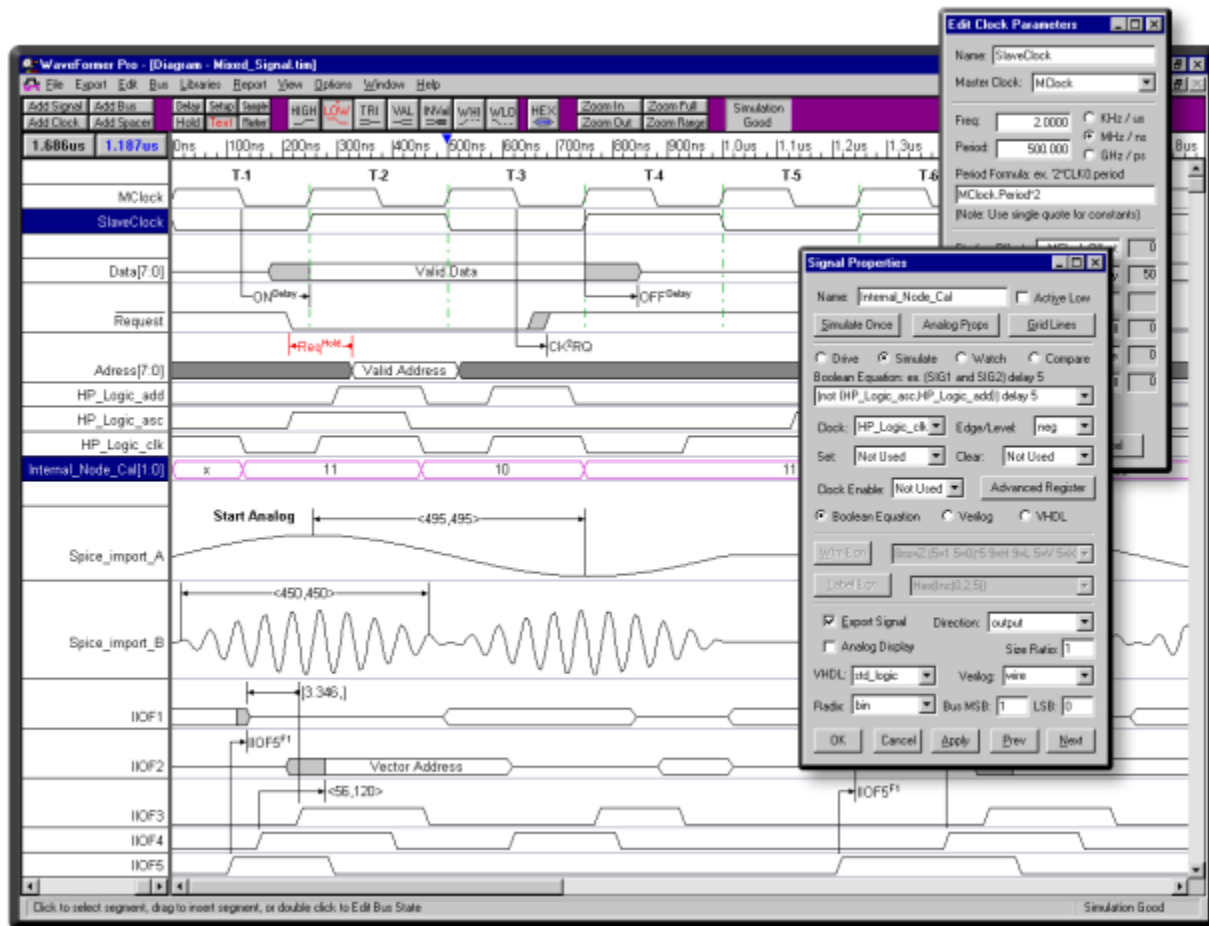


# SynaptiCAD Tutorials

SynaptiCAD 2007



# SynaptiCAD Tutorials

## Copyright SynaptiCAD 2007, version 12

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: March 2008 in (whereever you are located)

# SynaptiCAD Product Tutorials

**DataSheet Pro, WaveFormer Pro, Timing Diagrammer Pro  
BugHunter Pro, VeriLogger Extreme, VeriLogger Pro  
TestBencher Pro, GigaWave Viewer, Transaction Tracker**

---

*SynaptiCAD's tutorials demonstrate everything from how to draw basic timing diagrams to advanced VHDL and Verilog simulation techniques. The following chart describes the recommended tutorials for each of our products. If you are new to our product line, the best tutorial to start with is the Basic Drawing and Timing Analysis tutorial, because this demonstrates how to draw waveforms and the general design strategy for our timing diagram editors. Some of the features demonstrated require additional licenses for the particular software option. Please see our web site or contact our sales department for specific information on those features.*

# Table of Contents

Foreword	0
<b>Timing Diagram Editor 1: Basic Drawing And Timing Analysis</b>	<b>9</b>
(TD) 1.1 Timing Diagram Editor Choices.....	10
(TD) 1.2 Set the Base and Display Time Unit.....	11
(TD) 1.3. Add the Clock.....	12
(TD) 1.4 Add the Signals.....	14
(TD) 1.5 Drawing Signal Waveforms.....	15
(TD) 1.6 Editing Signal Waveforms.....	16
(TD) 1.7 Adjust Diagram to Match Figure.....	18
(TD) 1.8 Add the D Flip-Flop Propagation Delay.....	19
(TD) 1.9 Add the Inverter Propagation Delay.....	21
(TD) 1.10 Add the Setup for the Dinput to Clock.....	23
(TD) 1.11 Add a Free Parameter.....	24
(TD) 1.12 Drawing with Equations.....	26
(TD) 1.13 Drawing Virtual Busses.....	28
(TD) 1.14 Drawing Group Buses and Differential Signals.....	29
(TD) 1.15 Working with Drawing Environment.....	33
(TD) 1.16 Summary .....	33
<b>Timing Diagram Editor 2: Simulated Signals</b>	<b>35</b>
(TD) 2.1 Setup for Simulation.....	35
(TD) 2.2 Simulate a Boolean Equation.....	36
(TD) 2.3 Boolean Equations with Delays.....	38
(TD) 2.4 Register and Latch Signals.....	39
(TD) 2.5 Set and Clear Lines.....	41
(TD) 2.6 Multi-bit Equations.....	43
(TD) 2.7 Design a Multi-Bit Counter.....	45
(TD) 2.8 End Diagram Marker Stops Simulation.....	46
(TD) 2.9 Behavioral HDL Code.....	47
(TD) 2.10 Simulated Bus Signals.....	49
(TD) 2.11 Summary of Simulated Signals Tutorial.....	50
<b>Timing Diagram Editor 3: Display and Documentation</b>	<b>52</b>
(TD) 3.1 Controlling Parameter Display String.....	52
(TD) 3.2 Repeating Parameters.....	53
(TD) 3.3 Editing Waveform Edges From an Equation.....	54

(TD) 3.4 Drag and Drop Parameter End Points.....	54
(TD) 3.5 Adjusting the Vertical Placement of a Parameter .....	55
(TD) 3.6 Clock Jitter and Display.....	55
(TD) 3.7 Markers .....	56
(TD) 3.8 Edit Text Blocks.....	57
(TD) 3.9 Summary of Display and Documentation Tutorial.....	58

## **Timing Diagram Editor 4: Analog Signals 59**

(TD) 4.1 Viewing & Exporting Real Radix Signals as Analog Waveforms .....	59
(TD) 4.2 Creating Analog Signals using the Mouse.....	60
(TD) 4.3 Drawing a Step Signal.....	61
(TD) 4.4 Drawing Analog Ramps.....	62
(TD) 4.5 Generating Sine Waves.....	62
(TD) 4.5.a The SinStart(amplitude, period, duration) Function .....	63
(TD) 4.5.b The Sin(amplitude, period, duration) Function .....	63
(TD) 4.5.c The SinEnd(amplitude, period, duration) Function .....	63
(TD) 4.6 Generating Capacitor Charge and Discharge.....	64
6.1 The CapCharge(amplitude, RC, duration) Function .....	64
6.2 The CapDischarge(amplitude, RC, duration) Function .....	64
(TD) 4.7 Converting between Real and Multi-bit Signals.....	65

## **Timing Diagram Editor 5: Parameter Libraries 66**

(TD) 5.1 Adding Libraries to the Project's "Library Search List".....	66
(TD) 5.2 Setting Library Specifications.....	67
(TD) 5.3 Startup Library Configuration.....	68
(TD) 5.4 Referencing Parameters in Libraries.....	68
(TD) 5.5 Using Macros to Examine Tradeoffs Between Different Libraries .....	69

## **Timing Diagram Editor 6: Advanced Modeling and Simulation 71**

(TD) 6.1 Set up a New Timing Diagram.....	72
(TD) 6.2 Generate the Clock, Draw Waveforms, & Use Waveform Equations .....	72
(TD) 6.3 Modeling State Machines.....	74
(TD) 6.4 Checking for Simulation Errors.....	75
(TD) 6.5 Incremental Simulation.....	76
(TD) 6.6 Modeling Combinational Logic.....	77
(TD) 6.7 Entering Direct HDL Code for Simulated Signals.....	78
(TD) 6.8 Modeling n-bit Gates.....	78
(TD) 6.9 Incorporating Pre-written HDL Models into Waveformer Simuations .....	79
(TD) 6.10 Modeling the Incrementor and Latch Circuit.....	80

(TD) 6.11 Modeling Tri-State Gates.....	80
(TD) 6.12 Debugging External Verilog Models.....	81
(TD) 6.13 Verify the Histogram Circuit.....	81
(TD) 6.14 Controlling the Length of the Simulation.....	81
(TD) 6.15 Editing Verilog Source Files.....	82
(TD) 6.16 Simulating Your Model with Traditional Verilog Simulators .....	82
(TD) 6.17 Summary .....	82

## **Test Bench Generation 1: VHDL-Verilog Stimulus 83**

(TBench) 1.1 Getting Started.....	83
(TBench) 1.2 Default Mappings: Hex and Binary Translations.....	83
(TBench) 1.3 Generating Verilog Code.....	84
(TBench) 1.4 VHDL - Advanced Data Types.....	85
(TBench) 1.5. Exporting to VHDL.....	86

## **Test Bench Generation 2: Reactive Test Bench Option 87**

(TBench) 2.1 The Model Under Test.....	87
(TBench) 2.2 Create Signals.....	88
(TBench) 2.3 Draw Single Write (without waiting on TRDY).....	89
(TBench) 2.4 Export Diagram as an HDL Test Bench.....	90
(TBench) 2.5 Add Wait for TRDY Assertion.....	90
(TBench) 2.6 Draw Single Read.....	92
(TBench) 2.7 Add a Sample to Verify Data Read From MUT.....	93
(TBench) 2.8 Drive Data Using a "Test Vector Spreadsheet" File.....	93
(TBench) 2.9 Create For-Loop to Perform Multiple Writes and Reads .....	94
(TBench) 2.10 TestBench Pro Transactor - Add Address Argument .....	95
(TBench) 2.11 Alternatives.....	96

## **Test Bench Generation 3: TestBench Pro Basic Tutorial 97**

(TBench) 3.1 Run TestBench Pro.....	97
(TBench) 3.2.Create a Project.....	97
(TBench) 3.3 Add the SRAM model to the Project.....	100
(TBench) 3.4 Setup the Template Diagram.....	101
(TBench) 3.5 Create the Write Cycle Transaction Diagram.....	103
(TBench) 3.6 Create the Read Cycle Transaction Diagram.....	105
(TBench) 3.7 Add a Sample to Verify Data.....	106
(TBench) 3.8 Create the Initialize Transaction Diagram.....	108
(TBench) 3.9 Add Transaction Calls to the Sequencer Process.....	110

(TBench) 3.10 Setup the Simulator.....	113
(TBench) 3.11 Generate the Test Bench and Simulate.....	113
(TBench) 3.12 Examine Report Window Results.....	114
(TBench) 3.13 Examine the Stimulus and Results Diagram.....	115
(TBench) 3.14 TestBench Pro Basic Tutorial Summary.....	115
<b>Simulation 1: VeriLogger Basic Verilog Simulation</b>	<b>116</b>
(Sim) 1.1 Simulator Choices.....	116
(Sim) 1.2 Add Files to the Project.....	117
(Sim) 1.3 Build the Tree and Investigate the Project.....	119
(Sim) 1.4 Simulate the Project.....	121
(Sim) 1.5 Prepare for Graphical Test Bench Generation.....	123
(Sim) 1.6 Draw Test Bench in Debug Run Mode.....	124
(Sim) 1.7 Simulate in Auto Run Mode.....	127
(Sim) 1.8 Breakpoints, Stepping and Inspecting.....	128
(Sim) 1.9 Archiving Stimulus and Results.....	130
(Sim) 1.10 Saving the Project files.....	132
(Sim) 1.11 Summary of VeriLogger Basic Verilog Simulation.....	132
<b>Simulation 2: Using WaveFormer with ModelSim VHDL</b>	<b>134</b>
(Sim) 2.1 Compile SynaptiCAD Library Models.....	134
(Sim) 2.2 Create a project and extract the ports.....	136
(Sim) 2.3 Draw the test bench waveforms.....	137
(Sim) 2.4 Export Waveforms to VHDL.....	140
(Sim) 2.5 Simulate VHDL test bench using ModelSim.....	141
(Sim) 2.6 Compare simulation results against expected results.....	142
(Sim) 2.7 Summary of Using WaveFormer with ModelSim VHDL.....	145
<b>Waveform Comparison Tutorial</b>	<b>146</b>
1. Creating Individual Compare Signals.....	146
1.1 To Create a Compare Signal .....	146
1.2 Draw the Expected Waveform on the Compare Signal .....	147
1.3 Comparison Using Tolerance .....	147
2. Comparing Timing Diagrams.....	148
2.1 Import VCD Data and Save as a .btim .....	148
2.2 Open the Logic Analyzer Data File .....	149
2.3 Compare the Two Timing Diagrams .....	149
2.4 Editing all Compare Signal Properties .....	150
2.5 Methods of Examining the Differences Reported During a Comparison .....	151
3. Performing a Clocked Comparison.....	152
4. Comparing During Clock Cycle Windows.....	153
4.1 Add Offset Clock to Diagram .....	153
4.2 Change the Compare Signals to Use the Offset Clock .....	153

5. Masking Clock Segments During Comparison.....	154
5.1 Search and Replace Signal Names .....	154
5.2 Add the Compare Enable .....	155
5.3 Add the Simulated Masking Signal .....	155
5.4 Use Masking Signal to Mask Clock Segments .....	156
5.5 Setting up Don't Care Regions .....	156
6. Adjusting the Time Difference Between Two Diagrams.....	156
7. Summary .....	157
<b>Gigawave and WaveViewer Viewer Tutorial</b>	<b>159</b>
1. Converting a vcd file into a btim file.....	159
2. Importing a subset of the Waveforms.....	159
3. Creating a Filter File to selectively load signals.....	161
4. Show and Hide Signals in the display.....	162
5. Options: Gigawave, Waveform Comparison, Transaction Tracking .....	163
6. Waveviewer/GigaWave Viewer Tutorial Summary.....	164
<b>Transaction Tracker Tutorial</b>	<b>165</b>
1) Open the Example File.....	167
2) Match all occurrences of a simple pattern.....	167
3) Match Consecutive Occurrences with Concatenation Operator.....	168
4) Match with consecutive repetition Operator.....	168
5) Match with non-consecutive Repetition Operator.....	169
6) Bit-slices and the Boolean operators.....	169
7) Implication operator.....	169
8) Implication Next-Cycle operator.....	170
9) PSL Property .....	170
<b>Index</b>	<b>171</b>



## Timing Diagram Editor 1: Basic Drawing And Timing Analysis

This tutorial demonstrates the basic timing diagram editor features. It teaches you how to draw timing diagrams using delays, setups, clocks and part libraries and how to use timing diagrams to help detect timing errors in digital designs. It also covers the waveform editing features, measurement and quick access buttons.

You will draw the timing diagram for the circuit shown in Figure 1. This circuit divides the clock frequency in half. Both the flip-flop and the inverter have propagation times that delay the arrival of the Dinput signal. If the Dinput is delayed too long it will violate the data-to-clock setup time. This increases the risk of the flip-flop failing to clock in the data and may lead to the flip-flop entering a metastable state.

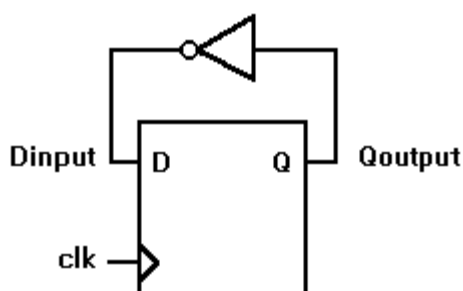


Figure 1: Tutorial circuit

Circuit Parameters:

clk	20MHz	(50ns period)
DFFtp	5-18ns	D flip-flop (74ALS74): Clock to Q propagation time
Dsetup	15ns minimum	D flip-flop (74ALS74): D to rising edge Clock setup time
INVtp	3-11ns	Inverter (74ALS04): propagation time

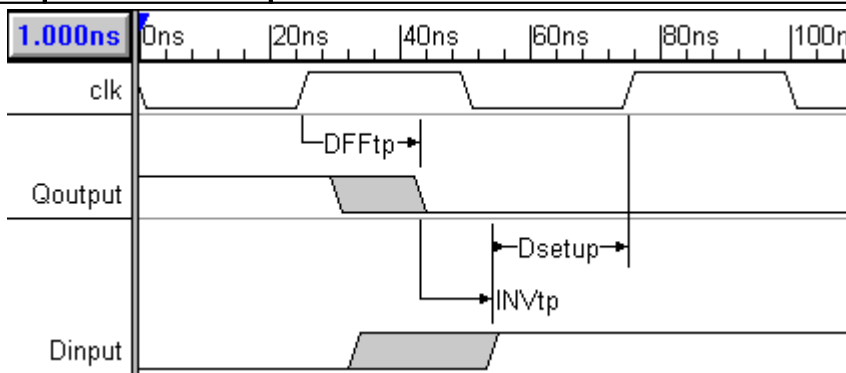


Figure 2: Completed timing diagram (captured with copy-to-clipboard function)

Figure 2 is the completed timing diagram. The first thing you may notice is the gray signal transitions caused by the min/max values of the component delays. The gray areas of the signal transitions are uncertainty regions, which indicate that the signal may transition any time during that period. This is a little disconcerting especially if you have been using a low-end simulator that

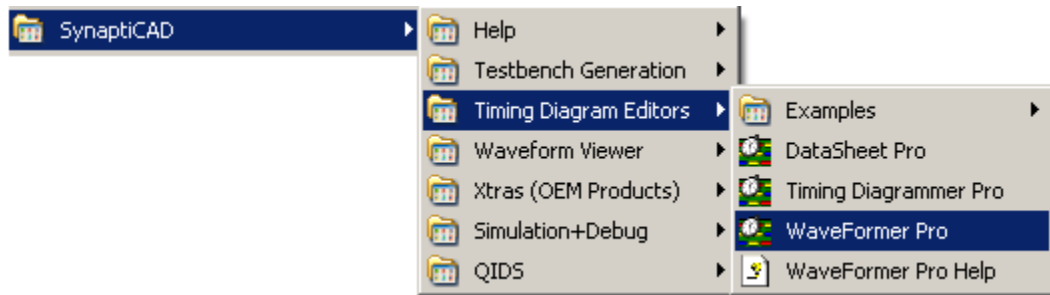
cannot compute both min and max at the same time. This representation shows the entire range of possible circuit performance. With a SynaptiCAD Timing Diagram Editor, there won't be any surprises during production when you get components at extreme ends of their tolerance range.

## (TD) 1.1 Timing Diagram Editor Choices

SynaptiCAD has three levels of timing diagram editors. The most basic is Timing Diagrammer Pro, which allows drawing and basic timing analysis using delays, setups and holds. The middle level is WaveFormer Pro, which has a built in simulation engine that allows signals to be described using Boolean and registered logic equation. WaveFormer is also a universal waveform translator and can take waveforms from one format and convert it to a different format. And the highest level is DataSheet Pro which supports multiple timing diagram display, object linking and embedding, and more image file formats. Each editor has all of the features of the products that are below it. This tutorial covers basic features which are supported in all three editors.

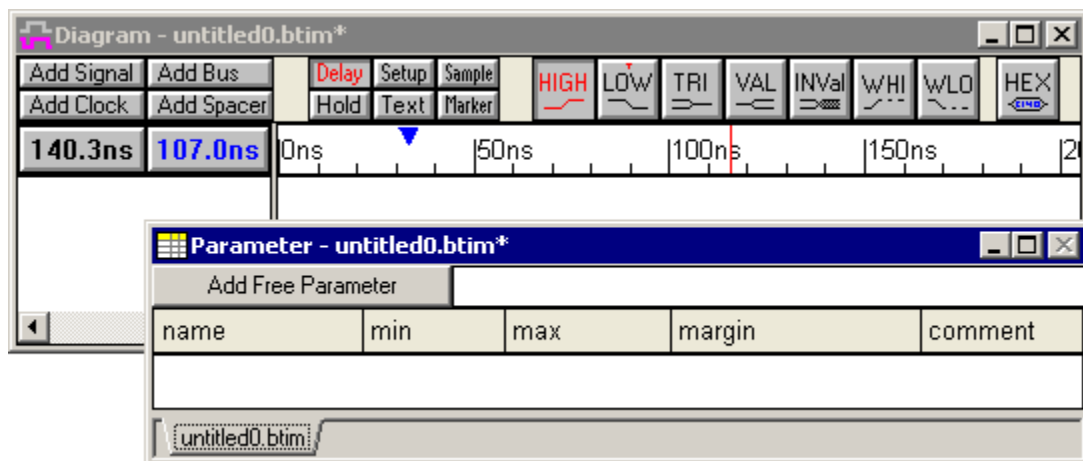
### *Run a Timing Diagram Editor:*

- Run one of the timing diagram editors from the Start Menu.



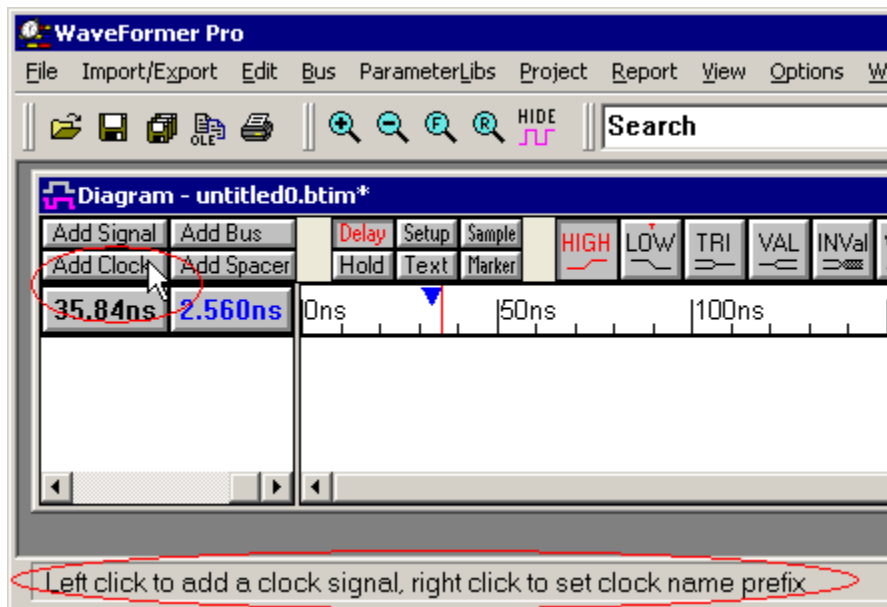
### *Open a New Timing Diagram File:*

- Select the **File > New Timing Diagram** menu to open a diagram editing window and a matching parameter window.



### *Investigate the Status Bar:*

- Move the cursor over the buttons on the diagram editor window, and then look at the bottom left corner of the big window to see the status bar.



- As you perform this tutorial, keep an eye on the status bar. It will give you hints on the types of functions that the mouse can perform at that particular place. The status bar also works when the mouse is inside the drawing window.

## (TD) 1.2 Set the Base and Display Time Unit

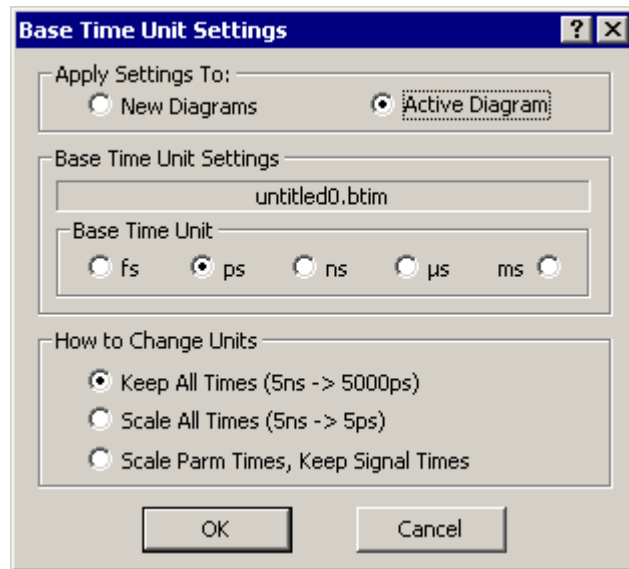
The base time unit controls the accuracy of the of time calculations. It is the smallest representable amount of time, and all time values are internally stored in terms of the base time unit. The display time unit controls the units for entering and displaying the results. At the beginning of a new design you should check these settings to make sure they are valid for the time ranges that you are working in.

### **Set the Base Time Unit:**

The base time unit for your project should be set at least one unit below the units you are working in for best rounding performance during division operations (clock frequencies are inverted and stored internally as clock periods). The circuit that we are modeling in this tutorial has gate propagation times in the range of 3 to 18 nanoseconds and a clock with a period of 20ns. Therefore we will set the base time units to picoseconds.

- Select the **Options > Base Time Unit** menu, to open the *Base Time Unit* dialog.

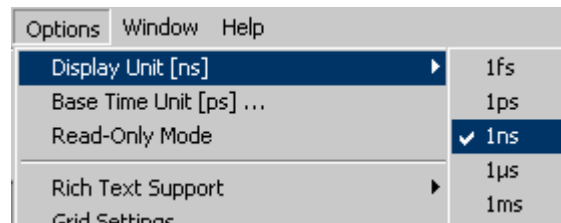
- Check the **Active Diagram** box so that the dialog operates on your new timing diagram. Notice that the dialog displays the name of the diagram so that you can tell which diagram is the active one in a multi-diagram display.
- Check the **ps** button to set the base time unit to picoseconds.
- The remaining options control how any existing parameters or signals are changed when the base time unit is changed and have no effect on an empty timing diagram.
- Press the **OK** button to close the dialog.



### Set the Display Time Unit:

Set the display time unit to the units you most commonly use in the design. In this tutorial, all of the times are listed in nanoseconds so that will be the easiest time setting to enter the values.

- Select the **Options > Display Unit** menu option. This will display a submenu of display time units. The checked time is the current display time unit (Default is ns = nanoseconds).
- Click on **ns** to make nanoseconds the display time unit.

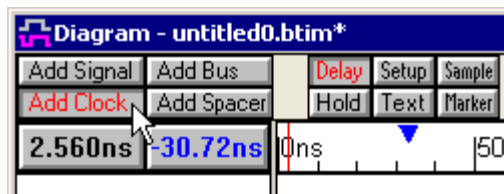


## (TD) 1.3. Add the Clock

First we will create a clock. Clocks draw themselves based on their parameters, so you will not be able to drag and drop clock edges or make a delay end on a clock edge. For this tutorial, the clock is named **clk**, has a period of **50ns** (20MHz), and starts with a low segment.

### Define the Clock Parameters using a dialog:

- Press the **Add Clock** button to open the *Edit Clock Parameters* dialog.



- In the **Name** box, type **clk** to set the clock name.



- In the **Period** box enter **50** and make sure that the **MHz/ns** button is selected. When you tab out of this box, the **Freq** box will change to 20 to match the new period value.
- Notice that the period can be also be defined by a **period formula** or in terms of a **reference clock**.
- Check the **Invert (Starts Low)** box at the bottom of the dialog. Clocks are normally displayed high at time zero, so "invert" makes the clock start low at time zero.
- Notice that the clock can have an offset starting time from time zero. The duty cycle can be set to any percentage value. And edge jitter is uncertainty around the occurrence of the clock edge.
- Notice that buffer delays represent uncertainty after the clock edge (used to model uncertainty from clock tree buffers), and delay correlation determines how closely delays are related to each other.
- For more information on correlation and the different types of delays, and clock grids read Chapter 2: Clocks in the on-line help
- Click the **OK** button to close the dialog. Make sure that the clock looks like the following image.

Reference Clk: None

Freq: 20. KHz / us  
MHz / ns  
GHz / ps

Period: 50

Period Formula: ex. 2\*CLK0.period

50

☒ Invert (Starts Low) Correlation

Starting Offset: 0

Duty Cycle %: 50

Rise Jitter (range): 0

Fall Jitter (range): 0

Buffer Delay

Min L to H: 0

Max L to H: 0

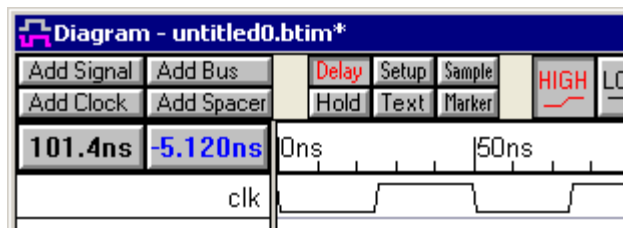
Min H to L: 0

Max H to L: 0

Rising Delay Correlation: 100 %

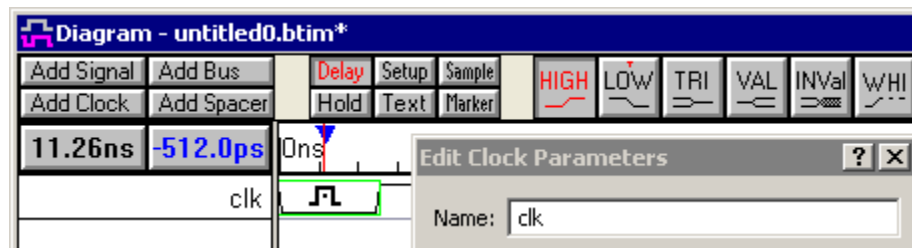
Falling Delay Correlation: 100 %

Rise to Fall Correlation: 100 %



### Reopen the Edit Clock Parameters dialog:

- Double left click on the clock waveform to reopen the *Edit clock parameters* dialog. Note, if you click too close to a clock edge it opens an edge dialog instead of the parameters dialog.



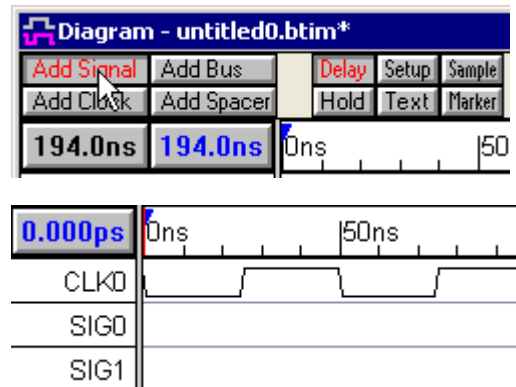
- Click the **Ok** button to close the dialog.

## (TD) 1.4 Add the Signals

Next, add two signals and name them "Qoutput" and "Dinput".

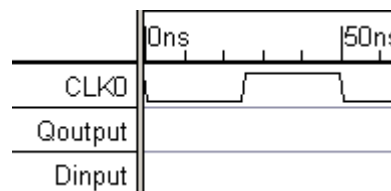
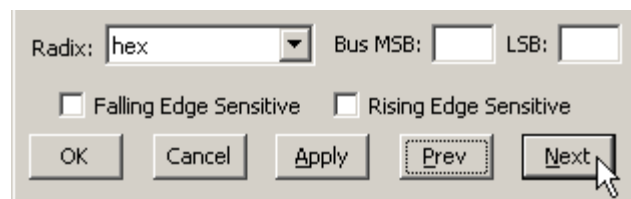
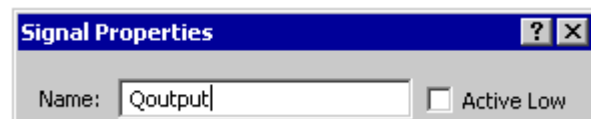
*Use the Add Signal button to create new signals:*

- Press the **Add Signal** button two times to add two signals to the diagram window.
- The signals will have default names such as SIG0 and SIG1.



*Double Click to rename the signals:*

- Double click** on the **SIG0** signal name to open the *Signal Properties* dialog.
- Enter **Qoutput** into the **Name** edit box. (DO NOT CLOSE THE DIALOG)
- Click the **Next** button or **ALT-N** to move to the next signal on the list. Notice that SIG1 is now displayed in the Name edit box.
- Enter **Dinput** into the **Name:** edit box and press the **OK** button to close the dialog.



**Tip:** The *Signal Properties* dialog is a modeless dialog - you can keep the dialog open while working with other drawing features. The Boolean Equation and Simulation features of the *Signal Properties* dialog are covered in the *Interactive HDL Simulation* tutorial. The *Signal Properties*

dialog is modeless, so you can leave the dialog open while you perform actions on the timing diagram.

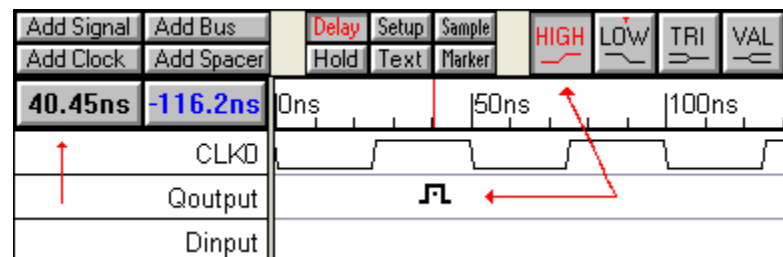
## (TD) 1.5 Drawing Signal Waveforms

Next, we will draw some random waveforms to become familiar with the drawing environment. 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 and forth between two states, and the next state is indicated by the little red T on top.

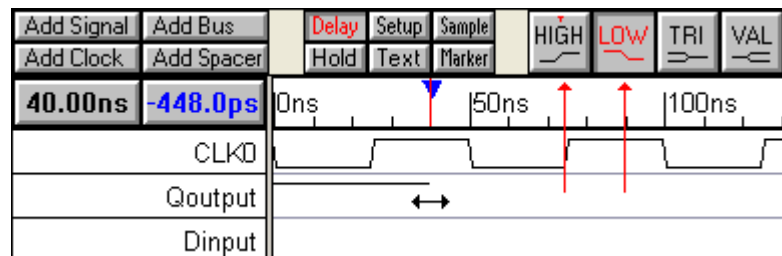


### Draw and watch the State Buttons:

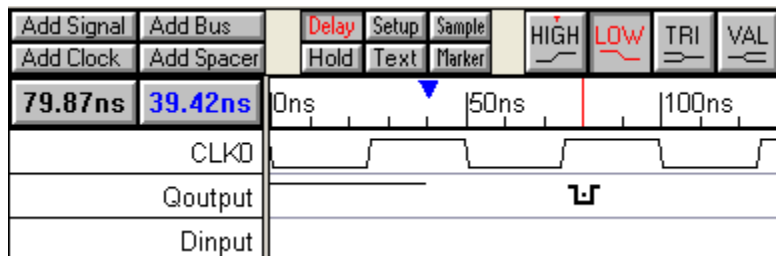
- Move the mouse cursor to about **40ns** and on the same level as **Qoutput**. Notice that the cursor has the same shape as the selected State Button.



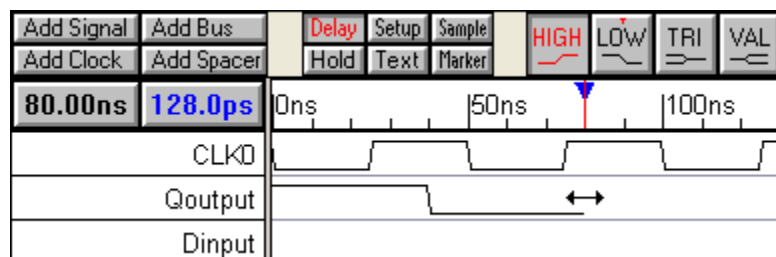
- Left Click** to draw a high waveform segment from 0ns to the cursor. Notice that the State Button toggled to low, and the toggle T moved to the High button.



- Move the cursor to about **80ns** on the same signal. Notice that the cursor looks like a low signal to match the active state button.

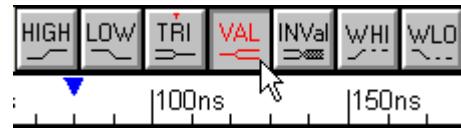


- Left click** to draw a LOW segment. It is drawn from the end of the HIGH signal to the location of the cursor

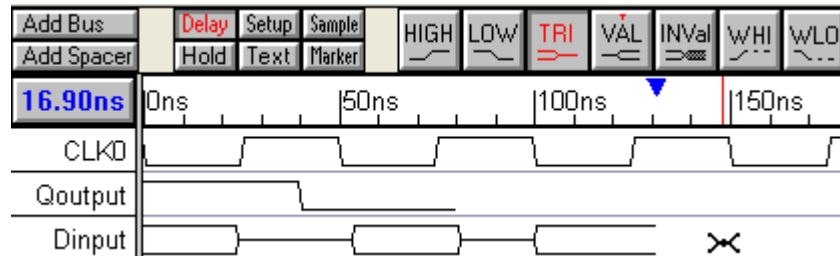


### Draw with other state buttons:

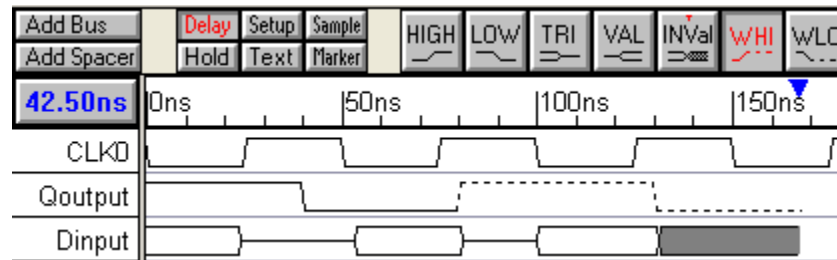
- Left click first on the **Tri-state** button then on the **Valid** button, so that the Valid button is red and the tri-state button has the red toggle T on it.



- Draw some valid and tri-state waveforms, while watching the cursor shape and the state buttons.



- Draw more segments, using all the states except the HEX button. The HEX state button is used in defining multi-bit signals and signals which have a user defined VHDL type. This button is covered in later tutorials. For now, experiment with the graphical states.



## (TD) 1.6 Editing Signal Waveforms

This section covers the main editing techniques used to modify existing signals (Note: these techniques will not work on clocks, because they draw themselves). The most commonly used technique is the dragging of signal transitions to adjust their location. Most of the other techniques all act on signal segments, the waveforms between any two consecutive signal transitions. The segment waveform can be changed, deleted, or a new segment can be inserted within another segment. Use each of the following techniques:

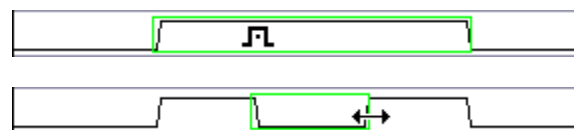
### 1) Drag-and-Drop Signal Transitions:

- Left click and hold down the mouse button on a signal transition and drag it to the desired location.
- To move transitions on different signals simultaneously, first select multiple transitions by holding the **<CTRL>** while clicking on them. Then drag the transition to desired location.



### 2) Click-and-Drag to insert a segment into a waveform or select to delete:

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

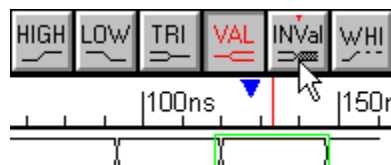




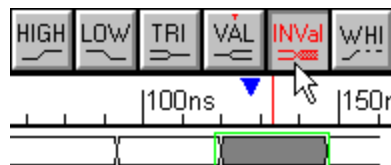
- **Delete a segment:** Select a segment (see above) and then press the **delete** key on the keyboard.

### 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).

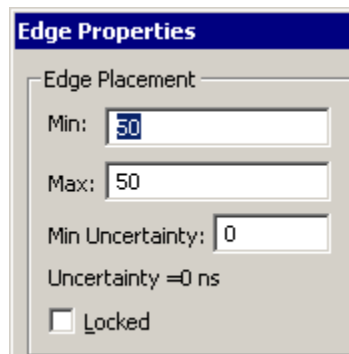


- Click on a state button to apply that graphical state to the segment. If you change a segment to the same state as an adjacent section, the transition will turn red to preserve the edge data. This transition can be deleted if necessary.



### 4) Find the exact edge time and see how to lock an edge

- Double-click on an edge of the signal transition to open the *Edge Properties* dialog.
- To move an edge, enter a new **min** or **max** time. An edge only has one time until uncertainty is added either by using a delay parameter or the **min uncertainty** box in this dialog.
- To lock an edge so that it cannot be moved, check the **Locked** checkbox. If a delay ends on a locked edge it will turn red if it cannot force the edge to the proper time.

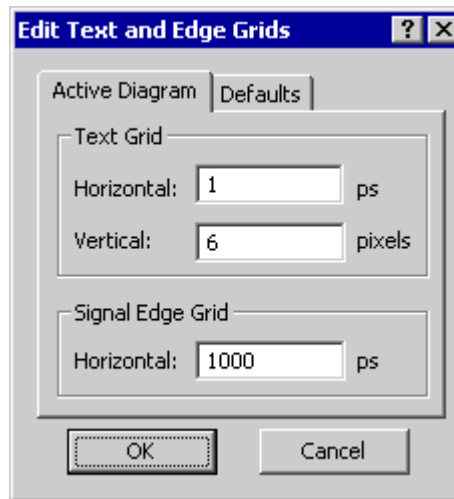


- Note: All edges on a signal can be locked by selecting the signal name, and then choosing the **Edit > (Un)Lock Edges of Selected Signals** from the main menu.
- Make sure to unlock any signals or edges you locked in this tutorial, or else your delay in the next section may not be able to force its ending edge.

### 5) Adjusting the drawing Grid

Drawn signal transitions are automatically aligned to the closest grid time. The grid does not affect the placement of edges that are moved by delays or formulas. By default the grid is set to the display time unit, because this generates nice VHDL and Verilog stimulus generation files with whole number times (like 2ns instead of 2.465ns). However, it is sometimes convenient to set the grid to a multiple of the clock frequency to make all new signal edges line up with the clock edges.

- Select the **Options > Grid Settings** menu item to open the *Edit Text and Edge Grids* dialog.
- You do not have to make any changes to this dialog. Just notice that you are able to control the **Signal Edge Grid**.
- Also notice that text objects have a different grid.



#### 6) Adding virtual state Information to a segment

- For Signals, double-click on the middle of a segment to open the *Edit Bus State* dialog, and then type in a new value into the **Virtual** edit box.
- For Clocks, press the **Hex** button and then double-click on the middle of the segment to open the *Edit Bus State* dialog. If the Hex button is not pressed, the double-click will open a different dialog to allow editing of the clock.

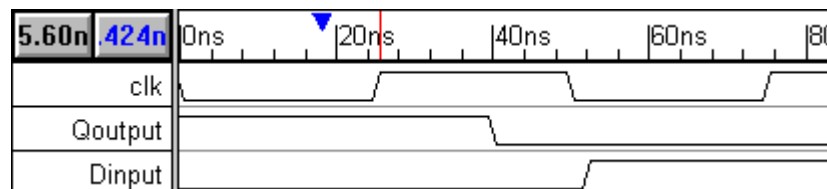


### (TD) 1.7 Adjust Diagram to Match Figure

In the next few sections we will be adding the delays and setups to the timing diagram. It is best to start with the waveform edges in the approximate position that they should be in when the timing diagram is finished.

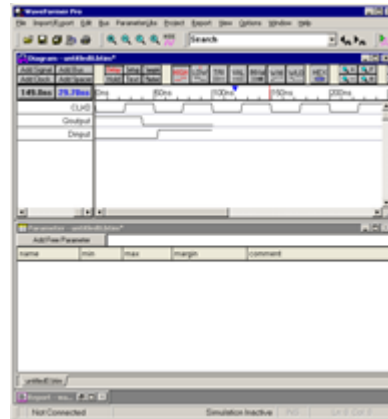
#### Adjust the Waveforms:

- Use the editing techniques in the previous section so that the waveforms have roughly the same transitions as the signals in the figure below.



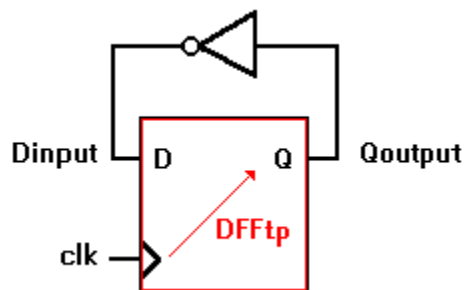
### Minimize the Report Window and tile the Parameter and Diagram Windows:

- Minimize the *Report* window because it is not used in this tutorial.
- Select the **Window > Tile Horizontal** menu to tile the *Parameter* and *Diagram* windows so that you will be able to see the interaction between the two windows.



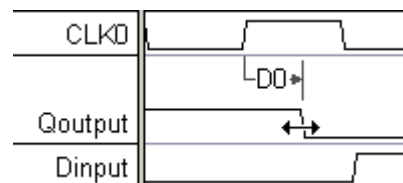
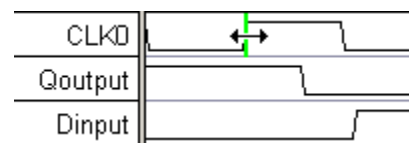
## (TD) 1.8 Add the D Flip-Flop Propagation Delay

Add the delay that represents the propagation time from the positive edge of the clock to the Qoutput of the D flip-flop. First we will add a blank delay between the edges and then we will edit it so that it is named "DFFtp" and has a propagation delay of 5-18ns.



### Add a blank graphical delay:

- Press the **Delay** button so that right clicks will add delays.
- Left click on the first rising edge of the clock to select it. This edge will be the first or driving edge of the delay.
- Right-click on the first falling edge of the **Qoutput** signal, to add the delay between the two edges. Since the delay is pointing to this edge, this will be the edge that moves in response to formula values entered into the delay
- Notice that D0 was also added to the *Parameter* window.



Parameter - untitled0.btim				
Add Free Parameter				
name	min	max	margin	comment
D0			na (delay)	

**Watch the delay as you change the min value:**

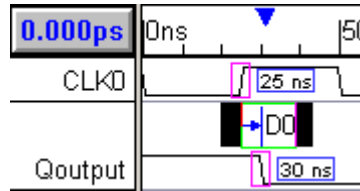
When delays are added, they are blank and do not enforce any timing restraints. Notice that the delay is drawn with gray colored lines: this indicates that the delay is not forcing either the min or max edge of the **Qoutput** signal. Now edit the delay's parameters.

- **Double-click** on **D0** in either the Diagram or Parameter window to open the *Delay Properties* dialog. For simplicity, we will refer to this dialog as **Parameter Properties**, even though the name at the top may say *Delay Properties* or *Setup Properties* depending on the type of parameter being edited.
- Adjust the position of the *Parameter Properties* dialog so that you can see the parameter in the *Diagram* window and at least part of the parameter in the *Parameter* window.
- Type **5** into the **min** edit box and press the **TAB** key to move to the **max** edit box (leave max blank for now). This enters 5 display time units (5ns for this timing diagram).

Delay Properties	
Name:	D0
Min:	5
Max:	
Comment:	

Several things happened when you pressed the TAB key:

- The falling edge of **Qoutput** was moved so that it is 5ns from the clock edge. If you hover over the delay with the mouse, blue boxes with the edge times will appear so that you can check the edge times.

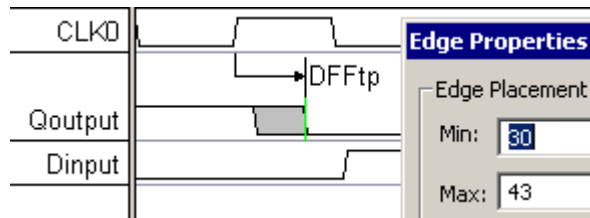
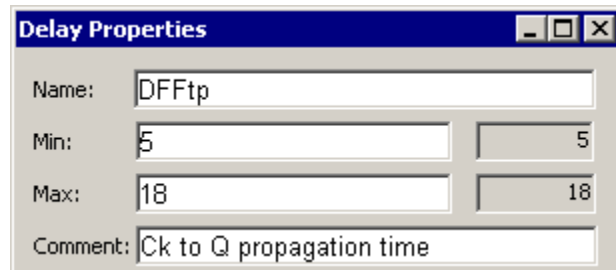


- Also note that the delay changed from a gray color to a **blue** color. Delays are color-coded to indicate which delays are forcing the min and max edges of a transition. This type of critical path display is necessary in diagrams where multiple delays drive a single signal transition. The colors are: Gray = none, Blue = Min only, Green = Max only, Black = both min and max. After this tutorial you may want to experiment with the **multdely.btim** file to see the effects of multiple delays on a single transition and critical path color coding.
- Finally, the parameter information also was updated in the *Parameter Window*.

name	min	max
D0	5	

**Edit the rest of the delay:**

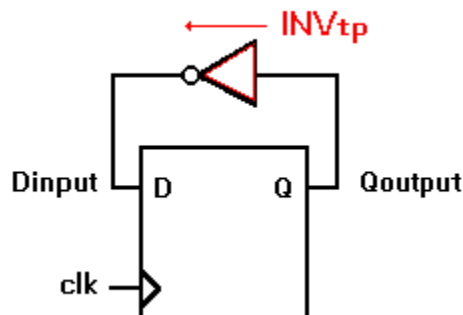
- Type **DFFtp** in the **Name** box.
- Type **18** in the **Max** box.
- In the **Comment** box, enter **Ck to Q propagation time**.
- Close the dialog when you are done.
- Notice that the DFFtp delay is black which indicates that it is forcing both edges of **Qoutput**.
- Also notice the falling edge of Qoutput now has a gray uncertainty region. Double click on the edge to verify that the edges of the region are **5ns** and **18ns** from the clock edge (13ns of uncertainty).



**Tip:** The *Parameter Properties* dialog is **modeless** (other operations can be performed while the dialog is open) and **interactive** (any changes in the dialog fields are reflected in the diagram after you move out of that field). When the *Parameter Properties* dialog is open you can edit a different parameter by double-clicking in the *Diagram* or *Parameter* window on the parameter you want to change. If you double-click in the *Diagram* window, that instance of the parameter will be edited (the **Change All Instances** checkbox will NOT be checked). If you double click in the *Parameter* window, ALL instances of the parameter will be edited (the **Change All Instances** checkbox will be checked).

**(TD) 1.9 Add the Inverter Propagation Delay**

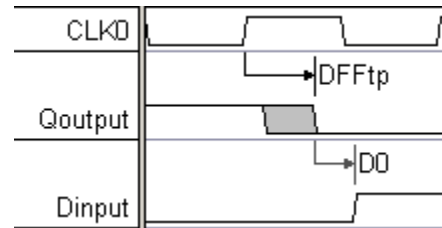
Next add the delay that represents the propagation time of the inverter from its input Q to its output D. Since this delay is the second in a chain starting with DFFtp, its uncertainty region will be larger than just the uncertainty caused by the inverter.

**Add the Inverter Delay:**

- Make sure the **Delay** button is red so that right clicks will add delays.



- Left click on the first falling edge of the **Qoutput** signal to select it (the same edge that ends the "DFFtp" delay).
- Right-click on on the first rising edge of the **Dinput** signal to add a blank delay.



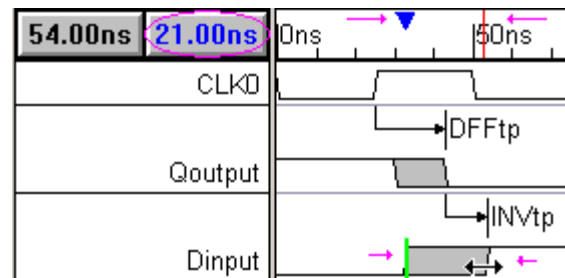
- Double-click on the new delay to open the *Parameter Properties* dialog and enter the following values: Name is **INVtp**, propagation delay of **3 to 11 ns**, and a comment of **Inverter (Q to D) delay**.
- Click on the **OK** button to close the dialog.

Delay Properties			
Name:	INVtp		
Min:	3		3
Max:	11		11
Comment:	Inverter (Q to D) delay		

**Verify that the Uncertainty region is correctly calculated:**

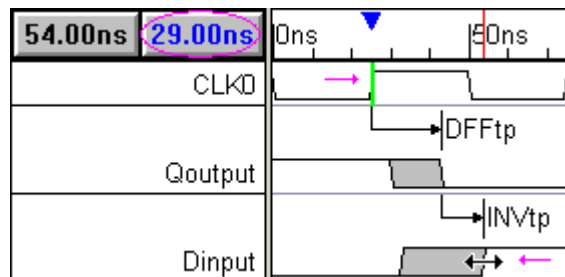
Notice the uncertainty region for the *Dinput* transition is much larger than the 3-11 ns that you entered in the last step. This is because the DFFtp uncertainty adds to the INVtp uncertainty.

- Click on the first rising edge of **Dinput** (to select it). This also moves the blue delta mark on the time line.
- Move the mouse cursor over the second edge of the uncertainty region. As you move the red line on the time line tracks your progress, and the Blue delta readout shows the exact distance from the blue delta mark.



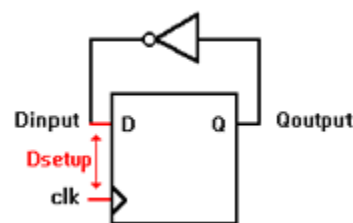
- Here the readout shows that the uncertainty region lasts for 21ns (13ns from DFFtp + 8ns from INVtp = 21ns).

- Next, click on the first edge of *clk* and measure to the end of the uncertainty region of *Dinput*. If both the inverter and the D flip-flop are slow, *Dinput* may not transition until 29ns after the clock edge.



## (TD) 1.10 Add the Setup for the Dinput to Clock

One of the most important features of a timing diagram editor is that setup and hold parameters can monitor pairs of signals transitions to make sure that they do not violate the timing constraints of the circuit. In this design, if Dinput changes too close to a clock edge then there is a risk that the flip-flop will go into a meta-stable state. We will use a setup parameter to make sure the Dinput does not violate the setup time for the clock.

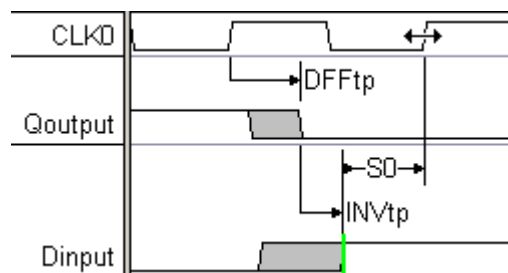


### Add a Setup parameter:

- Press the **Setup** button so that right clicks will add setups.



- Left click on one of the rising edges of the **Dinput** signal to select it.
- Right click on the second rising edge of the **clock** to add a blank setup between the selected edge and this one.
- Notice that the arrows of the setup are pointing to the control signal. This means that you added the setup correctly.



- Double-click on the new setup to open the *Parameter Properties* dialog and enter the following values: Name is **Dsetup**, min time is **15**, and the comment is **Check for metastable condition**.
- Press the **OK** button to close the dialog.

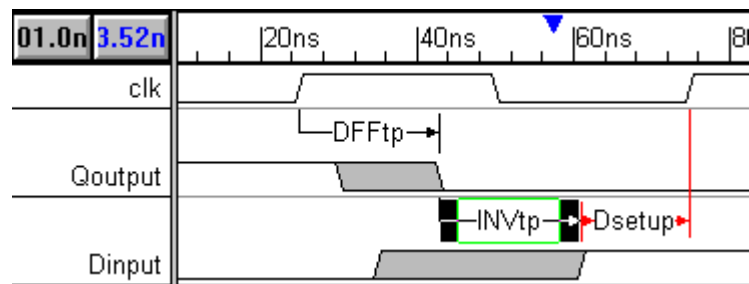
Notice that the margin column in the *Parameter* window says that there is a **6ns** safety region before the setup is violated. Verify this by clicking on the maximum edge of the **Dinput** signal (to place the blue delta mark on the time line), then placing the cursor on top of the second rising edge of the clock. The blue time readout should say 21ns (measured time 21ns - setup time 15ns = 6ns margin).

name	min	max	margin	comment
DFFtp	5	18	na (delay)	Ck to Q propaga
INVtp	3	11	na (delay)	Inverter (Q to D)
Dsetup	15		6	Check for metas

### Cause the Setup to be violated:

Next, we will demonstrate what happens when a setup is violated by increasing the inverter's delay.

- Double-click on **INVtp** to open the *Parameter Properties* dialog and change the **max** time to **18** ns. Then press the **Apply** button to apply the change.



- Notice that the setup has turned red in the *Diagram* window and that the **Margin** value of the *Parameter* window has also turned red.
- Change the inverter delay back to **11ns** and click **OK** to close the dialog.

## (TD) 1.11 Add a Free Parameter

So far we have always directly edited a parameter's values. This is inefficient and error prone if the circuit is large. It would be better to define one variable that held the value and make everything that needed that value reference this variable. Then if the value needs to be changed, you only have to edit one variable.

Free parameters act as variables that can be referenced by other parameters. They are called "free" because these parameters are not attached to any signal transitions in the *Diagram* window. Let's add a free parameter to hold the propagation times for the inverter.

### Add the free parameter:

- In the *Parameter* window, press the **Add Free Parameter** button to create a blank free parameter.

Parameter - untitled0.btim\*

Add Free Parameter

name	min	max
DFFtp	5	18
INVtp	3	11
Dsetup	15	
F0		

- Double click on the free parameter to open the *Parameter Properties* dialog box and enter the following: Name is **tpFreeInv**, min time is **3ns**, max time is **11ns**, and comment is **74ALS04 inverter delay**. (Leave the dialog open).

Free Parameter Properties

Name:

Min:

Max:

Comment:



- Make the dialog point to the **INVtp** delay, either by pressing the **Previous** button, or by double clicking on **INVtp**.
- Type **tpFreeInv** into the min and max cells of **INVtp** and notice that the calculated values show the actual times. Any changes to the timing values of the free parameter will now affect this delay.
- Notice that the row for **tpFreeInv** turned white to indicate that it is being referenced by another parameter.
- Select the max value of **INVtp** in the parameter window and notice that the formula is displayed in the box above.

**Delay Properties**

Name: INVtp

Min: tpFreeInv 3

Max: tpFreeInv 11

Comment: Inverter (Q to D) delay

☐ Hide Row ☒ Change all instances

☐ Is Apply Subroutine Input

Parameter - untitled0.btim\*

Add Free Parameter tpFreeInv

name	min	max
DFFtp	5	18
INVtp	3	11
Dsetup	15	
tpFreeInv	3	11

- You can make the parameter window display formulas in the table part by choosing the **Options > Parameter Window Preferences > Display min/max formula**.

Parameter Window Preferences

☒ Display min/max value

☐ Display min/max formula

Diagram Simulation Preferences...

### Using Formulas in the Parameter time boxes:

Parameters can contain mathematical formulas as well as numeric time values. Common operations include multiplication(\*), division(/), addition(+), and subtraction(-). For example, the inverter in this circuit could represent 3 cascaded inverters used to generate a minimum delay of 9ns. To represent this in your timing diagram:

- Enter **3 \* tpFreeInv** into **INVtp**'s **min** edit box. Then tab to a new box and see that the equation correctly calculated **9ns**.

**Delay Properties**

Name: INVtp

Min: 3 \* tpFreeInv 9

Max: tpFreeInv 11

Comment: Inverter (Q to D) delay

- Free parameter names can also be used with an attributed parameter name such as **tpFreeInv.min** and **tpFreeInv.max**. This gives you the flexibility to specify formulas any way you need. If no attribute is added then a min or max is assumed depending on whether the formula is in the min or max column.

### Create Libraries of Free Parameters:

Free parameters can be saved to special library files which can later be merged into other projects. You can also reference free parameters without including them into your project file by placing libraries in your library search path (**Libraries > Library Preferences** menu option). For more information on free parameters and libraries read the on-line help *Chapter 10: Libraries* or perform

the Parameter Libraries Tutorial.

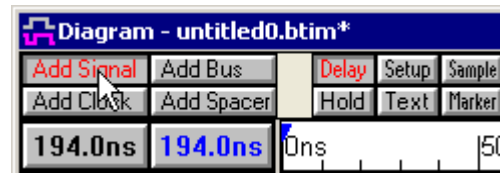
## (TD) 1.12 Drawing with Equations

We have finished with the timing analysis section of this tutorial, and next we would like to take the time to show you a few more drawing techniques that will help you create and manage complex timing diagrams. One such technique is to use equations to draw and label waveforms. Waveform and label equations provide a quick way to generate signals that have a known pattern that is more complicated than a periodic clock. WaveFormer (and higher editors) also support simulated signals based on Boolean Equations which are covered in the next tutorial Simulated Signals.

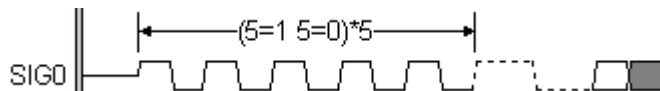
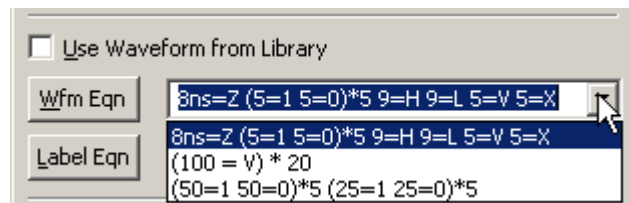
### Use the Waveform Eqn Feature:

The waveform equation box in the *Signal Properties* dialog accepts a list of time/value pairs, and the default equation has all of the syntax and states that are supported by this box. Each time you press the button more waveforms will be added on to the end of the signal.

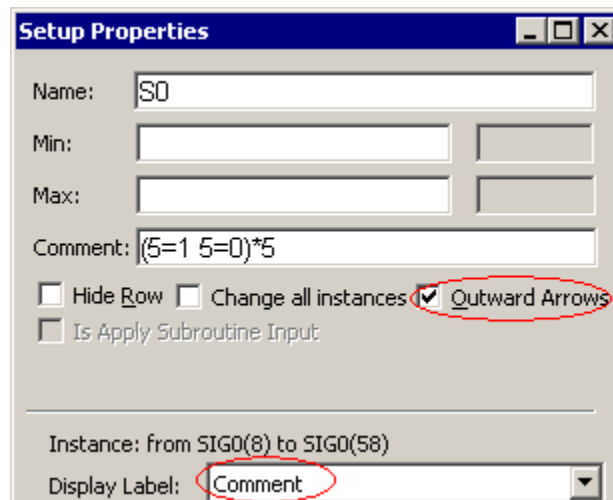
- Press the **Add Signal** button two times to add two new signals to the diagram window.
- Double click on **SIG0** to open the *Signal Properties* dialog.



- Press the **Wfm Eqn** button to apply the default equation to SIG0.
- Look at the waveform and try to match it to the different parts of the equation.
- The first pair, **8ns=Z**, causes an 8ns long tri-state segment to be drawn.



- The next terms, **5=1 5=0**, draws a 5 ns long high segment followed by a 5ns low segment, where the **ns** is implied by the display time unit. Enclosing it in **(...)\*5** causes the sequence to be repeated 5 times.
- We annotated the last sequence using a setup parameter and changing the display label from name to comment.
- The values H and L draw weak high and low waveforms, and V and X draw valid and invalid sections.

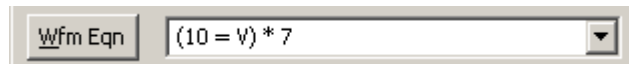


### Use the Label Eqn Feature:

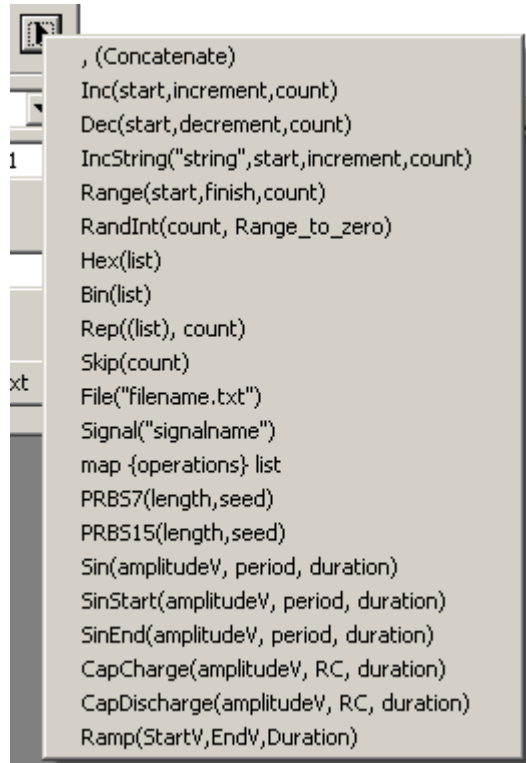
Label equations are used to automatically insert data on waveform segments. All of the equations are listed in the label fly-out box. Here we will draw and label a counter signal that first counts up

and then counts down.

- Double click on **SIG1** to open the *Signals Properties* dialog, then use a waveform equation to draw seven valid segments that are 10ns long.



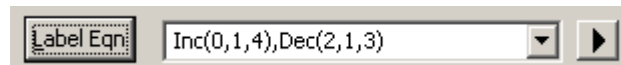
- Open the fly-out to the right of the **Label Eqn** button and take a look at the list of available functions. Choosing any function puts it at the end of the current label equation, then you can edit the parameters of the function call.
- Functions can be concatenated together by separating them with a comma.
- The Analog Label equations are covered in the Tutorial called Analog Signals.



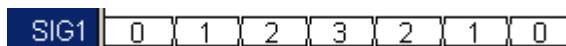
- Use the Label equation quick-fill box and choose **Inc** then **Concatenate** and then **Dec** to add those equations to the edit box.



- Edit the parameters as shown, so that the counter first starts from 0, adds 1 each time, and counts up for 4 cycles. Then make the counter count down starting at 2, for 3 cycles.



- Press the **Label Eqn** button to apply the equation.



## (TD) 1.13 Drawing Virtual Busses

Buses are multi-bit signals. The timing diagram editor supports several different kinds of buses to accommodate all the different ways signal information may be imported or exported from the tool. Virtual and Group buses have the ability to be converted from one to the other type by right clicking on the name.

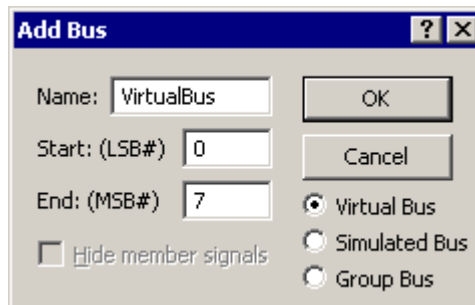
- **Virtual Bus** is a single signal defined as multiple bits. This is the most common and easiest to work with because all of the normal signal editing techniques work on it.
- **Group Bus** displays the aggregate values of its member signals. This is handy way to manage lots of single bit signals that have been imported from other sources.
- **Simulated Bus** is a simulated signal defined as a concatenation of its member signals. This is primarily designed for the testbench products so that both a member signal and the whole bus can be passed to models as needed. This is covered in the Simulated Signals Tutorial.
- **Differential Signals** are two-bit group buses that display a superimposed image of the member signal waveforms. This can also be a useful technique for overlaying two analog signals to compare them visually.

### Draw a Virtual Bus:

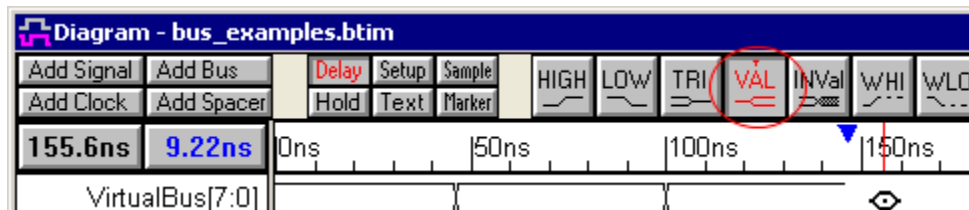
Virtual Buses are the recommended way to display and work with bus information. Virtual Buses are regular signals that have the LSB and MSB values set.

- Add an 8-bit virtual bus named **VirtualBus** with an LSB of 0 and an MSB of 7 using one of the following methods.

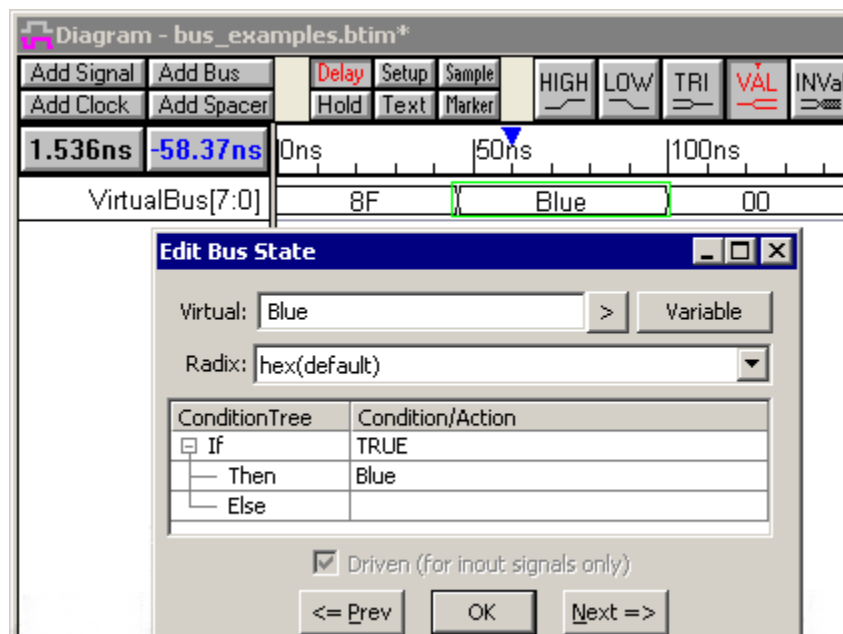
- **Fastest method:** Make sure no signals are selected, then click the **Add Bus** button to open the *Add Bus* dialog. Then select the **Virtual Bus** radio and set the **MSB** and **LSB** values.
- **Alternate method:** Add a signal and then double-click on the name to open the *Signal Properties* dialog. In the dialog edit the **MSB** and **LSB** values.



- You can sketch the virtual bus waveform using any of the graphical states, but normally virtual buses are drawn with all valid states. Press the **Valid** state button twice so that it is red and also has the red T on the top of the button. Then draw some consecutive valid states.



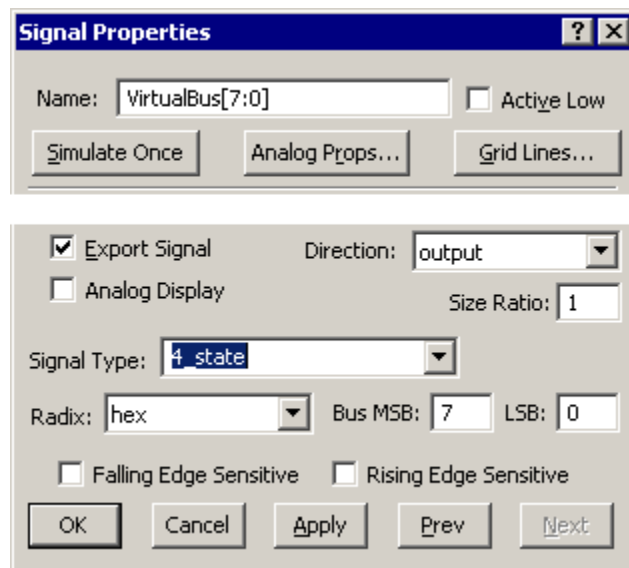
- Open the *Edit Bus State* dialog by either double-clicking on a segment OR first selecting a segment and then clicking the **HEX** button on the button bar.



- In the **Virtual** field, type in the segment value. This can be any type of data including text with spaces (e.g., A0C, 5 + 3, blue level, and 24 are all valid virtual states).
- Use **Next** and **Prev** buttons, or the **<Alt>-N** and **<Alt>-P** keys, to move between the different segments on the same bus. When you are done press the **OK** button to close the dialog.

#### *Investigate the Virtual bus using the Signal Properties dialog:*

- Double click on the **VirtualBus** signal name to open the *Signal Properties* dialog.
- On the bottom of the dialog, notice the **MSB** and **LSB** settings are the same as what you typed in the *Add Bus* dialog
- Notice the **Radix** setting which controls how the tool interprets the data in the virtual states of the waveform.



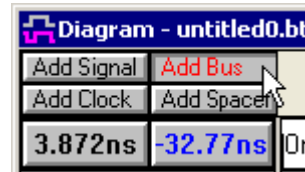
## (TD) 1.14 Drawing Group Buses and Differential Signals

Group Buses display the aggregate value of their member signals. Normally, you would use them after importing from a format that treated all signals like one-bit signals (like from a logic analyzer). They are also used to create differential signals, which are just two bit group buses with some special display settings. Before a group bus can be created, its member signals must either be specified by selecting the signal names or new signals need to be created. We will use both

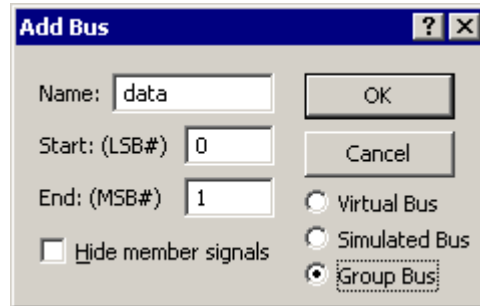
methods in this tutorial.

**To create a group bus and its member signals:**

- Make sure that no signal names are selected (clear selected signals by clicking in the *Diagram* window), then press the **Add Bus** button to open the Add Bus dialog box. If signals are selected, they will become the member signals of the bus (we will do that next).



- Name the bus **data** and set the LSB to **0** and the MSB to **1**.
- Check the **Group Bus** button.
- Verify that the **Hide member signals** check box is **NOT** checked. We want to be able to see the member signals in this demonstration.

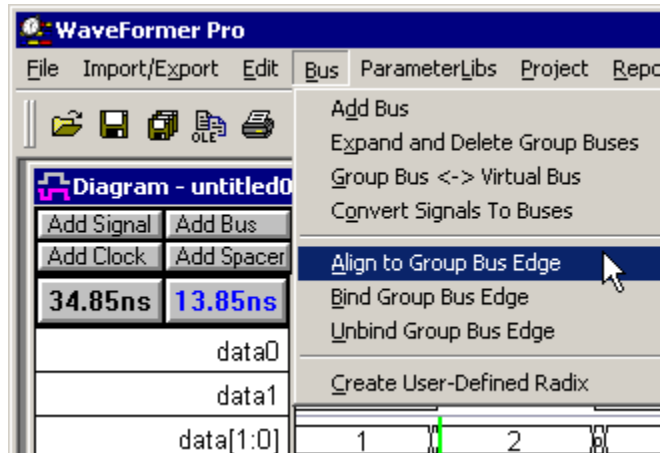


- Press the **OK** button to close the dialog and create the bus. There should be 3 signals generated: **data** (the bus), and **data0** and **data1** (the bus member signals). If the member signals are not shown, use the **View > Show Hidden Signals** to show them.

- Draw High and Low segments on **data0** then draw the opposite on **data1** (later we will use these to make a differential signal).



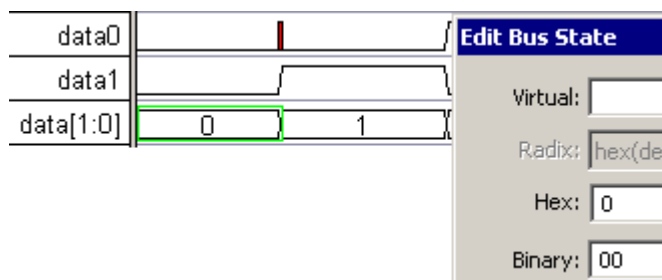
- Notice that the edges in the **data** bus are a little ragged because the edges of the member signals are not on exactly the same times.
- Select an edge on **data** and choose the **Bus > Align to Group bus edge** to snap all the edges together. All the edges can be locked together by using the **Bind** menu.



- Note the bus edge can be locked to the member edges at a particular time by selecting an edge and choosing the **Bind Group Bus Edge** menu as shown above.



- Note that you can edit the **data** bus values by double clicking on a segment to open the *Edit Bus State* dialog and changing the **Hex** or **Binary** state (but not the virtual state). The member signals change to reflect the new value.



- The red event on **data0** preserves the edge on the member signal, so that you can make consecutive changes in the bus values without stopping to add edges. To remove the red events choose the **Edit > Clear Red Events** menu, but don't do it now. Just return the first state to 1.

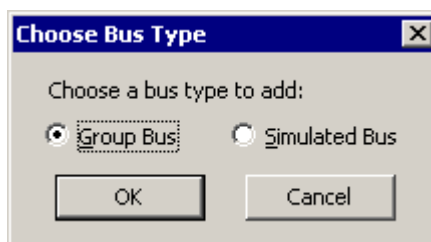
### Creating a group bus from existing signals:

Here we will create a bus using existing signals by selecting the signal names in order from LSB to MSB, then adding the bus. We will make a bus with the opposite order from the last bus.

- Select **data1** by clicking on the name. This will be the LSB of the new bus.
- Select **data0** by clicking on the name. This will be the MSB of the new bus.



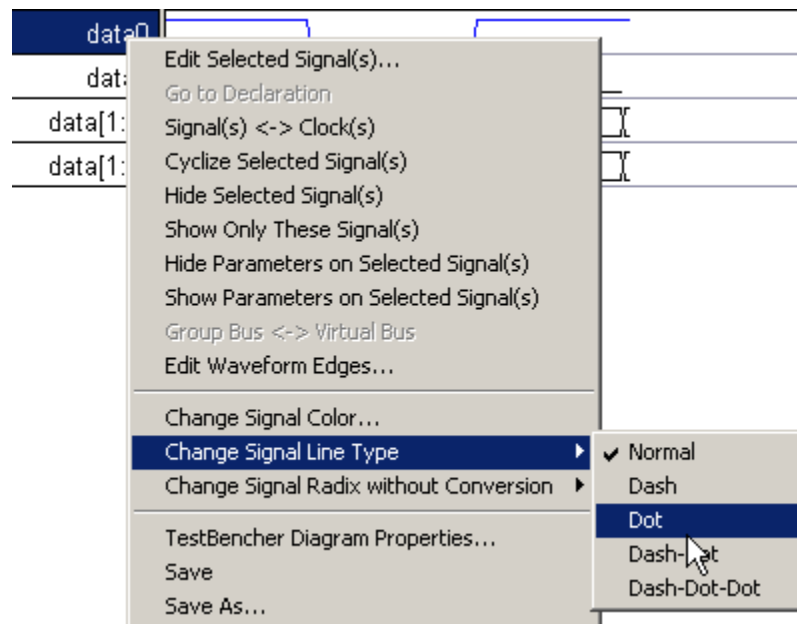
- Press the **Add Bus** button to open the *Choose Bus Type* dialog. Notice that the *New Bus* dialog did not open up because this bus will be automatically created from the selected signals.



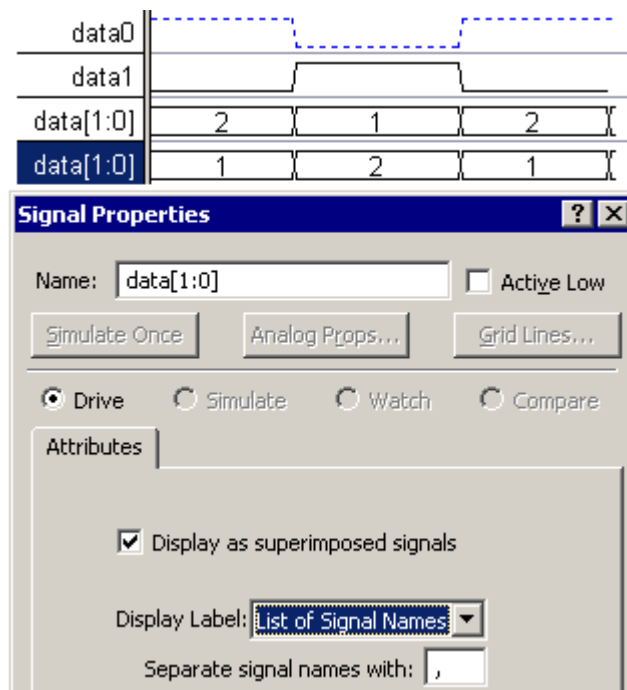
- Select the **Group Bus** radio button and click **OK** to close the dialog. Notice that a new bus, **data**, was added to the diagram and that it has a different **MSB** and **LSB** than **data**.

### Create a Differential Signal

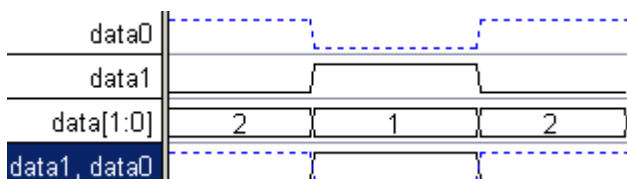
- Differential signals look best when the line type and color of one of the member signals is different. Right click on the **data0** signal and use the menus to change the *signal color* to **blue** and the *signal line type* to **dot**.



- Double click on the second **data** bus to open the *Signal Properties* dialog.
- Check the **Display as Superimposed signals** so that the signals will draw on top of each other, instead of the normal state value display.
- Change **Display Label** drop-down to **List of Signal Names** so that the bus name is replaced with the list.



- Press the **Ok** button to close the dialog and display the bus as a differential signal.



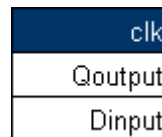


## (TD) 1.15 Working with Drawing Environment

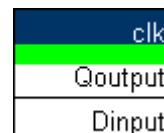
Play with moving signals and zoom in and out of the diagram.

### *Move a single signal:*

- Select the signal **clk** by clicking on the name (a selected signal will be highlighted).



- Move the mouse cursor near the very bottom of the selected signal. When the mouse cursor changes from a normal arrow to an up/down arrow, click and hold the left mouse button down. A green bar will appear.



- **Drag** the green bar until it is in between **Qoutput** and **Dinput**.
- **Drop** the green bar by releasing the mouse button. Notice that the timing diagram has redrawn itself.
- Try dropping **clk** at the very top and at the very bottom of the diagram. Leave **clk** at the bottom of the diagram.

### *Moving and reordering multiple signals:*

When several signals are highlighted and moved as a group, they will reorder themselves according to the order in which they are selected. This ability to quickly reorder signals by the order of selection will help you deal with the large numbers of member signals of buses.

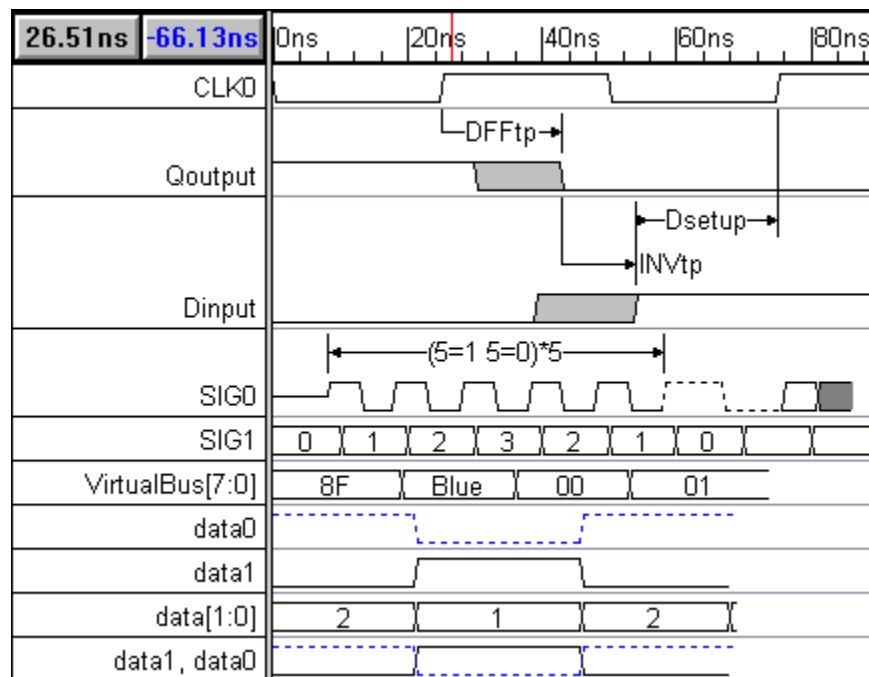
- Select **Dinput**, then select **Qoutput** by left clicking on the signal names in that order.
- **Move** the signals to the bottom of the diagram. Notice that Dinput is above Qoutput because that is the order in which they were selected.
- Select **Qoutput** and then select **Dinput**.
- **Move** the signals to the top of the diagram. Notice that Qoutput is above Dinput, because the signals were selected in that order. This is a quick way to reorder a large group of signals.
- Return the signals to their original order, (clk, Qoutput, Dinput).

### *Play with the Zoom Level*

- Click the **Zoom In** or **Zoom Out** buttons, which are located on the right hand corner of the button bar, to show less or more of the waveforms. Zooming in can also be performed with the 'Click-and-Drag' method. Simply click in the time bar over the waveforms and drag the cursor to specify the region to zoom to. Right clicking in the time bar will zoom out by a factor of two (same as pressing Zoom Out button).

## (TD) 1.16 Summary

Congratulations! You have completed the **Basic Drawing and Timing Analysis** tutorial. In this tutorial we covered how to create a timing diagram, drawing simple signals and clocks, basic timing analysis with delays and setups, and advanced drawing techniques using equations and buses.

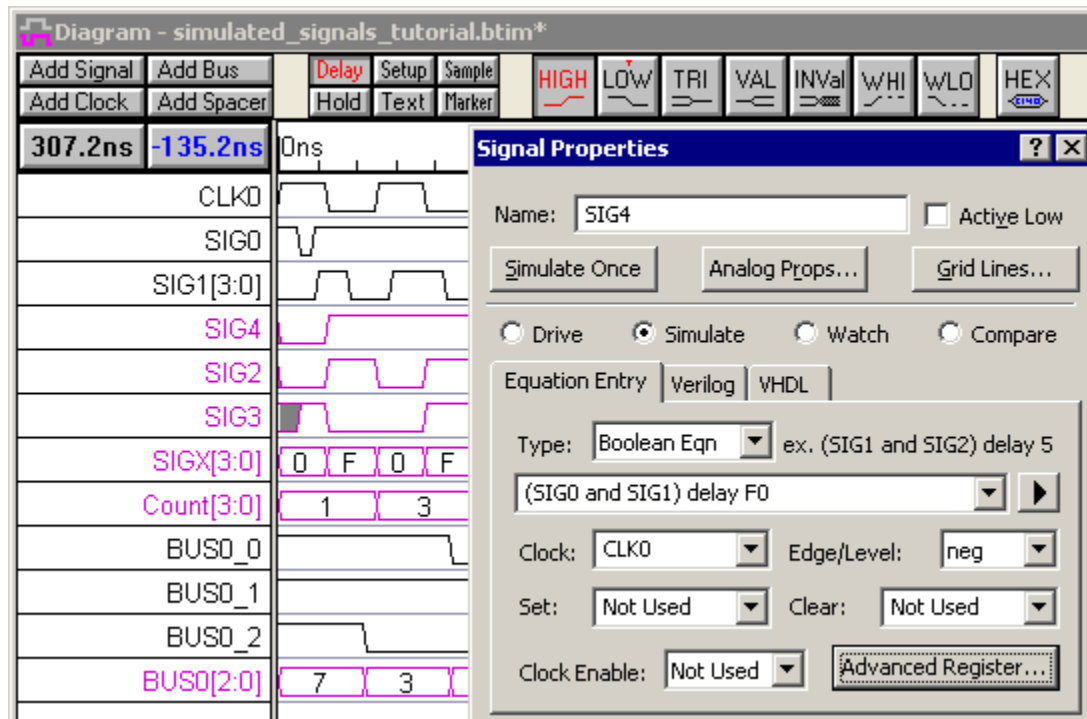


What to do next:

- If you will be doing lots of timing analysis, you need to read Section 5.1 Delays in the Timing Diagram Editor menu to find out about delay correlation and how delay times are calculated.
- If you will need to make timing diagram documentation, then do the Display and Documentation tutorial.
- If you are purchasing WaveFormer Pro or Data Sheet Pro, then do the Simulated Signals tutorial to discover the fastest way to generate timing diagrams.
- If you are working with analog signals, then do the Analog Signals tutorial.

## Timing Diagram Editor 2: Simulated Signals

Simulated Signals reduce the amount of time needed to draw and update a timing diagram, because the waveform is described using a Boolean or registered logic equation. With Simulated Signals you will no longer have to figure the output of a combinational circuit or calculate the critical path of a synchronous circuit by hand. WaveFormer Pro has an internal interactive simulator that supports multi-bit equations with true min-max timing, unlike traditional simulators that can only represent single-valued delays. This tutorial contains some simple examples of Boolean and registered logic equations that showcase the simulator's capabilities.



To do this tutorial, you will need WaveFormer Pro or a higher level product. SynaptiCAD has also included Simulated Signals with the VeriLogger and TestBench products, even though they have a built-in Verilog simulator, because this feature makes it easier to generate test benches and timing diagrams. In WaveFormer, it is the backbone of the timing analysis and design features.

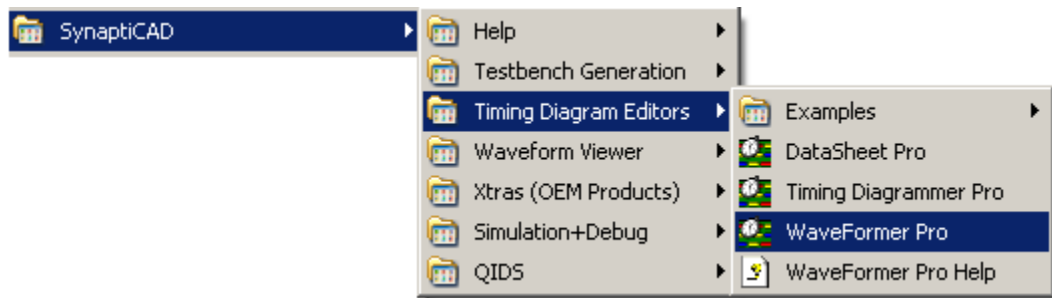
This tutorial assumes that you are able to draw signals and can add delays, setups, and holds to those signals. We recommend that beginners start with the Basic Drawing and Timing Analysis Tutorial to learn the basics of timing diagram editing before attempting this tutorial.

### (TD) 2.1 Setup for Simulation

In the next few sections we will simulate signals using Boolean and registered logic equations. The inputs to a simulated signal are other drawn signals, so in this section we will create a timing diagram and a free parameter that we will use in the subsequent steps.

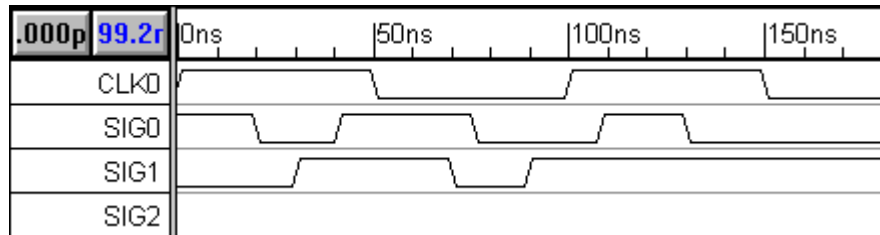
#### *Run WaveFormer Pro or higher:*

- Run WaveFormer Pro, DataSheet Pro, VeriLogger, or one of the more advanced products. If you are evaluating Timing Diagrammer Pro or one of our Viewers and you would like to learn about the simulation features, close the program and restart the evaluation version in WaveFormer Pro mode.



### Create a Timing Diagram:

- Choose **File > New Timing Diagram** menu to open an empty timing diagram window.
- Sketch the following timing diagram. Clock **CLK0** has the default 100ns period. Just approximately sketch the waveforms for SIG0 and SIG1; exact edge placement is not necessary for this tutorial. Leave SIG2 blank, because it will be the simulated signal.



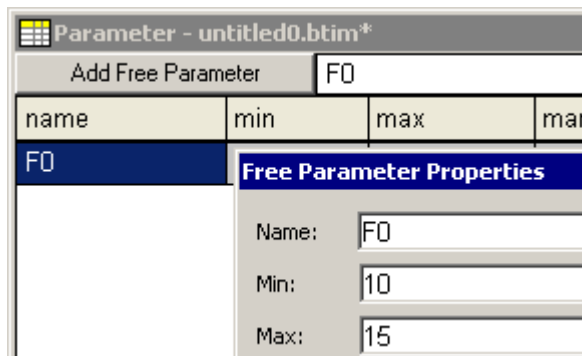
### Create a Free Parameter:

We will also be experimenting with the min and max timing features of the Boolean equations, so create a Free Parameter to use in the equations.

- In the *Parameter* window, press the **Add Free Parameter** button to add a free parameter **F0**.



- Double-click on **F0** to open the *Parameter Properties* dialog.
- Enter a min time of **10**, and a max time of **15**, then press the OK button to close the dialog.

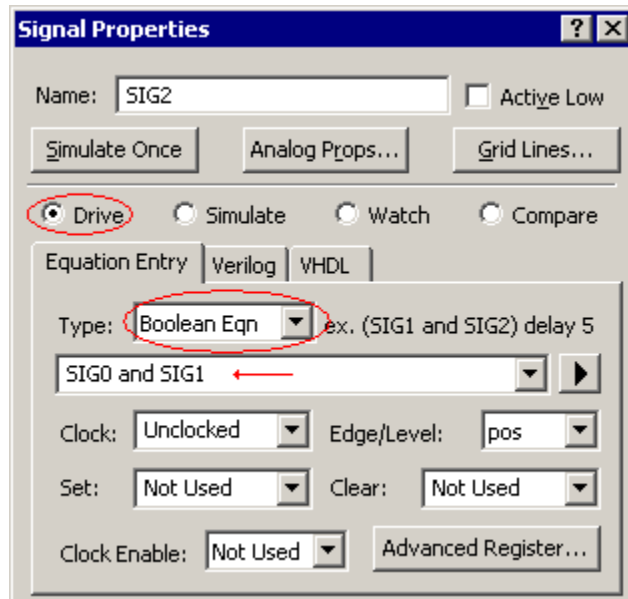


## (TD) 2.2 Simulate a Boolean Equation

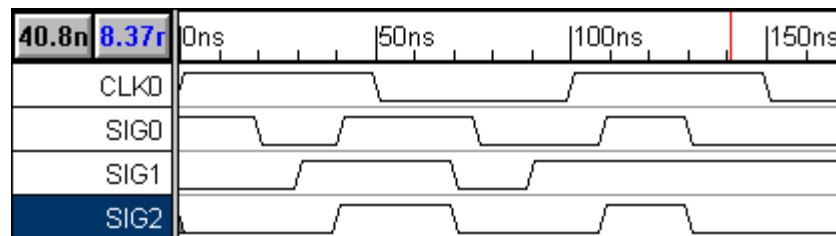
A simulated signal is created by adding a Boolean equation to the *Signal Properties* dialog for that signal. The dialog accepts Boolean equations in either VHDL, Verilog, or SynaptiCAD's enhanced equation syntax. The SynaptiCAD format supports the following operators: **and** or **&**, **or** or **|**, **nand**, **nor**, **xor** or **^**, **not** or **~** or **!**, and **delay**. We will cover the **delay** operator in the next section.

**Simulate a Boolean equation once:**

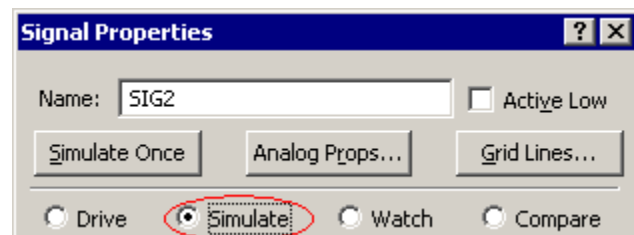
- Double click on the **SIG2** signal name to open the *Signal Properties* dialog. Move the dialog so that you can see the dialog and the 3 signals at the same time.
- Enter **SIG0** and **SIG1** into the edit box below the **Boolean Eqn** type box (signal names are case sensitive).
- By default, all signals are **Drive** signals that will only simulate when the user presses the Simulate Once button.



- Click the **Simulate Once** button at the top of the dialog and watch the signal draw itself. Notice that SIG2 is the result of the Boolean Equation "SIG0 and SIG1". SIG2 is drawn in black to indicate that it will not re-simulate automatically.

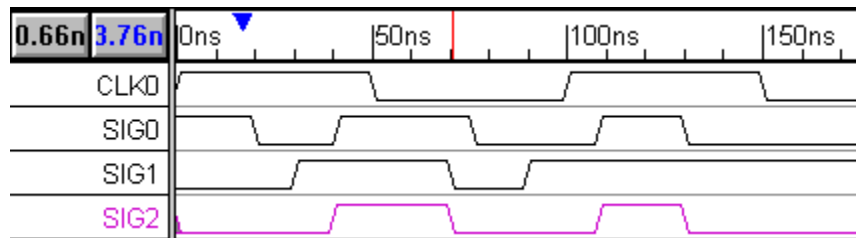
**Continuously Simulate the Boolean Equation:**

- First make sure the program is in the continuously simulate mode:
  - If you are using VeriLogger Pro or TestBench Pro, make sure that the program is in **Auto Run** simulation mode. **Debug Run** mode will not continuously update signals. The Auto Run/ Debug Run simulation mode button is located on the simulation toolbar, in the upper left of the window below the Project menu.
  - If you are using WaveFormer, then check the **Options > Diagram Simulation Preferences** menu to make sure that the **Continuously Simulate** box is checked. WaveFormer does not have the Auto Run/Debug Run button.
- To make the signal continuously simulate, check the **Simulate** signal type button.



- Notice that the SIG2 is now drawn in purple. This color means that the signal is being

continuously simulated, and changes in the input waveforms cause automatic re-simulations.



- Move some of the edges on SIG0 and SIG1 and watch SIG2 re-simulate. (Notice that you cannot drag and drop SIG2's signal edges because they are calculated edges).

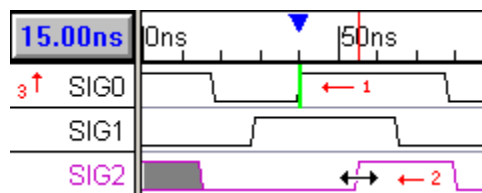
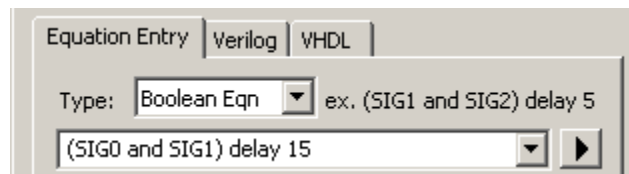
## (TD) 2.3 Boolean Equations with Delays

Next we will modify the Boolean equation to take into account the propagation delay through the AND gate. First we simulate a simple 15ns delay, then we will simulate a min/max delay. The SynaptiCAD **delay** operator takes a signal on the left, and a time or parameter name on the right, and returns a signal. If a parameter name is used on the right hand side of the delay operator, then the equation will simulate true min/max timing. This true min/max timing is the main advantage that SynaptiCAD's format has over the VHDL or Verilog format.

### Simulate a simple delay:

- Add a **15ns** delay to the **SIG2** Boolean equation. The box accepts Verilog, VHDL, or SynaptiCAD syntax, so these equations are all equivalent to each other.
- Press the **Apply** button at the bottom of the dialog to notify the simulator about the change in the equation.
- Verify that **SIG2** is 15ns delayed, by first selecting an input edge then moving the mouse over the resulting edge on **SIG2**. The blue delta read out will say 15ns.

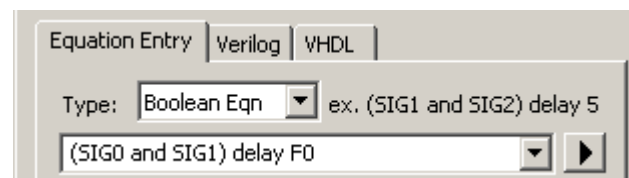
```
#15 (SIG0 & SIG1)
(SIG0 and SIG1) after 15
(SIG0 and SIG1) delay 15
```



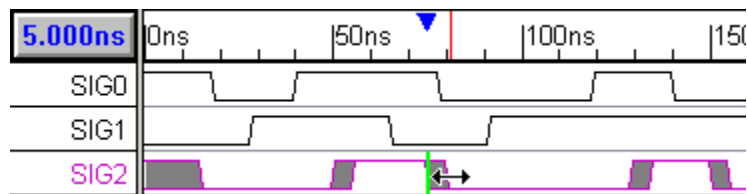
### Simulate a true min/max delay using SynaptiCAD syntax:

The most powerful feature of a timing diagram editor is the ability to display min/max timing using the grey uncertainty regions. To make the Simulated Signals support min/max timing, we created the SynaptiCAD **delay** operator, because the delay operators in VHDL and Verilog only support a single delay.

- Edit the equation so that the delay references the free parameter **F0** then press the **Apply** button to notify the simulator about the change in the equation.



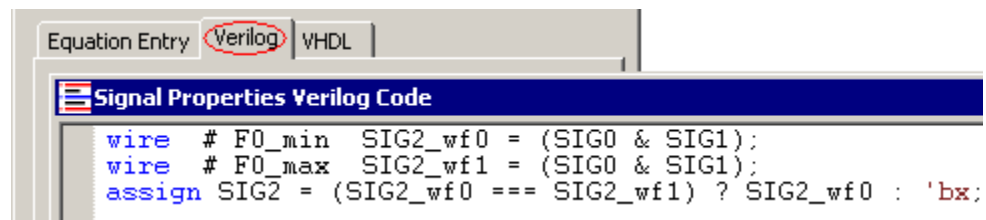
- Notice that SIG2 has grey uncertainty regions that are 5ns wide (F0.max - F0.min).



#### View the HDL code that models the Boolean equation:

The timing diagram editor takes the Boolean equation and generates Verilog or VHDL code necessary to perform the equation. You can edit this code directly to perform more complex functions. The tool ships with an embedded Verilog simulator that executes the code, so if you change VHDL code you will have to provide the tool with a VHDL simulator. The manual explains how to configure for a different simulator.

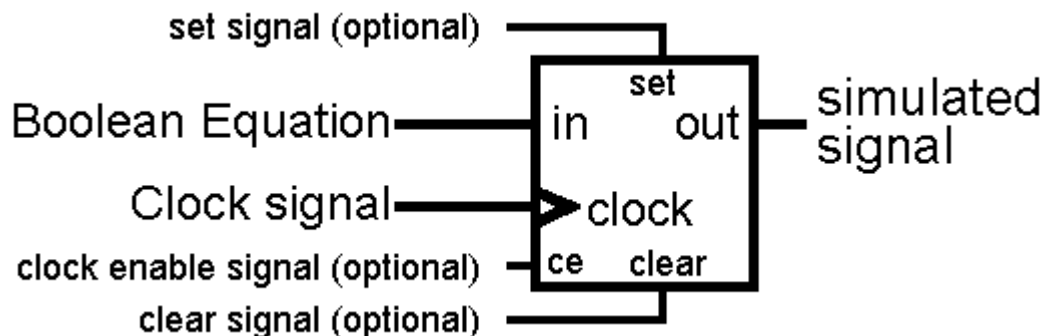
- Press the **Verilog** tab to open an editor window that displays the generated Verilog code. Do not change the code now.



- This example demonstrated true min/max simulation, however Min-Only and Max-Only simulations can be performed by changing the selection in the **Timing Model** drop-down list of the *Simulation Preferences* dialog box. The *Simulation Preferences* dialog can be opened using the **Options > Diagram Simulation Preferences** menu option. The **Timing Model** drop-down list is in the upper right corner.

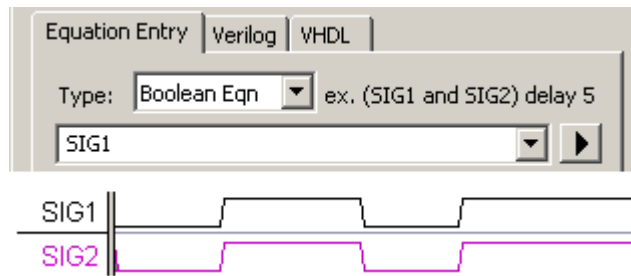
## (TD) 2.4 Register and Latch Signals

The Interactive Simulator can register or latch the result of a Boolean equation. This circuit is similar to most FPGA cells and can model a large number of components. Below is a figure of the register that a Simulated Signal models. If no clocking signal is chosen, then the Boolean equation goes straight to the signal output as shown in the previous sections. Note: setting the MSB/LSB fields in the Signal Properties dialog will "parallelize" the circuit, allowing multi-bit registers such as counters, shifters, etc. to be modeled).

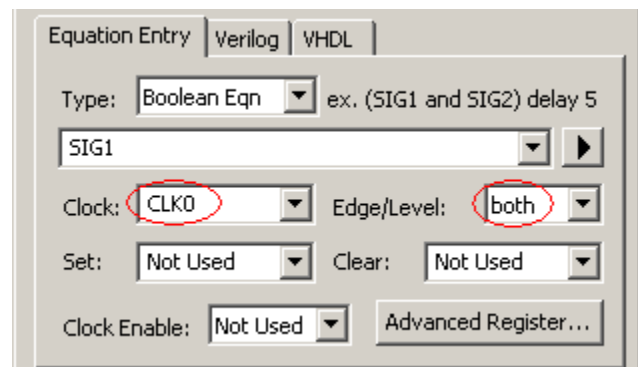


### Experiment with the register and latch equations:

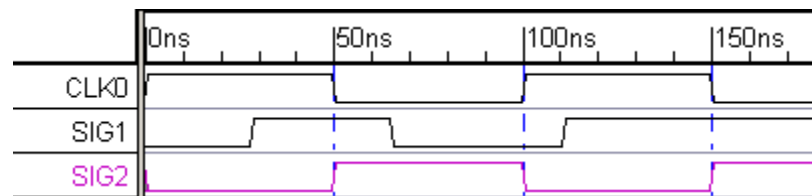
- Change the **SIG2** equation to just one term **SIG1** and press the **Simulate Once** button. SIG2 should be an exact copy of SIG1. When we register SIG2 you can visually compare it to SIG1 to see the effects of the register.



- Set the clock control to **CLK0**. Choosing a clock brings in the register/latch model to buffer the Boolean equation.
- Set the edge/level control to **both**, to indicate that both the rising and falling edges of the clock are triggering edges. Since it is edge triggered a register circuit will be created rather than a level sensitive latch.

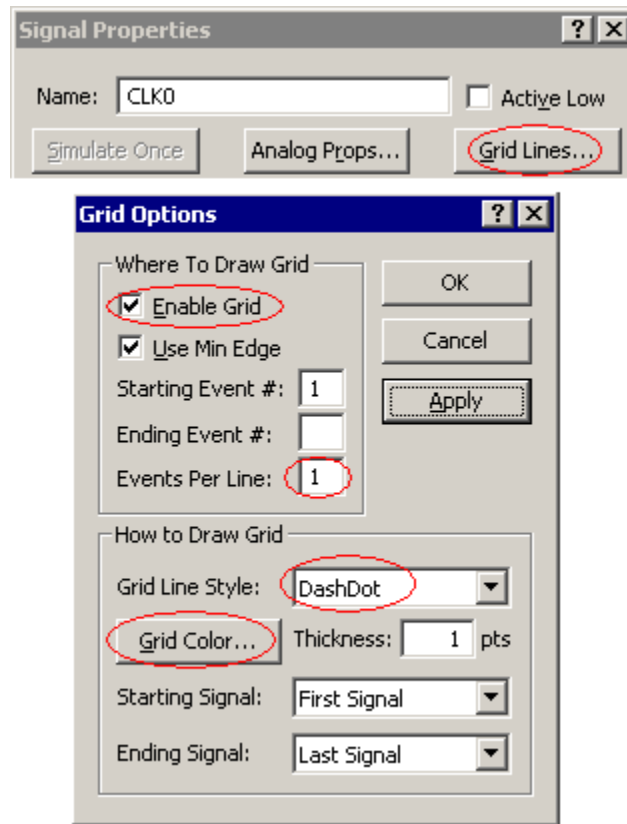


- Click the **Simulate Once** button to simulate the circuit. Notice that SIG2 only transitions when CLK0 has a positive or negative edge transition (move some edges on SIG1 to verify this).





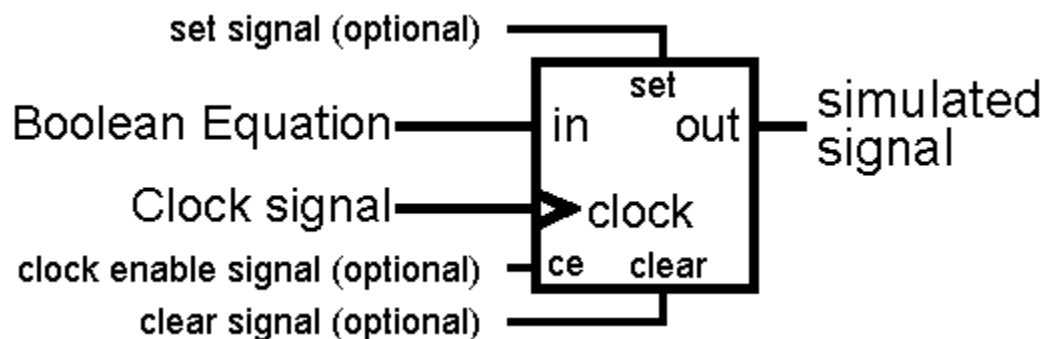
- To make the diagram look like the above picture, we hid **SIG0** because it is not being used.
- We also added **Grid Lines** to every edge of the clock. To do that double click on the clock name to open the *Signal Properties* dialog and press the **Grid Lines** button. This opens the *Grid Options* dialog. Play with the controls and hit **Apply** until you get an image that you like.



- Change the Edge/Level control value and press the **Simulate Once** button to verify the operation of the different register and latch circuits.

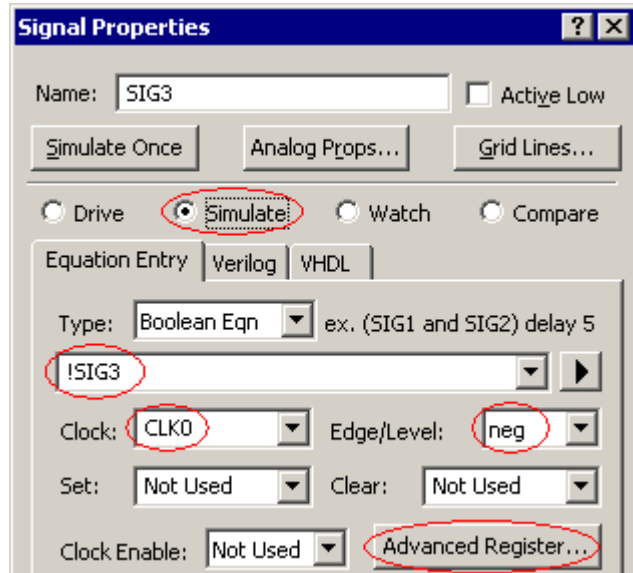
## (TD) 2.5 Set and Clear Lines

The **Set** and **Clear** lines are useful when defining circuits whose initial value needs to be specified. In this example we demonstrate how to design a **divide by 2 circuit** using a negative edge triggered register with an asynchronous active-low set line.



*Use the Set line to define an initial state:*

- Click the **Add Signal** button to create a new signal named **SIG3**. Then double click on the signal name to open the *Signal Properties* dialog.
- Check the **Simulate** button.
- Type **!SIG3** into the *Boolean Equation* edit box (it references itself).
- Choose **CLK0** from the *Clock* drop down list box.
- Choose **neg** from the *Edge/Level* box.



- Notice that the waveform for **SIG3** is completely gray but that the status bar (in the lower right corner of the window) reports **Simulation Good**. This is because **SIG3**'s Boolean equation references itself but it does not provide the simulator with a known start state.

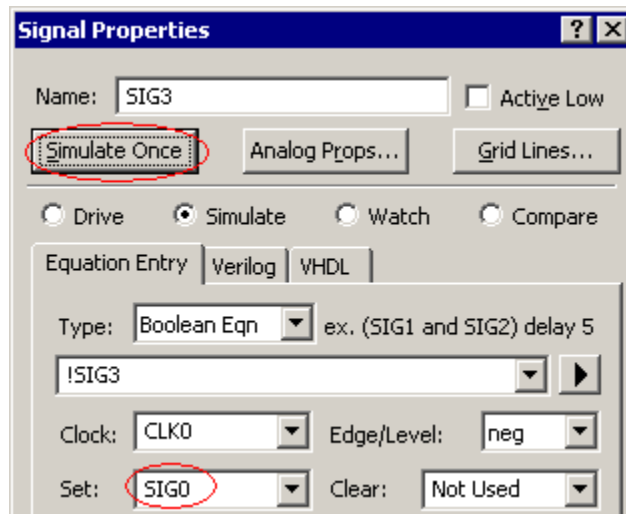
SIG3

- Press the **Advanced Register** button (see above) to open the *Advanced Register and Latch Controls* dialog. Notice that all the register and latch individual propagation times, setup/hold constraints, clock enable, and set/clear options are set here. Tip: the Global defaults are set using the **Options > Simulation Preferences** menu.

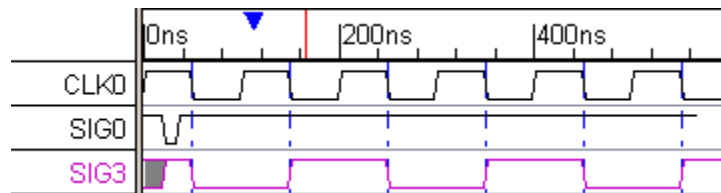


- Make sure the **Active Low** and the **Asynchronous** check boxes in the *Set and Clear* section are checked. Click **OK** to close the dialog.

- Choose **SIG0** from the **Set** drop down list box.
- We hid SIG0 in a previous section, so choose the **View > Show Hide Signals** menu and show SIG0.
- Press the **Simulate Once** button to notify the simulator of the change in the model.



- Notice that **SIG3** now has a simulated waveform. Redraw **SIG0** so that it goes **low** early in the timing diagram, and then stays **high** for four or five clock cycles.



- Experiment with **SIG0** to see how the active low set line affects the operation of the flip-flop. Remember that we set the model to have an asynchronous low set signal.
- To make the above diagram, we hid the unused signals. We also changed the clock grid so that the **starting event** was 2 and there were **2 events per line**, that way we got grid lines on just the negative edges.

## (TD) 2.6 Multi-bit Equations

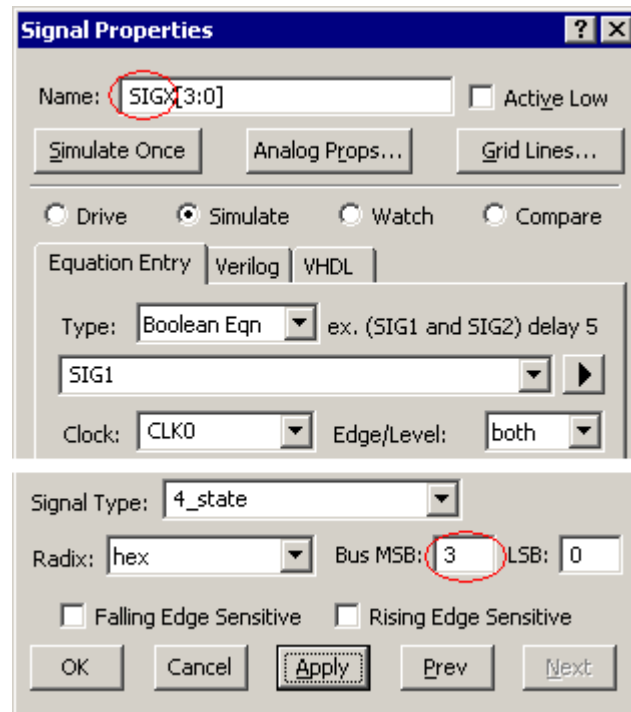
The Interactive Simulator can automatically generate multi-bit equations for the register, latch and combinatorial logic circuits. To convert a register or latch circuit into a multi-bit signal, change the MSB of the input signal and the MSB of the register or latch. If the sizes of the signals do not match, WaveFormer maps as many LSB's as it can. The following example uses only a simple equation to demonstrate the LSB mapping feature, however multi-term Boolean equations are completely supported.

### Create a Multi-bit Signal by changing the MSB setting:

- Create a copy of **SIG2**. Click on the **SIG2** name in the Label window to select it. Select the **Edit > Copy Text and Signals** menu option to copy the signal, then the **Edit > Paste** option to paste the signal. There are now two signals named **SIG2** in your diagram.



- Double click on the bottom **SIG2** to open the *Signal Properties* dialog, and rename the signal to **SIGX**.
- Type **3** in the **Bus MSB** edit box. This will make **SIGX** a 4-bit signal and add a **[3:0]** to the end of the name.
- Press the **Apply** button to notify the simulator of the changes.

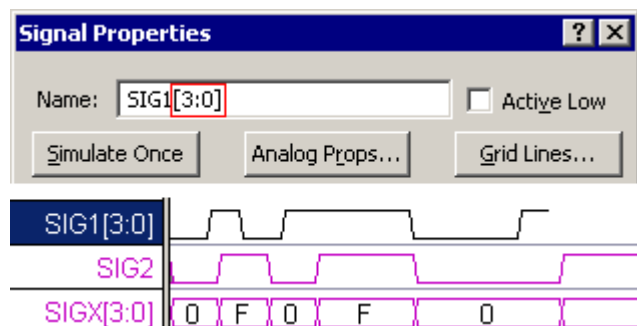


- **SIGX**'s waveform is now drawn as a bus with a 4 bit binary display. Only the LSB of **SIGX** is working because the input signal **SIG1** is a single bit. Compare **SIG2** and **SIGX** and verify that they are the same values.



#### Change the input signal to a multi-bit signal:

- Double-click on **SIG1** to open the *Signal Properties* dialog, and add **[3:0]** to the end of the name. This has the same effect as changing the values in the **MSB** and **LSB** edit boxes.
- Press the **Apply** button. Now all four bits of **SIGX** should be toggling between 0 and F. If the radix is in Binary, the signal will toggle 1111 and 0000. The radix box is located in the lower left part of the dialog.
- If you want to further experiment with multi-bit signals, change **SIG1**'s graphical segments to Valid regions instead of Highs and Lows. Then double click on a valid region to open the Edit Bus State dialog box. Type different hex values from 0 through F, like 5 or A, into the Virtual edit box and watch how it affects the output of **SIGX** and **SIG2**. Since **SIG2** is a single bit signal it uses only the LSB of the input signals.



## (TD) 2.7 Design a Multi-Bit Counter

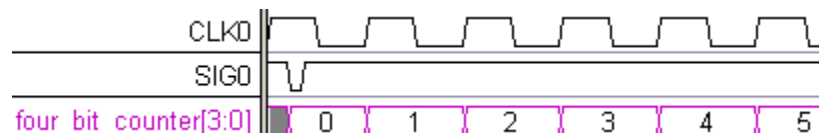
When a multi-bit signal is clocked the register/latch circuit shown will be instantiated one time for each bit of the signal. This allows you to use one signal to represent the operation of a multi-bit counter or buffer.

### Use a multi-bit signal to make counter:

- Press the **Add Signal** button to add a new signal.
- Rename the signal to **four\_bit\_counter**.
- Set the type to **simulate**.
- Type in the equation **four\_bit\_counter +1**.
- Set the clock to **CLK0** with an trigger edge of **pos**. This will make the signal change only on the positive edges of the clock.
- Set the clear line to **SIG0**. If you press the **Advanced Register** button you can verify that the clear line will be active low and asynchronous, so that a pulse low on this line will clear the signal registers to zero.
- Set the Msb to **3** to instantiate the multiple registers.

The **Signal Properties** dialog box for the signal **four\_bit\_counter[3:0]** is shown. The **Name** field contains **four\_bit\_counter[3:0]**. The **Active Low** checkbox is unchecked. The **Simulate** radio button is selected. The **Equation Entry** tab is active, showing the **Type** as **Boolean Eqn** and the **Equation** as **four\_bit\_counter+1**. The **Clock** is set to **CLK0** with an **Edge/Level** of **pos**. The **Set** is **Not Used** and the **Clear** is **SIG0**. The **Advanced Register...** button is visible. At the bottom, the **Radix** is **hex**, **Bus MSB** is **3**, and **LSB** is **0**.

- Press the **Apply** button to simulate the signal. Notice that the signal will be a grey unknown region until the SIG0 goes low to clear the register.



- If you press the **Verilog** tab, in the **Signal Properties** dialog you can see that the register is going to be 4 bits wide (it is the first parameter that is passed into the register).

The **Signal Properties Verilog Code** window shows the Verilog code for the signal **four\_bit\_counter**. The code is as follows:

```

wire [3:0] four_bit_counter_wf21 = four_bit_c
wire [3:0] four_bit_counter_wf20;
registerP_Asyn #(4,1,1) registerP_Asyn_four_b:
four_bit_counter_wf20,CLK0,four_bit_counte
1'b0,1'b1,SIG0,
$realtobits(0.0),$realtobits(0.0),
$realtobits(0.0),$realtobits(0.0));
assign four_bit_counter = four_bit_counter_wf:

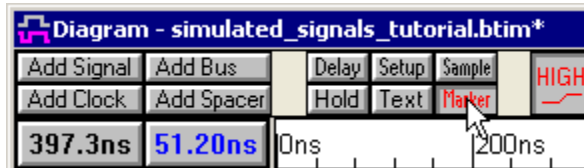
```

## (TD) 2.8 End Diagram Marker Stops Simulation

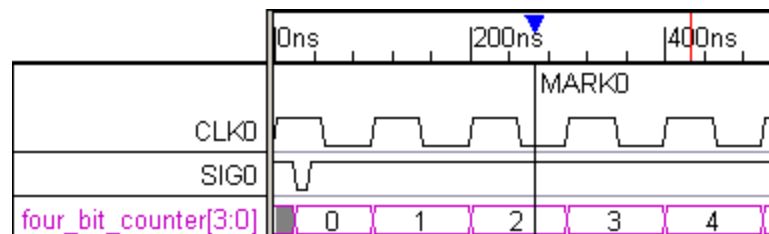
Normally the simulation will continue to the end of the last drawn signal or about one clock cycle past the drawn signal. However, the exact end of simulation can be controlled using Marker line.

*Use an End Diagram Marker to control the end of simulation time:*

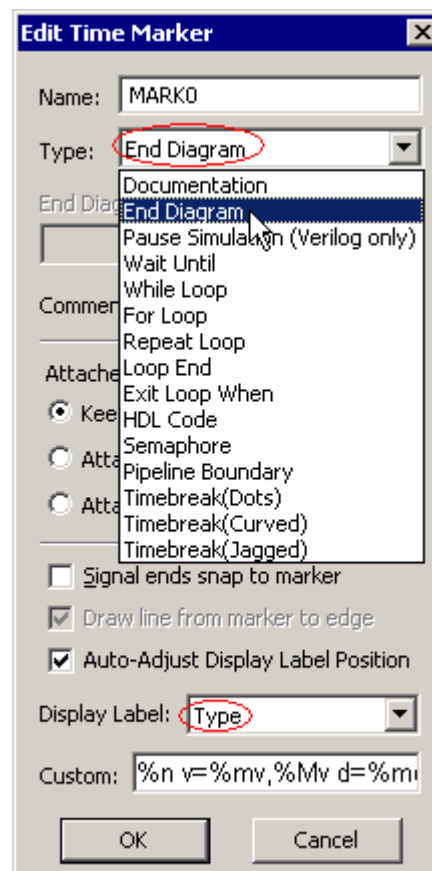
- Press the **Marker** button so that right clicks will add marker lines to the diagram.



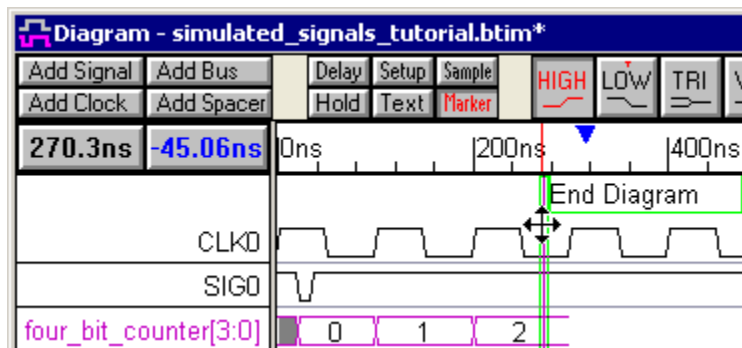
- Right click inside the diagram window to add a **Marker** line. By default all new markers are just documentation lines that will not effect the timing or simulation of the diagram.



- Double click on **MARK0** to open the *Edit Time Marker* dialog.
- Set the type to **End Diagram**. But also notice all the other types available. The *timebreak* types compress time and hide parts of the dialog. The loop markers are used by TestBench and Reactive Test Bench generation to create complex test bench code.
- Set the display label to **Type**. This will make the marker display its type instead of its name. Also notice all the display label options to control exactly what the marker displays.
- Press the **Ok** button to close the dialog



- Notice that the simulation ends at the clock cycle after the End Diagram Marker. Grab the marker with the mouse and drag it around to control the simulation end.



## (TD) 2.9 Behavioral HDL Code

In addition to the simulation of Boolean and registered logic circuits, SynaptiCAD products can simulate behavioral Verilog code, and if you provide a VHDL simulator it can also simulate behavioral VHDL code. The behavioral code is entered directly in to the **Verilog** or **VHDL** tabs in the *Signal Properties* dialog, instead of using the **Equation Entry** tab that we used for the Boolean equations in the last sections. There is also a template feature that generates code from a Boolean equation and allows you to modify the generated code.

In this section we will use a register template as a starting point to build a circuit that asynchronously counts the number of edges that occur on SIG1 and synchronously presents the total number of edges on the positive edge of the clock.

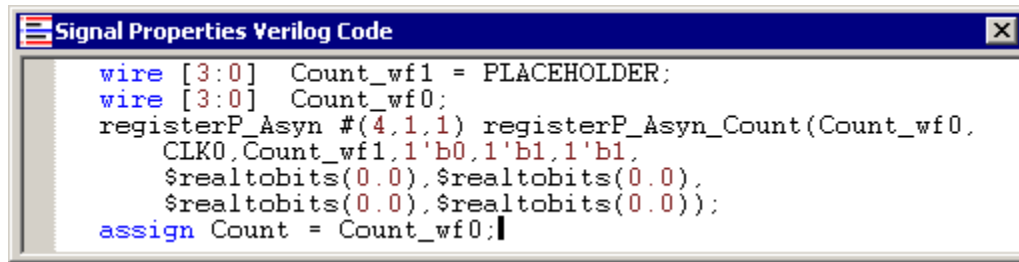
**Use a partially defined signal to generate PLACEHOLDER code:**

- Add a new signal named **Count**.
- Select the **Simulate** type setting.
- Set the clock to **CLK0** with an edge/level of **pos**.
- Set the MSB to **3**.
- Press the **Apply** button to apply the changes and generate the code. Since there is no Boolean equation for the signal, this is also going to generate a compile error and all the simulated signals in the diagram will go grey. If no compile error happens, then choose **Options > Diagram Simulation Preferences** menu and check the **Continuously Simulate** box.

Compile Error

- Next, press the **Verilog** tab to open and editor window containing the the template code. The

internal wire names Count\_wf\*\*\* will vary depending on how many signals you have simulated.

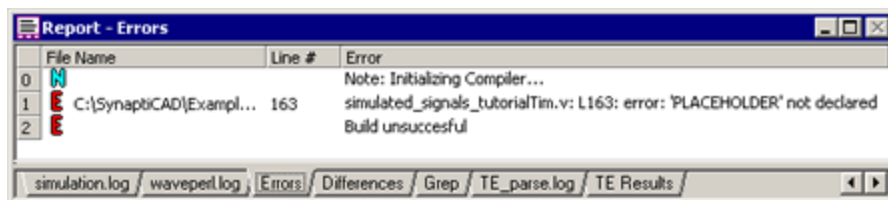


```

wire [3:0] Count_wf1 = PLACEHOLDER;
wire [3:0] Count_wf0;
registerP_Asyn #(4,1,1) registerP_Asyn_Count(Count_wf0,
    CLK0,Count_wf1,1'b0,1'b1,1'b1,
    $realtobits(0.0),$realtobits(0.0),
    $realtobits(0.0),$realtobits(0.0));
assign Count = Count_wf0;

```

- The **registerP\_Asyn** line instantiates (defines an instance of) a 4 bit positive-edge-triggered register of the type used by the logic wizard. This register takes PLACEHOLDER as an input and outputs a synchronized version on **Count**.
- The auto generated variable **PLACEHOLDER** is undefined and will not simulate. If a Boolean equation was defined for the circuit, it would replace the PLACEHOLDER variable. This error will be displayed in the Report window under the error tab.



#### Add behavioral code to the generated code:

We will use the PLACEHOLDER variable to store the edge count. First we will define PLACEHOLDER, give it an initial starting value, then define an *always* process that triggers each time the SIG1 changes. Since Count is buffered by a positive edge triggered register, it will not display the PLACEHOLDER value until the positive edge of the clock.

- Either copy-and-paste or type the first four lines the below code directly into the *Signal Properties Verilog Code* window (add the bold lines):

```

reg [3:0] PLACEHOLDER;
initial PLACEHOLDER = 0;
always @(SIG1)
    PLACEHOLDER = PLACEHOLDER + 1;
wire [3:0] Count_wf1 = PLACEHOLDER;
wire [3:0] Count_wf0;
registerP_Asyn #(4,1,1) registerP_Asyn_Count(Count_wf0,
    CLK0,Count_wf1,1'b0,1'b1,1'b1,
    $realtobits(0.0),$realtobits(0.0),
    $realtobits(0.0),$realtobits(0.0));
assign Count = Count_wf0;

```

- Click the **Apply** radio button. Verify that Count is counting the edges of SIG1. The new edge count is presented on each positive edge of CLK0. The Count starts at one because there is a 1'bz to 1'b0 transition at time zero on SIG1.



#### Code explanation:

The code that you just entered is behavioral Verilog code.



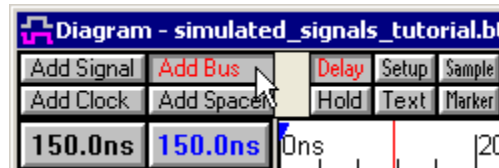
- The first line defines PLACEHOLDER as a 4-bit register. PLACEHOLDER needs to be defined as a register rather than a wire in this case because it must "remember" its value. Verilog wires don't remember their values so they must be constantly driven to retain their value.
- The second line initializes the value of PLACEHOLDER to 0 when the simulator first runs.
- The third and fourth lines contain an always block (note for VHDL users: these work like VHDL process blocks). Whenever **SIG1** changes state, the always block will execute, incrementing PLACEHOLDER.
- The rest of the lines consist of the automatically generated template code that instantiates the synchronizing register.

## (TD) 2.10 Simulated Bus Signals

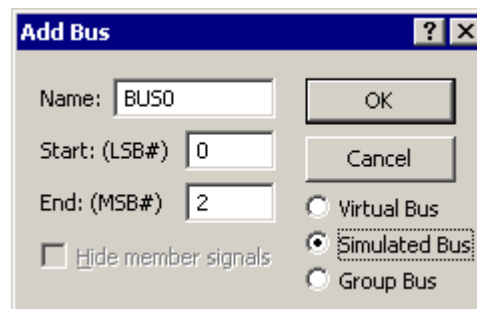
Simulated Buses are similar to Group Buses. The primary difference is that the bus is generated using a Boolean Equation. A simulated bus can be referenced in another signal's Boolean equation, (group buses cannot). Also, TestBencher will generate a Boolean equation for the timing transaction so that the simulated bus can include input signals as member signals.

### Create a Simulated Bus:

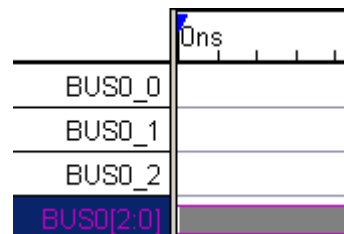
- Make sure that no signals are selected, and press the **Add Bus** button to open the *Add Bus* dialog. Note, if a signal was selected a different dialog will open and you cannot add a simulated bus from there.



- Set the **LSB** to **0** and the **MSB** to **2**.
- Select the **Simulated Bus** radio button.
- Press the **OK** button to close the dialog and create the bus and member signals.



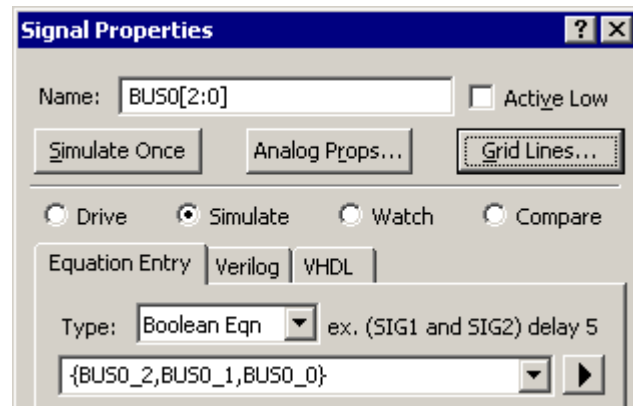
- BUS0's waveform will be grey because none of the member signals are defined.



- Draw on the member signals and see the bus simulate.

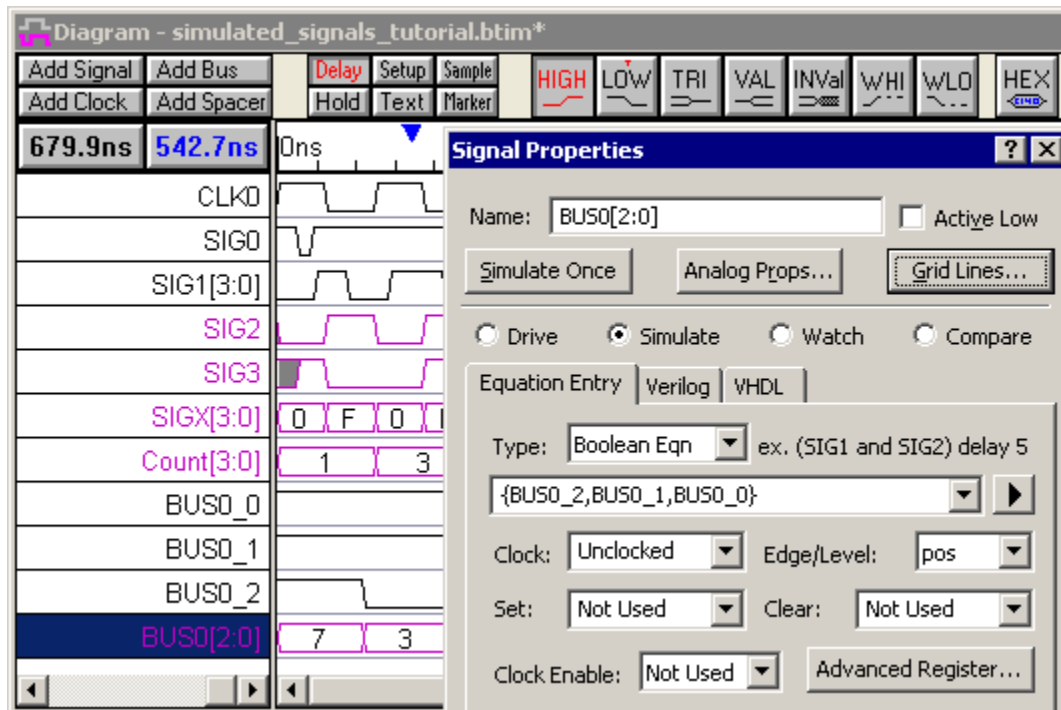


- Double click on **BUS0[2:0]** label to open the *Signal Properties* dialog. Notice that the bus is defined as a concatenation of the member signals.



## (TD) 2.11 Summary of Simulated Signals Tutorial

Congratulations! You have completed the Simulated Signals tutorial. In this tutorial, you have used a Boolean Equation to define the waveform of a signal, and experimented with the **delay** operator and multi-bit equations. You have also worked with the internal register and latch circuits and the template function for generating a starting point for writing behavioral code. And finally you have generated a simulated bus.



More information on simulated signals and the internal simulator look at:

- Chapter 4: Simulated Signals and VHDL and Verilog Export in the Timing Diagram Editor manual.
- Advanced Modeling and Interactive Simulation Tutorial demonstrates how to model a complex circuit using external models, behavioral HDL code, and incremental simulation techniques.

If you currently own Timing Diagrammer Pro or one of the SynaptiCAD Viewers, then these products can be upgraded to WaveFormer Pro or Data Sheet Pro so that you will have access to

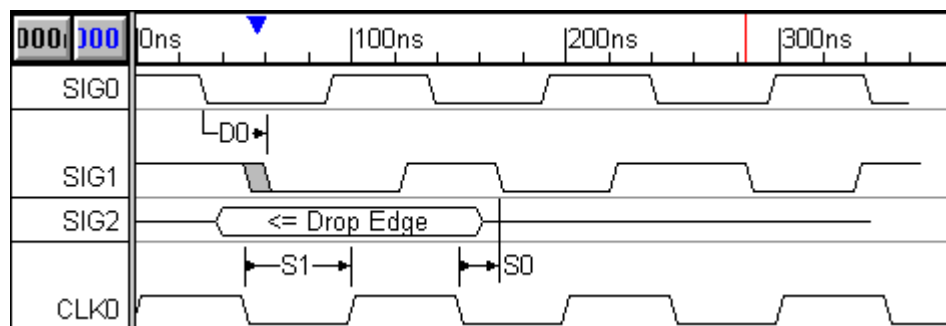
the Simulated Signal features.

## Timing Diagram Editor 3: Display and Documentation

This tutorial introduces techniques for controlling the display of parameters, clocks, waveforms, markers and text objects. These techniques that will allow you to control exactly what your timing diagrams look like and what information is displayed. It is recommended that you are comfortable drawing waveforms and adding parameters before you begin this tutorial. These features are covered in the *Basic Drawing and Timing Analysis* tutorial.

Load the starting timing diagram for this tutorial:

1. Open the file **tutdocstart.btim** in the **SynaptiCAD\Examples\Tutorial\DisplayAndDocumentation** directory.
2. Select the **File > Save As** menu option, and save this file as **mystart.btim**.

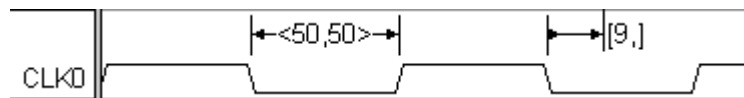


### (TD) 3.1 Controlling Parameter Display String

A Delay, Setup, Hold, or Sample parameter can display a specific attribute or a custom display string. The *Parameter Properties* dialog box has the **Display Label** and **Custom String** controls that manage the display properties of the parameter. Individual instances or all instances of a parameter are configured depending on where the *Parameter Properties* dialog is opened. For individual instances double click on a parameter in the *Diagram* window. To configure all instances of a parameter double click on a parameter in the *Parameter* window.

Setups and Holds are often used in a timing diagram to display information like distance measurements or used for cycle annotation, because these parameters monitor state information instead of forcing edges like a delay parameters. First lets experiment with using simple attributes to display margin and distance calculations of the setup parameters.

1. Double-click on the setup label **S0** to open the *Parameter Properties* dialog. Arrange the dialog so that you can see the S0 in the diagram window and the dialog at the same time.
2. Use the **Display Label** drop-down list box to select the **min/max Margin** display. Notice that the label for the parameter now displays, **[9,]**, the min/max margin, instead of the name S0. This display will change if the setup's edges are moved. Margin is the amount of time available before a setup or hold constraint is violated. The max is blank because there is no maximum setup time specified in the parameter.
3. Click the **Next** button to display the setup **S1** in the *Parameter Properties* dialog. We will use S1 to display the distance between two edges, so we have not bothered to define the min and max values.
4. Select the **Distance** from the **Display Label** drop-down list box. The label now shows the minimum and maximum distances between the transitions.
5. Check the **Outward Arrows** check box to make the parameter's arrows display the type of arrows that are usually used for distance measurements.



NOTE: The default display for all parameters can be set using the **Options > Drawing Preferences** dialog box.

### 1.1 Parameter Custom Strings

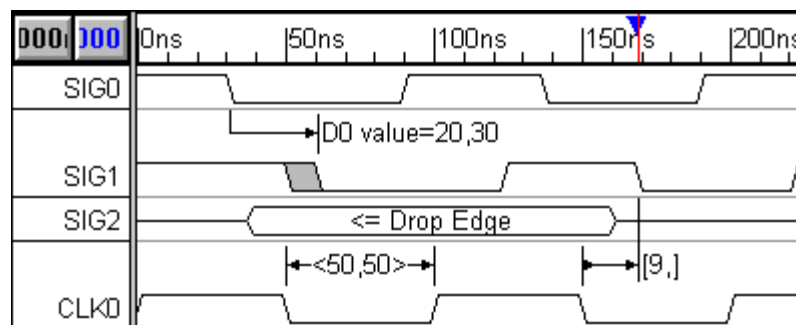
A parameter label can be made to show more than one attribute or to show a custom string of characters and attributes using the **Custom** string in the *Parameter Properties* dialog. In a custom string, certain character sequences are interpreted as attribute control codes, and when such a sequence is found it is replaced with that parameter's attribute.

Attribute control codes start with a % character followed by one or two letters. The control codes are: name (%n), value (%mv, %Mv), formula (%mf, %Mf), margin (%mm, %Mm), distance (%md, %Md), and comment (%c). The lower case m means minimum, and the upper case M means maximum. Now let's experiment with D0's custom string.

1. Double click on **D0** delay parameter to open the *Parameter Properties* dialog.
2. Select **Custom** from the **Display Label** drop-down list box. This will cause the string in the *Custom* edit box to be displayed as the parameters label.
3. Compare the default **Custom** string to the label that is displayed in the diagram. The default custom string is a little messy to look at, however it contains all of the control codes so you don't have to remember them. When you want to make a custom label just edit the default string. The default custom string should be:

```
%n v= %mv,%Mv f=%mf,%Mf m=%mm,%Mm d=%md,%Md %c
```

4. Next, make the parameter label display only the parameter name and min and max values. Edit the contents of the custom string so that the string reads: **%n value = %mv,%Mv**
5. Click the **Apply** button. The and D0's label will show:



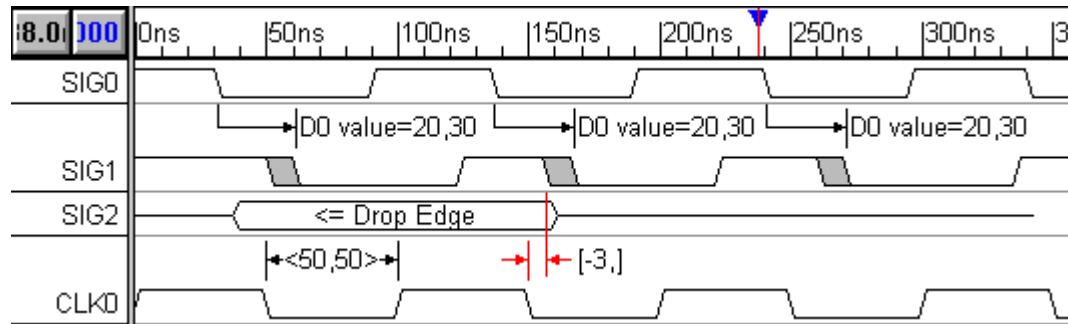
## (TD) 3.2 Repeating Parameters

Once you have drawn a delay, setup, or hold parameter, that parameter can be automatically drawn between similar edges across the timing diagram. When the Repeat button, in the *Parameter Properties* dialog, is pushed the program will search for the next beginning edge, and add a parameter between that edge and the next ending edge. This will continue until the end of the diagram. Some caution should be taken when repeating delays because the delays cause edges to move.

Repeat D0 across the timing diagram:

1. For this demonstration arrange *Diagram* window so that you can see the entire diagram. You may need to use the zoom in buttons.
2. In the diagram window, double-click on **D0** to open the *Parameter Properties* dialog.

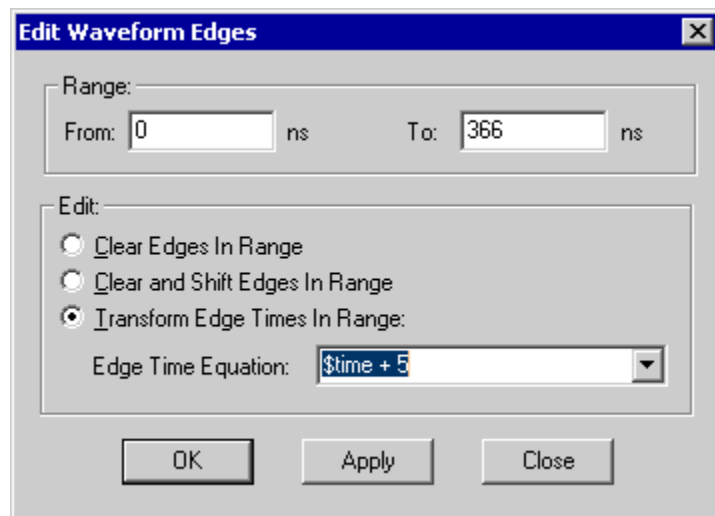
- Press the **Repeat** button. This will cause delays to be added to each of the falling edges of SIG0 that have a matching edge on SIG1. Also notice that the margin for setup S1 is now violated and is displayed in red. This happened because the second D0 moved the edge that S1 is attached to.
- Close the Parameter Properties dialog.



### (TD) 3.3 Editing Waveform Edges From an Equation

In the last section, our new delay caused the setup S1 to fail. To fix the setup, we would like to shift all of the edges on both SIG0 and SIG1 over by 5ns. This could be done by dragging and dropping each edge, but a faster way would be to apply an equation to the waveform edges.

- Select the **SIG0** and **SIG1** names by clicking on the signal names.
- Choose the **Edit > Edit Waveform Edges** menu to open a dialog of the same name that will act on all of the selected signals. Notice that the dialog can be setup to act on a range of edges, clear sections of the waveforms, or apply an equation to each edge of the waveform.

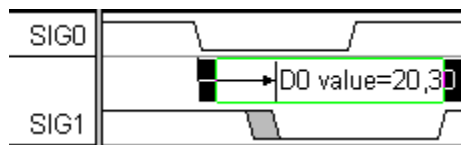


- Type **\$time + 5** into the **Edge Time Equation** edit box. The \$time variable represents the time of each edge.
- Press the **OK** button to apply the equation and close the dialog. Notice the edges have shifted over and the **S1** setup is satisfied.

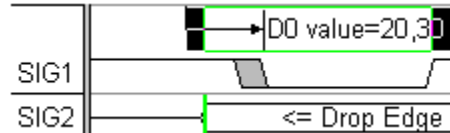
### (TD) 3.4 Drag and Drop Parameter End Points

When a parameter is created it is attached to two signal transitions. These signal transitions can be changed by dragging and dropping one of the parameter endpoints to a new signal transition. To demonstrate dragging and dropping a parameter's endpoint:

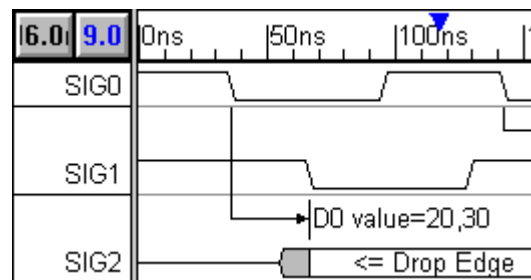
- In the *Diagram* window, select the first delay parameter **D0** to select it by clicking on it. A selected parameter is surrounded by a rectangle with a solid handle box on either end.



2. Place the mouse over the solid handle box on the right side of the selection rectangle.
3. Click and drag the mouse to the edge indicated on SIG2 so that it is highlighted. If the entire parameter is changing its vertical position then you clicked on the middle of the parameter instead of a handle box.



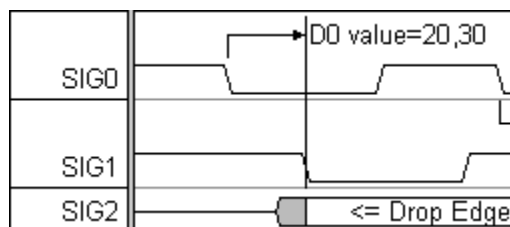
4. Release the mouse button. Now *D0* ends on this transition.



### (TD) 3.5 Adjusting the Vertical Placement of a Parameter

Normally, the vertical placement for parameters on the screen is set automatically. However, you can also place parameters at a specific height by dragging the parameter to a new position.

1. Click and hold on the center of the delay parameter, **D0**, and **drag** it up to a new vertical position closer to the top of the screen.
2. Release the mouse button to place the parameter.



After you move a parameter, it is considered user placed and it will not be moved from that position unless you choose to move it. Any new parameters will arrange themselves around user placed signals. To return vertical placement control to the program:

1. Open **D0's** *Parameter Properties* dialog box by double-clicking on the parameter.
2. Uncheck the **User Placed** box, and the delay will return to its original position.
3. Click the **OK** to close the dialog box.

### (TD) 3.6 Clock Jitter and Display

Clocks have many display and timing analysis settings that are covered in *Chapter 2: Clocks*. In this section we will add edge jitter and see the effect on the distance measurement. We will also add arrows to the falling edge of the clock and change the slant of the waveform edges.

The timing analysis features are controlled through the *Edit Clock Parameters* dialog:

1. Double click on waveform segment on CLK0 to open the *Edit Clock Parameters* dialog.
2. Type **4** into the **Rise Jitter (range)** edit box and tab to another control. This will add an uncertainty region to the rising edge of the clock and also change the distance measurement.
3. Click **OK** to close the dialog

The display features for signals and clocks are controlled through the *Signal Properties* dialog:

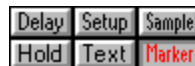
1. Double click on the **CLK0** signal name to open the *Signal Properties* dialog.
2. Check the **Falling Edge Sensitive** box and push the **Apply** button. This causes arrows to be added to the falling edge of the clock.
3. Press the **Analog Props** button to open the *Analog Properties* dialog.
4. Check the **Use Straight Edges** box and press **OK** to close the analog dialog. This will cause the clock to be drawn with straight edges instead of the normal slanted edges.
5. Press the **Grid Lines** button to open the *Grid Options* dialog.
6. Check the **Enable Grid** box and press the **Apply** button. This draws grid lines on the clock.
7. Play around with the grid options and make the grid draw on different edges. Also draw different color edges and line styles.
8. When you are done uncheck **Enable Grid** and close both dialogs.

## (TD) 3.7 Markers

Time markers (vertical lines) can be added to a timing diagram for documentation, time compression, and to indicate the end of the diagram. TestBench Pro also uses markers to specify loops and to insert HDL code into a transaction.

Next add a documentation marker the diagram and experiment with the display and time compression.

1. Press the **Marker** button, on the top of the *Diagram* window, to put the program in marker drawing mode.



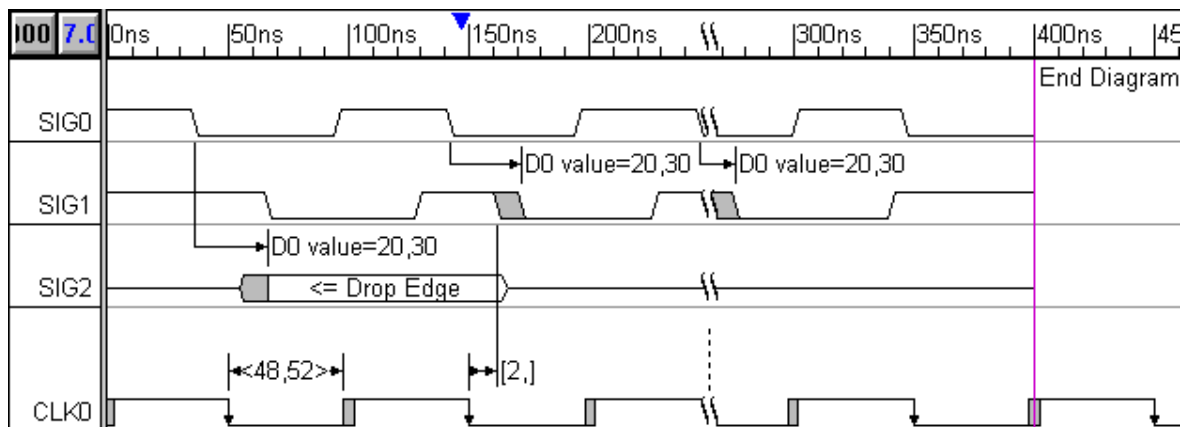
2. Left click on the third falling edge of CLK0 (250ns), to select it, and then right click to add a Marker.
3. Double click on the marker to open the *Edit Time Marker* dialog. Since an edge was selected when you added the marker it is automatically be attached to the selected edge, and the attachment is listed in the middle of the dialog as **EDGE CLK0 250**.
4. Uncheck the **Draw line from marker to edge** box. When marker is attached to an edge, this box determines if a dotted line will be drawn between the edge and the marker.
5. From the **Display Label** box, choose **Comment**. Since the comment for the marker is blank, no label will be displayed for the marker.
6. From the **Marker Type** box, choose **Timebreak(Curved)** to make the marker use a double curved line display.
7. Press **OK** to close the dialog. Notice that the marker is curved and does not display its label. Double click on the marker to open the *Edit Time Marker* dialog again.
8. Type **15** into the **Time Break compresses time by** box and press **OK** to close the dialog. Notice that 15ns of the next clock cycle is not displayed in the diagram. All the parameters inside a compressed region continue to function, just part of the diagram is not shown.



9. Drag and Drop the parameter and watch the compression marker make objects disappear.

A marker can also be used to indicate the end of a timing diagram. This is a useful feature if you are using the export scripts. You can also make the ends of all the signals snap to the marker for a cleaner looking timing diagram.

1. Make sure that no edges are selected in the diagram, and then right click at the top of the diagram at about 400ns. This will add a marker to the right of all the drawn signals.
2. Double Left click on the marker to open the *Edit Time Marker* dialog. Notice that the attachment is listed as **Time** because no edges were selected when the marker was added.
3. From the **Marker Type** box, choose **End Diagram** to indicate that the marker is the end of the diagram. This causes the marker to draw itself with the purple simulation line.
4. From the **Display Label** box, choose **Type** to make the marker display *End Diagram* as the display label.
5. Check the **Signal ends snap to marker** box and press **OK** to close the dialog. Notice that all of the drawn waveforms have drawn themselves over to the marker.
6. Drag and drop the end diagram marker and notice how the waveforms draw themselves.



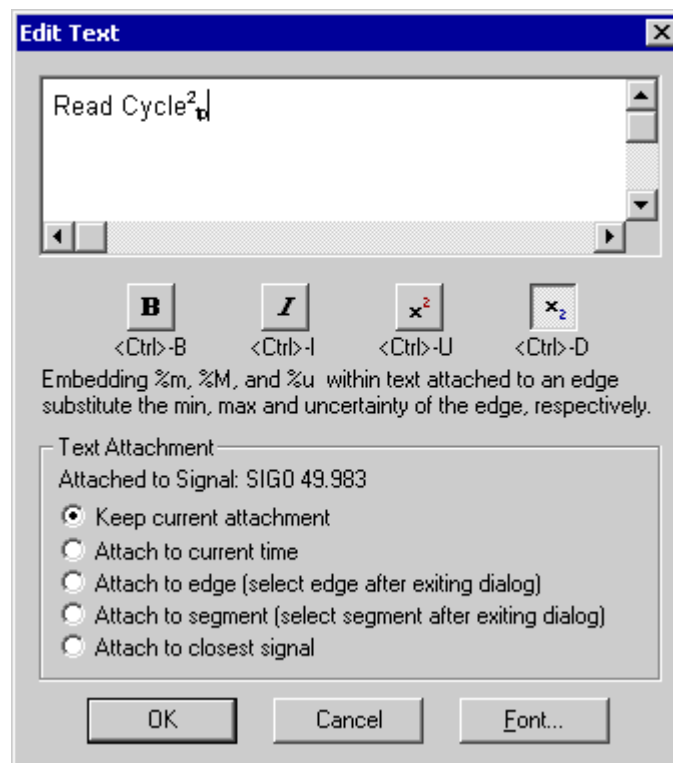
## (TD) 3.8 Edit Text Blocks

Text objects can be placed anywhere in a diagram to annotate cycles, edges, or segments. The font and color of each text object can be changed to stress the importance of that particular text object. The fonts also support superscripts, subscripts, and bold and italic attributes so your timing diagrams can use the same names and comments that are commonly used in data books

1. Press the **Text** button, on the top of the *Diagram* window, to put the program in text drawing mode.



2. At the top of the diagram, around 50ns, right click to open a text editing box and type **Read Cycle2tp**, and then press the **Enter** key to close the editing box. This will add a text block to the top of the diagram using the default font.
3. Double click on the text block to open the *Edit Text* dialog. From this dialog you can edit the text, add multi-line text blocks, and set the bold, italics, superscript, and subscript settings.



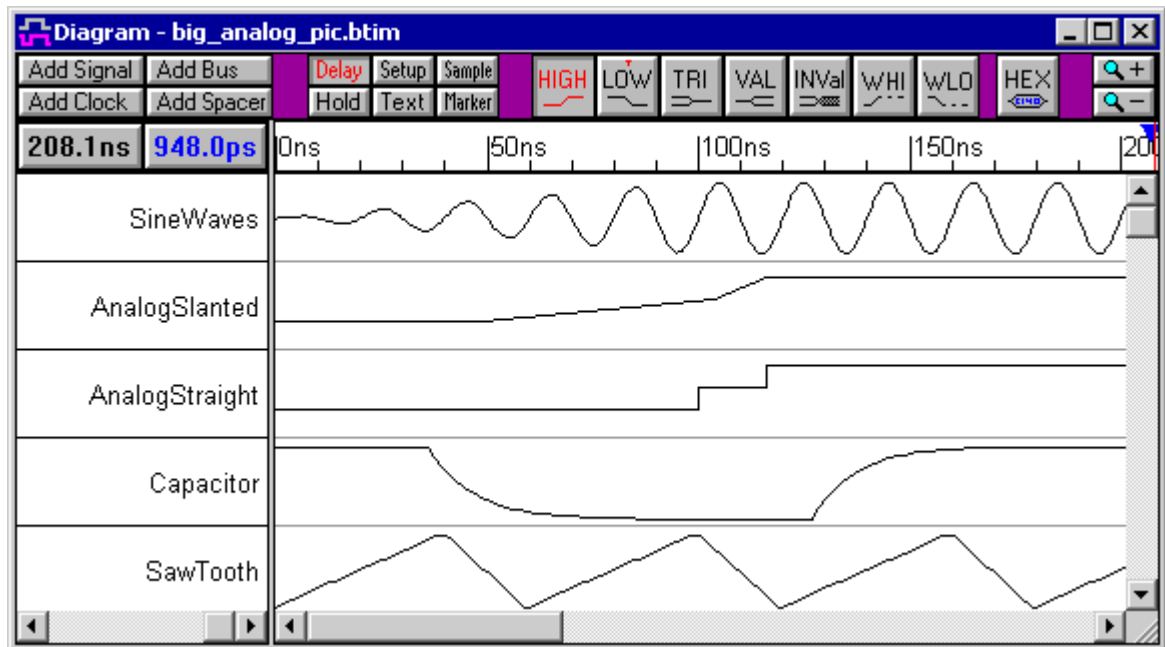
4. Select the **2** in the text box and press the super script button. Select **tp** and press the subscript button.
5. Press the **Font** key to open the *Font* dialog.
6. Change the font size to **16** and the color to **blue** and close both dialogs.
7. Drag and Drop the text object to a new location.
8. Experiment by adding more text blocks. In the *Edit Text* dialog, add a multiline text block.

### (TD) 3.9 Summary of Display and Documentation Tutorial

Congratulations! You have completed the *Display and Documentation* tutorial. In this tutorial you experimented with parameter display settings including how to add distance measurements and custom display strings. You have also touched on the some of the display options for markers, text objects, and clocks but these objects have many more features that are covered in the manual.

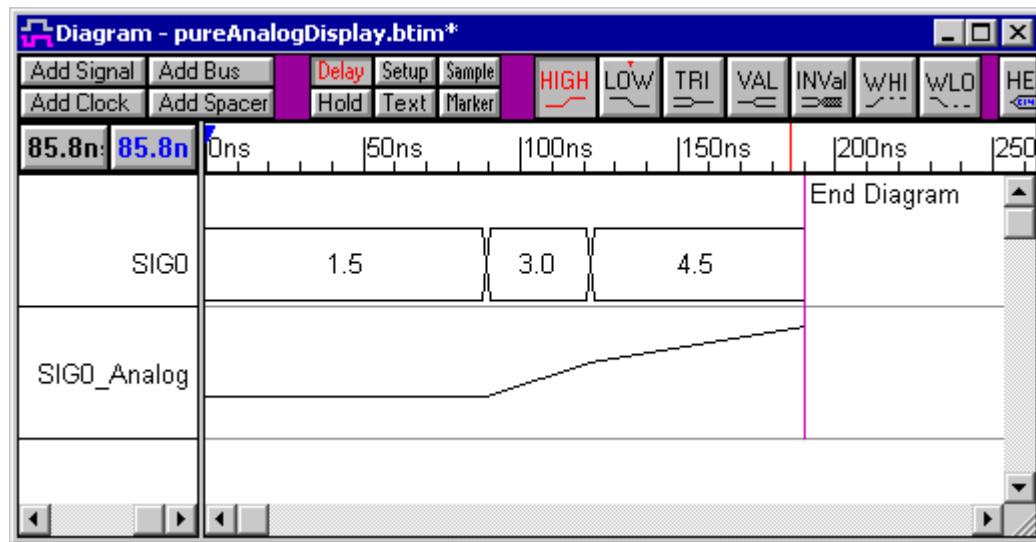
## Timing Diagram Editor 4: Analog Signals

This tutorial describes how to easily create and display analog waveforms by directly drawing them and using pre-written label equations.



### (TD) 4.1 Viewing & Exporting Real Radix Signals as Analog Waveforms

The three signals in the diagram below have the same waveform values, but the bottom two signals have their **Analog Display** checkboxes checked, so they draw as magnitude plots instead of as bus values. This checkbox is located in the *Signal Properties* dialog, which is opened by double-clicking on a signal's name. The second signal, **RealRadix\_AnalogSlanted**, shows the waveform using the analog values of each segment drawn from point to point (piecewise-linear). The third signal, **RealRadix\_AnalogStraight**, shows the waveform using the analog values of each segment drawn as step voltages.



**Analog Signal Display Methods (piecewise-linear or as step voltages):** Analog signals can be rendered as piecewise-linear (slanted) or as step voltages (straight). By default, analog signals draw piecewise-linear, as this makes for smoother curves. You can change the rendering method for an individual analog signal by setting the **Use Straight Edges** checkbox in the *Analog Properties* dialog (to open this dialog, press the **Analog Props** button in the *Signal Properties* dialog).

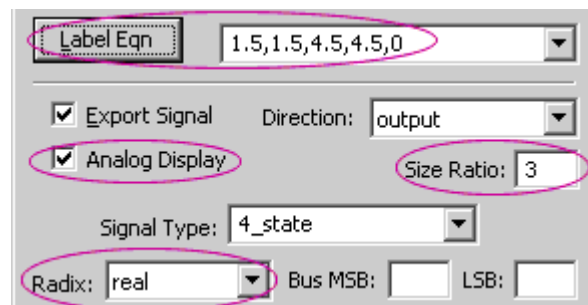
**Exporting Analog Signals To HDL Simulators:** When exporting analog signals to a discrete event simulator such as VHDL or Verilog, analog signals must be exported as step voltages (discrete event simulators cannot model a true ramp, for example, so ramps must be approximated by step voltages). So regardless of whether an analog signal is being rendered piecewise-linear or as step voltages in the timing diagram window, it will export as step voltages to HDL simulators.

**Exporting Analog Signals To SPICE:** When exporting analog signals to SPICE, the signals can be exported piecewise-linear or we can approximate step voltages depending on whether the "straight edges" option is unchecked or checked. Step voltages are approximated by adding an additional point in the spice PWL statement immediately after each drawn point. Note: digital signals always export to SPICE as quasi step voltages using PWL statements.

## (TD) 4.2 Creating Analog Signals using the Mouse

Analog signals can be created by drawing valid segments and inserting *real* values for the segment states. By inserting an extra segment for each value change, we will be able to render a "stepped" waveform even though we have not checked the **Use Straight Edges** checkbox.

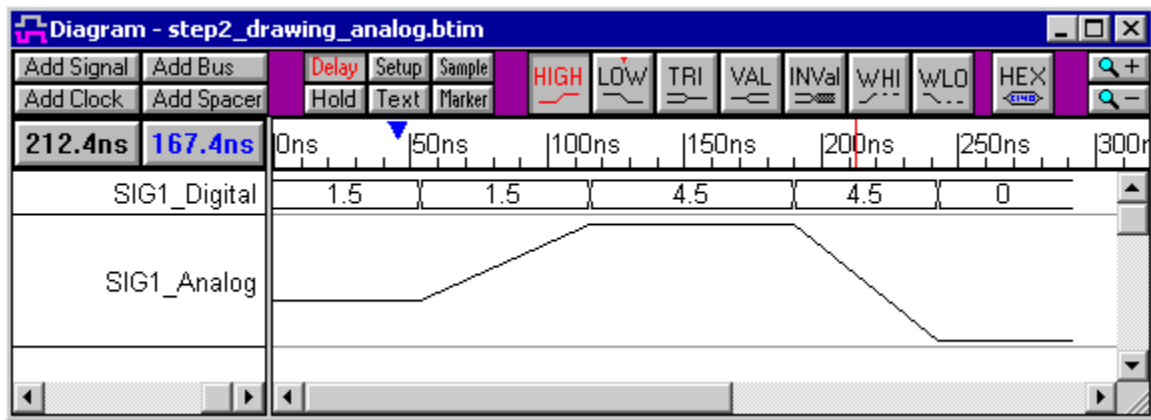
- Add a new signal and draw 5 valid segments.
- Double click to open the *Signals Properties* dialog.
- Set the **radix** to *real*.
- In the **Label Eqn** box, type **1.5, 1.5, 4.5, 4.5, 0** and then press the **Label Eqn** button to apply the equation and label the valid segments.



- Check the **Analog Display** check box, to switch from a digital display to an analog display.

- Set the **Size** to **3**, to expand the waveform.

The picture below shows what your signal looks like before and after the **Analog Display** is checked.



Reminder: although the signal above appears to have ramps in it, it will look like a stepped waveform when exported to a digital simulator. To create ramps that export to a digital simulator, use the **Ramp** function discussed later in this tutorial.

## (TD) 4.3 Drawing a Step Signal

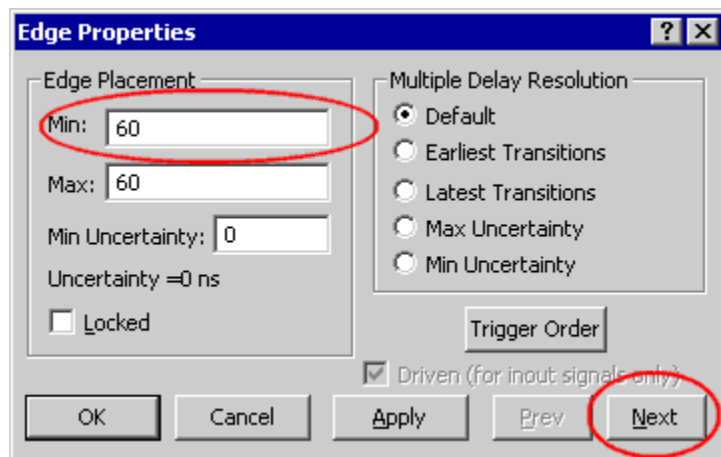
The ramp signal can now be easily converted to a step signal by adjusting the edge times for the segment that displays the change in value (The second and the fourth segments in our example). You can either use mouse to move one of the edges of the segment to the same time as the other or you can use the *Edge Properties* dialog to adjust the edge times.

### To use the mouse:

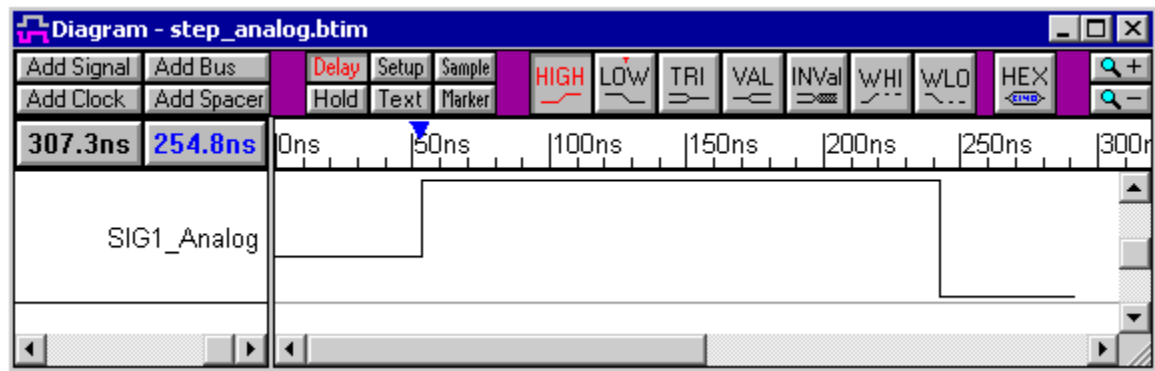
- Click the edge to move to select it.
- Drag the edge to the same time as the start or end edge of the segment. This shows that the value change is instantaneous (or not visible in the current time scale).
- Collapse all edges in this manner.

### To use the *Edge Properties* dialog:

- Double-click the first edge to open the edge properties dialog.
- Make note of the **Min** time.
- Click the **Next** button to move to the next edge.
- Enter the Min time from the previous edge in either the **Min** or **Max** edit box. Click Next to move to the next edge and apply the changes.



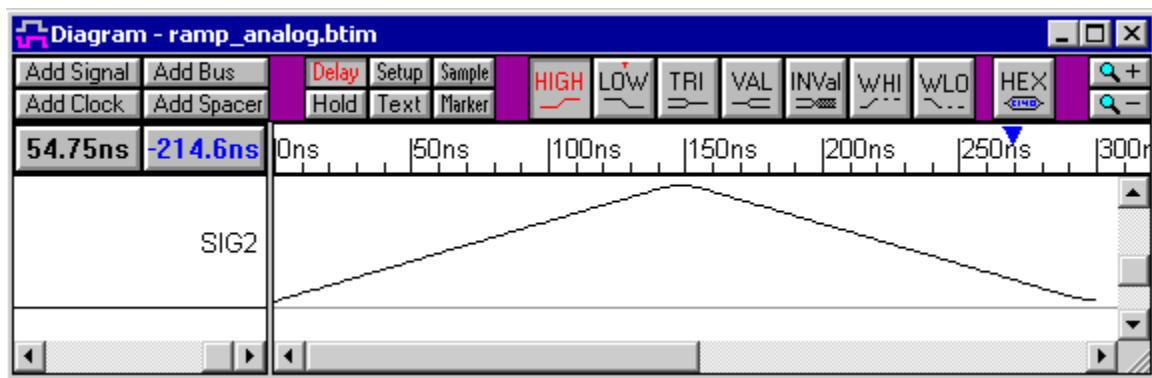
Once this process is completed for the transition segments the signal displays as a Step signal, such as below:



### (TD) 4.4 Drawing Analog Ramps

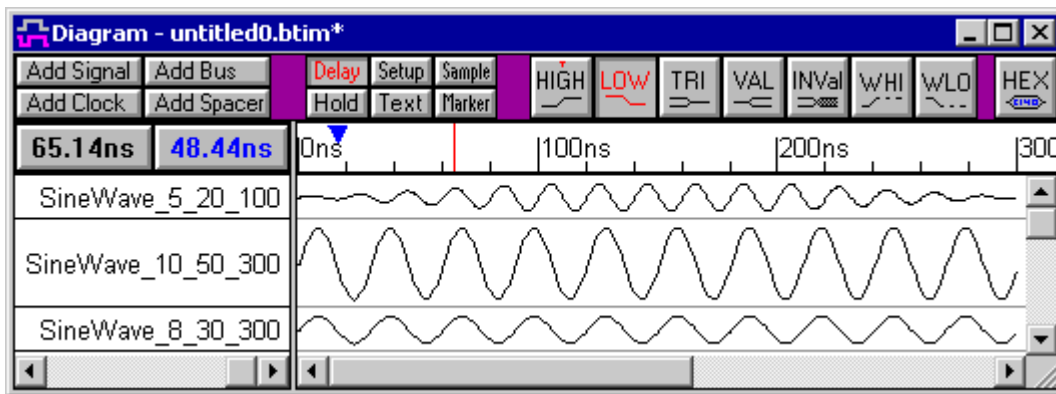
The best way to create a ramp signal is to use a **Ramp** label equation. This function can create a ramp signal with plenty of data points, so that it will export accurately to both analog and discrete-event digital simulators. The function is in the form of **Ramp(startVoltage, endVoltage, duration, numberOfPoints)**:

- Add a new signal to your diagram and double click on the signal name to open the *Signals Properties* dialog.
- Change the signal's **Size** to **3** so that the ramp will look nice.
- In the **Label Eqn** box, type **Ramp(0, 5, 150, 40)** and press the **Label Eqn** button. This will draw a ramp that starts at 0 volts and ramps up to 5 volts, over a period of 150 ns, using 40 points.
- Now add a down ramp by changing the equation to **Ramp(5, 0, 150, 40)** and pressing the **Label Eqn** button.



### (TD) 4.5 Generating Sine Waves

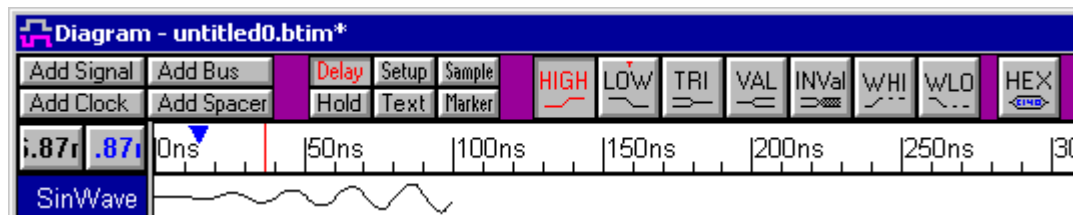
This step will describe how to use the Label Equation generator to create Sine waves on signals. There are three functions **SinStart**, **Sin**, and **SinEnd** that generate sine waves with a growing, steady, or decreasing amplitude. All of these functions take three parameters - **amplitude**, **period** and **duration**.



### (TD) 4.5.a The SinStart(amplitude, period, duration) Function

This function starts a Sine wave with amplitude of 0 and grows to the amplitude specified over the duration specified. The duration should be specified in Display Time Units.

- Add a new signal to the diagram named **SinWave**.
- Double click the signal name to open the *Signal Properties* dialog.
- Type **SinStart(5, 20, 100)** in the *Label Eqn* history box. This tells WaveFormer to start a Sine wave that will have amplitude of 5, a 20ns period and has reached its full amplitude by 100ns (assuming that ns is the display time unit).
- Click **Apply** to generate the start of a Sine wave. The waveform should look similar to this:



### (TD) 4.5.b The Sin(amplitude, period, duration) Function

This function draws a continuous Sine wave using the specified parameters. Here, we will append to the starting Sine wave drawn on the **SinWave** signal.

- Enter **Sin(5, 20, 100)** in the *Label Eqn* history list of the *Signal Properties* dialog for **SinWave**.
- Press the **Label Eqn** button to continue the Sine wave generation



Notice that by using the same parameters the generated Sine wave matches the Start Sine wave that was drawn.

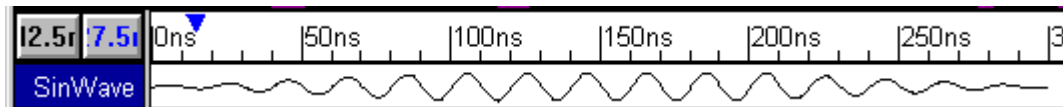
### (TD) 4.5.c The SinEnd(amplitude, period, duration) Function

The SinEnd function does the inverse of the SinStart function - it starts with a Sine wave with the amplitude specified and diminishes over the specified duration until it has amplitude zero by the end of the duration.

- Enter **SinEnd(5, 20, 100)** in the *Label Eqn* history list of the *Signal Properties* dialog for

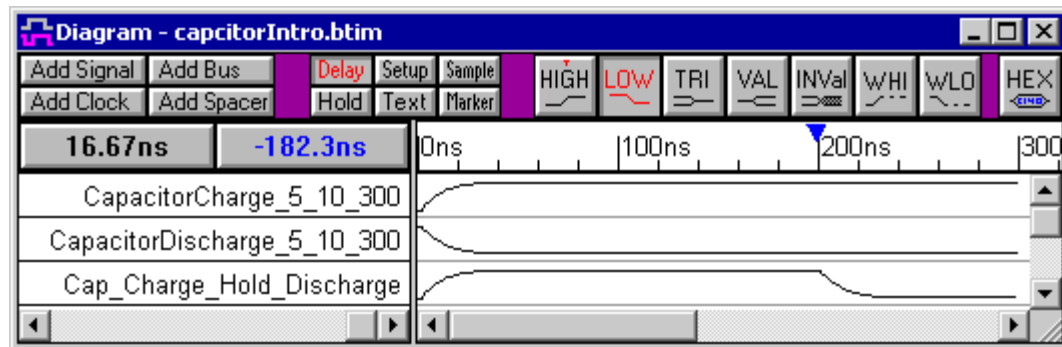
### SinWave.

- Click **Apply** or the **Label Eqn** button to finish our Sine wave generation



## (TD) 4.6 Generating Capacitor Charge and Discharge

The label equation generator also has two functions that are useful for drawing capacitor charge and discharge waveforms. These two functions have three required parameters - **amplitude**, **RC** and **duration** - that are used to generate the waveform.



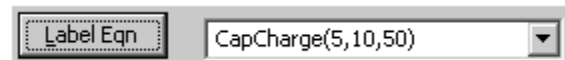
### 6.1 The CapCharge(amplitude, RC, duration) Function

This function generates the waveform for a charging capacitor based on the amplitude, RC and duration specified. The waveform will start at the lowest point specified by the amplitude and gradually rise (using the RC value) over the duration specified.

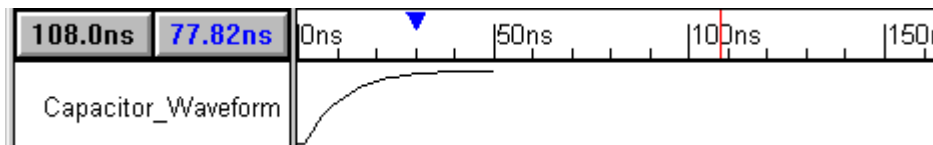
To generate a waveform for a charging capacitor:

- Add a new signal to the diagram named **Capacitor\_Waveform**.
- Double click the signal name to open the *Signal Properties* dialog.

- Type **CapCharge(5, 10, 50)** in the *Label Eqn* history box. This tells WaveFormer to generate a capacitor charge waveform that will have amplitude of 5, a 10ns period and has reached its charge by 50ns (assuming the ns is the display time unit).



- Click **Apply** to generate the start of a Sine wave. The waveform should look similar to this:



### 6.2 The CapDischarge(amplitude, RC, duration) Function

The **CapDischarge** function performs the inverse of the **CapCharge** function. The initial state that it generates is the peak charge specified by the amplitude and it slowly declines based on the RC

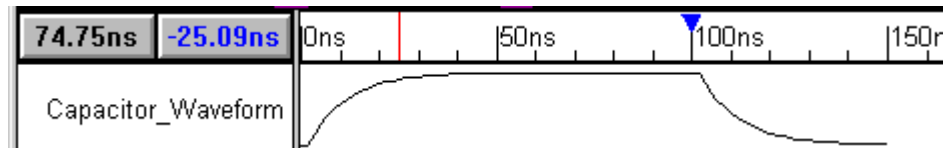


provided over the duration until it reaches the lowest point specified by the amplitude.

To generate a waveform for a discharging capacitor:

- Type **CapDischarge(5,10,50)** in the *Label Eqn* history box. This tells WaveFormer to generate a capacitor discharge waveform that has amplitude 5, an RC value of 10ns and a duration of 50ns (assuming that the display time unit is ns).
- Click the **Label Eqn** button to generate the waveform.

Note that in the image below a 50ns high segment was added to **Capacitor\_Waveform** prior to generating the discharge.



## (TD) 4.7 Converting between Real and Multi-bit Signals

Signals can be converted between real and digital multi-bit signals by changing the radix in the *Signal Properties* dialog. This is particularly important when waveform data comes in as either analog or digital, but the eventual use will be the opposite format.

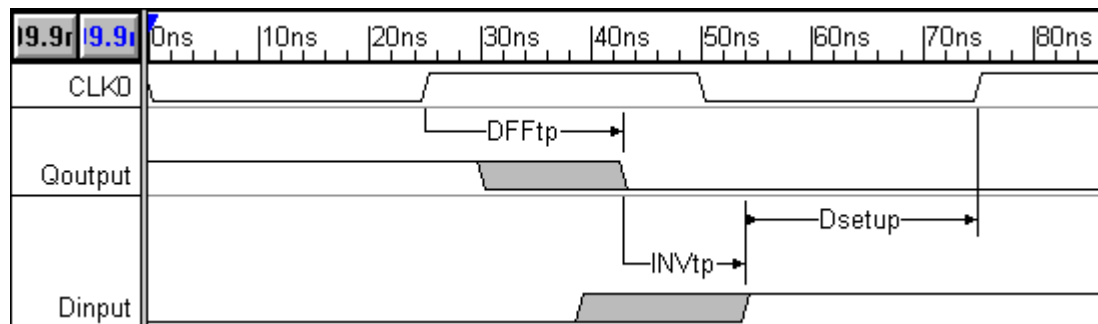
## Timing Diagram Editor 5: Parameter Libraries

This tutorial explains how to use the library functions of WaveFormer Pro, TestBench Pro, VeriLogger Pro and Timing Diagrammer Pro. It starts up where the basic tutorial ends. If you do not want to go through the first tutorial, a completed diagram of the first tutorial is on your disk under the filename tutorial.btim (can be loaded directly by clicking on the tutorial icon). A completed diagram of the Library tutorial is also on your disk under the filename tutlib.btim if you wish to check your diagram at the end of this tutorial.

### Getting Started

First, configure your program, and load the file **tutlib.btim**.

1. Minimize the Report window (and Project window if applicable). They are not used in this tutorial.
2. Select the **Window > Tile Horizontally** menu option. Both the *Diagram* and the *Parameter* windows should be visible during this tutorial. If you are unable to view one of the windows, use the **Window > Parameter** or **Window > Diagram** menu option to open the missing window.
3. Select the **File > Open Timing Diagram** menu option and load **tutlib.btim** from the **SynaptiCAD\Examples\TutorialFiles\ParameterLibraries** directory.
4. Select the **File > Save As** menu option and save the file as **mylib.btim** (this will keep the original file intact).
5. Click **Open** to load the file.

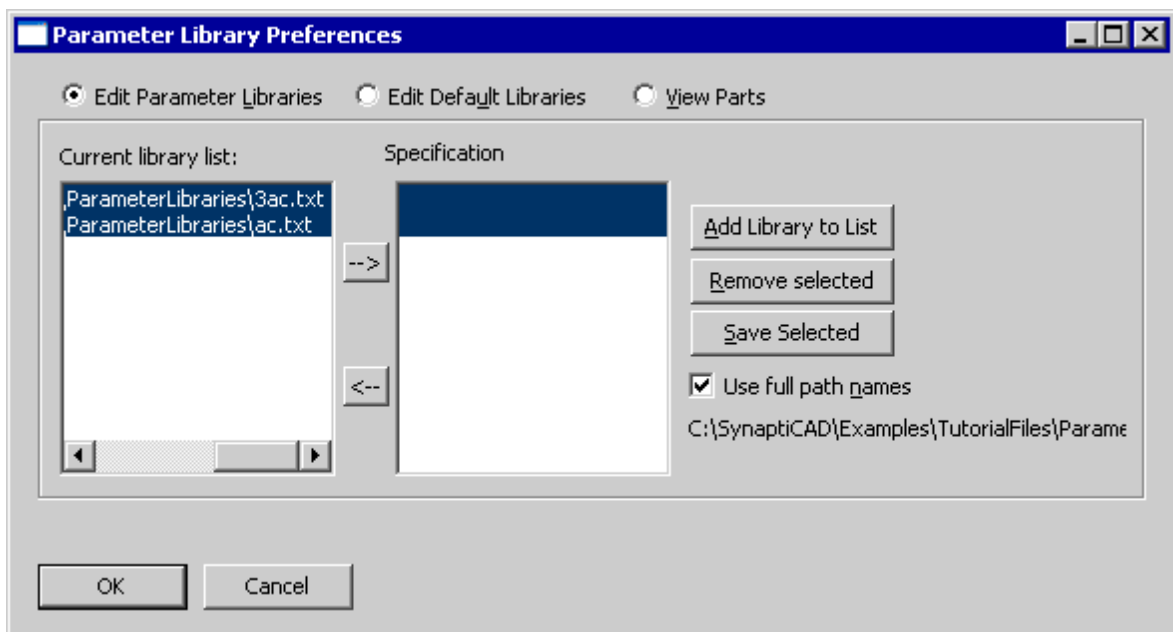


### (TD) 5.1 Adding Libraries to the Project's "Library Search List"

In order for a project to use a library, it must know the library's name and path location. This information is kept in the project's library search list.

To edit the library search list of the **mylib.btim** file:

1. Select the **ParameterLibs > Parameter Library Preferences** menu option. This opens the *Parameter Library Preferences* dialog.
2. Click the **Add Library to List** button to open the *Parameter Library Browse* dialog to search for libraries on your disk.
3. Select the two sample libraries **ac.txt** and **3ac.txt**, located in the **SynaptiCAD\Examples\TutorialFiles\ParameterLibraries** directory.
4. Click the **OK** button. This adds the selected files to your search list, and close the *Parameter Library Browse* dialog.



Now in the *Parameter Library Preferences* dialog, both libraries should be selected in the library search list path section. The next section also uses the *Parameter Library Preferences* dialog so leave it open.

**Note:** the filenames for the libraries will have their path names attached unless you have unchecked the "Use full path names" check box. Generally you will want to leave this option checked as this allows you to use libraries in multiple directories.

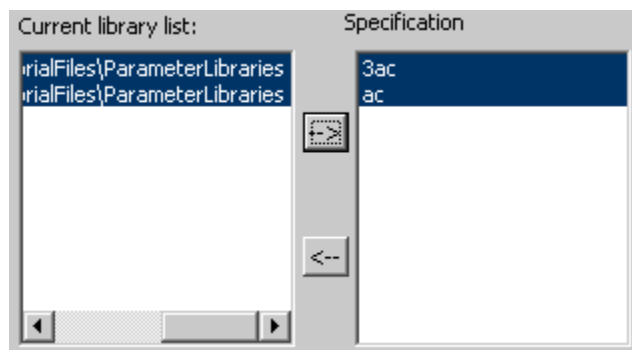
## (TD) 5.2 Setting Library Specifications

After adding libraries to the project's search list, you need to define the library specification. Library specifications allow SynaptiCAD products to distinguish between similarly named parts in different parameter libraries. Libraries 3ac.txt and ac.txt contain parameters with the same names. If you do not assign specifications and you referenced these parameter names in your design, the values from the first library in the list would be used.

To assign specifications:

1. Select both the **3ac.txt** and the **ac.txt** library files.
2. Click on the right arrow button in the *Parameter Library Preferences*. This will assign specifications to the selected libraries.

Now the specification for ac.txt is **ac** and the specification for 3ac.txt is **3ac**. To eliminate the specification for a library, select it and press the left arrow button.



## (TD) 5.3 Startup Library Configuration

Another useful feature of the *Parameter Library Preferences* dialog which we will not use in this tutorial is the **Edit Parameter Libraries** and the **Edit Default Libraries** radio buttons. The Edit Parameter Libraries radio button should currently be selected. This allows you to change the parameter library settings for the current project. If you have a set of libraries that you wish to use with all new projects, select the Edit Default Libraries radio button and add these libraries to the Startup library search list. These libraries will not be added to the current project, but any new project will automatically have these libraries included in their library search list.

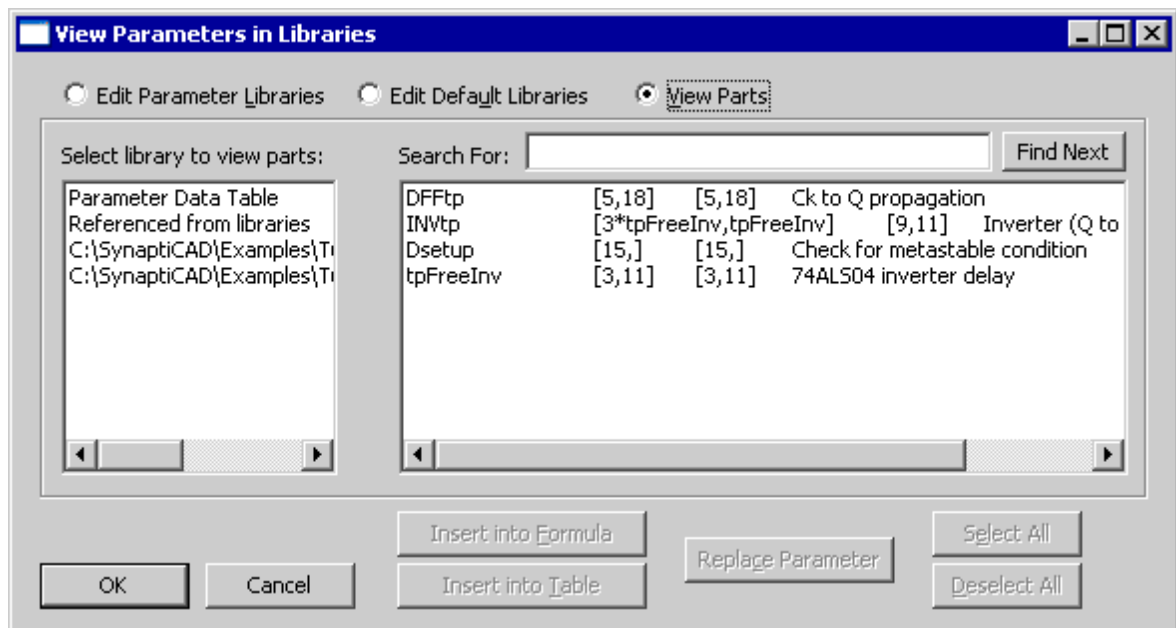
Close the Parameter Library Preferences dialog:

1. Make sure the **Edit Parameter Libraries** radio button is selected.
2. Click the **OK** button to close the *Parameter Library Preferences* dialog.

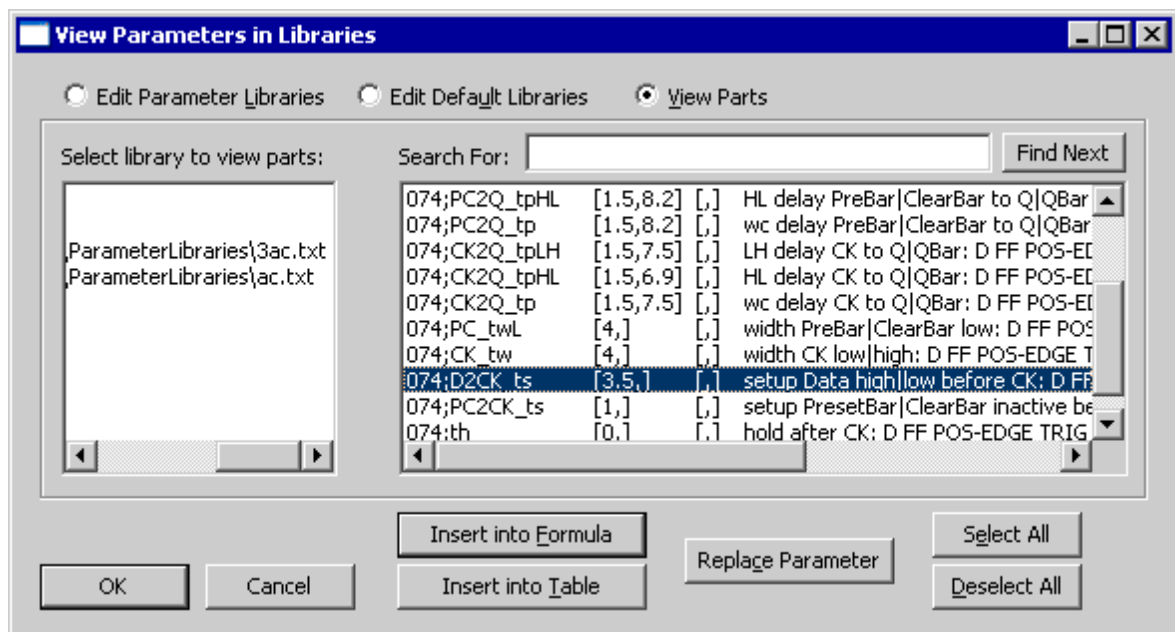
## (TD) 5.4 Referencing Parameters in Libraries

Now that we have added the libraries and set the specifications, we want to reference the library parameters in our project.

1. Double click on the **min value of the DSetup parameter** (in the parameter window) to edit the value. This opens the *Parameter Properties* dialog box.
2. Delete the value in the **min** edit box, then press the **Library** button to open the *View Parameters in Libraries* dialog.
  - Notice that there are **three** libraries on the library list; the 3ac.txt and Ac.txt that you added, and one called Parameter Data Table. This extra library is a virtual library that lists all the user-added parameters in the project. You can use virtual library parameters in formulas just like regular library parameters.



3. Select the **Ac.txt** library from the library list. This displays the parameters in this library in the library parts list on the right.
4. Scroll down in the library parts list to find the parameter **074;D2CK\_ts**. Left click to select the parameter, and press the **Insert Into Formula** button.



5. Click **OK** to close the *View Parameters in Libraries* dialog.

You are still in edit mode in the formula edit box, but now it should contain the name of the parameter we just inserted (Note: the library specification "Ac" is added to the parameter name, separated name by a colon, i.e. **+ac:074;D2CK\_ts**).

Next, we will edit both the min and max value of the delay **INVtp** at the same time.

1. Double click on the **INVtp** parameter (in the *Parameter* window) to update the *Parameter Properties* dialog display with the values for INVtp.
2. Click on the Library button to open the *View Parameters in Libraries* dialog.
3. Select the **ac.txt** library and insert the parameter **004;tp** into min value. (Select the parameter and press the **Insert Into Formula** button).
4. Choose the **OK** button to close the *View Parameters in Libraries* dialog.
  - Notice that the **ac:004;tp** parameter was added to the values that were already in the min and max edit boxes.
5. Delete the original values from the min and max edit boxes, leaving only the **ac:004;tp** value.
6. Click the **OK** button to close the *Parameter Properties* dialog.

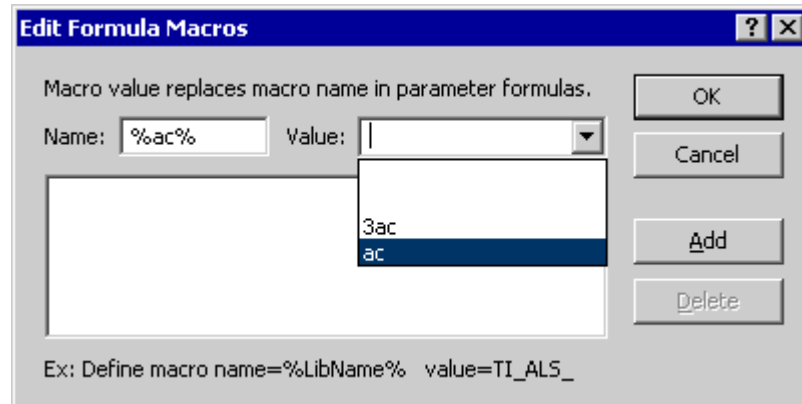
Repeat the above process for the min and max values of DFFtp, inserting **074;CK2Q\_tp** from the ac.txt library. Try using the **Search For** edit box in the *View Parameters in Libraries* dialog, instead of scrolling, to find a parameter name.

## (TD) 5.5 Using Macros to Examine Tradeoffs Between Different Libraries

Your diagram is now using values for the AC logic family operating at 5V. If you want to examine the impact of changing your design to 3.3V, you need to change the library specifications of the parameters to "3ac". It can get tedious changing back and forth between different libraries when you have to change the name of each parameter. To avoid this you can create a macro which you use in place of the library specification in your parameter names. Then to change libraries you just need to change the value of the macro.

To create a macro:

1. Select the **ParameterLibs > Macro Substitution List** menu option to open the *Edit Formula Macros* dialog.
2. Enter **%ac%** into the name edit box.
3. Select **ac** from the **Value** drop down box. The drop down box contains all libraries that have specifications.



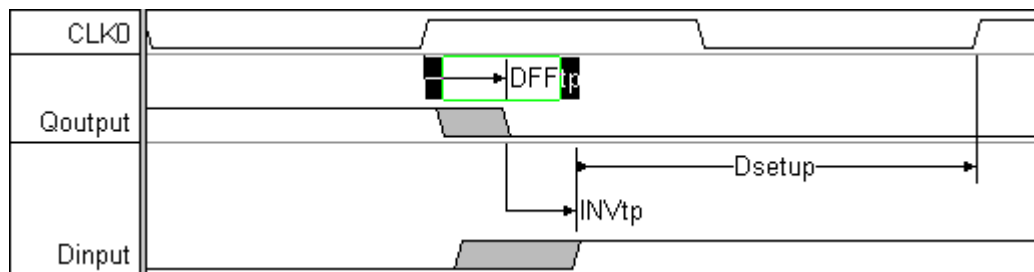
4. Click **OK** to add the macro to your macro list.

Now edit the five min & max values of your parameters, replacing **ac** with **%ac%**. Your design should still be using the 5V AC values. When editing the values, try using the Next and Previous buttons in the *Parameter Properties* dialog to move between parameters.

To make your design reference the 3V library, change the value of the macro.

1. Select the **ParameterLibs > Macro Substitution List** menu option, to open the *Edit Formula Macros* dialog.
2. Click on the macro **%ac%** in the list box. This places this macro into the Name/Value edit boxes.
3. Use the **Value** drop down box to change the value of the macro to **3ac**. Click **OK** to close the dialog.

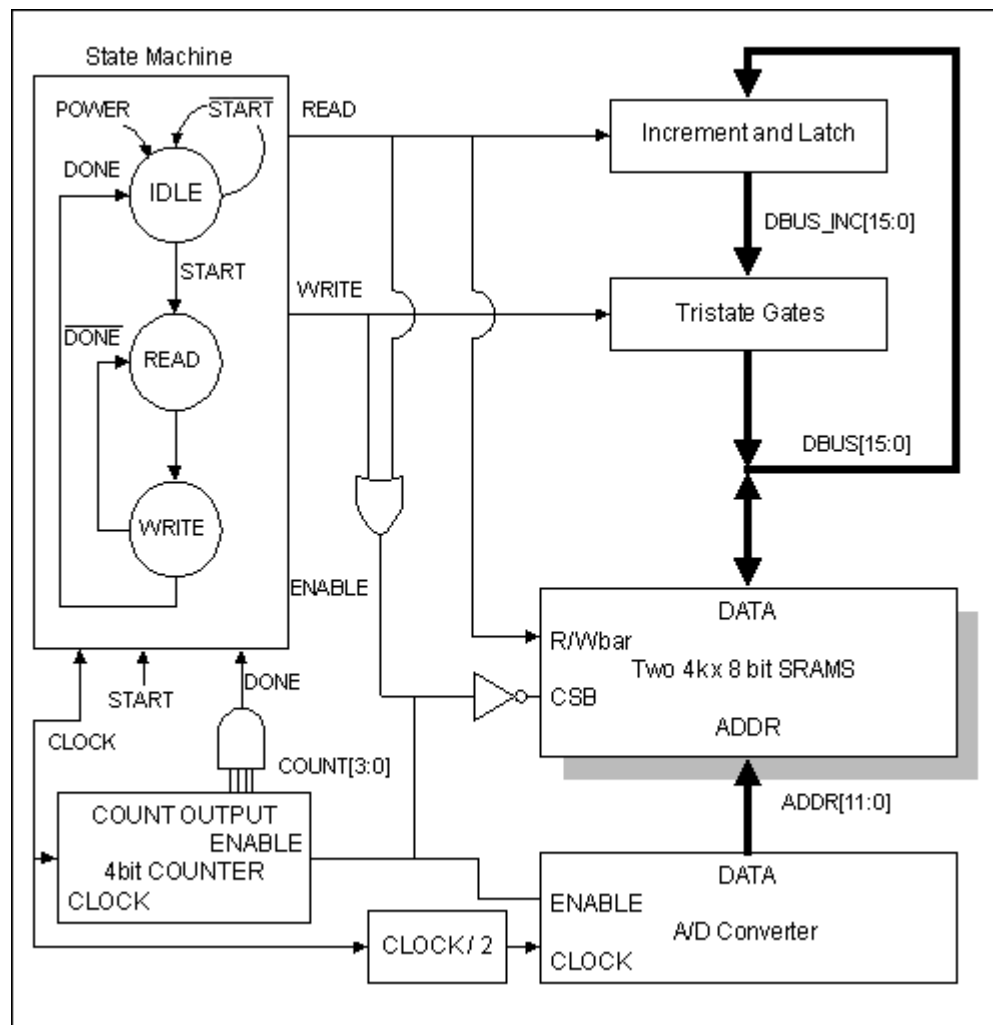
Your design should now be using the 3V AC values (the delays should be longer due to the decreased supply voltage). You have now completed the parameter library tutorial.



**Note:** Macros can also be used to make short or alternative names for library parameters without having to edit the library names.

## Timing Diagram Editor 6: Advanced Modeling and Simulation

This tutorial demonstrates how WaveFormer Pro can quickly model and simulate a digital system of moderate complexity. We will be modeling a circuit that computes histograms for 64K of data generated by a 12-bit Analog-To-Digital converter (this is a popular method for testing dynamic SNR for ADCs). This circuit is a simplified form of a real VME board that would take several months to model and simulate using conventional EDA tools. Using WaveFormer, we will model and simulate this simplified circuit in 20 minutes. The full circuit with the complete VME bus interface protocol could be modeled and debugged in about 4 hours.



**Figure 1: Histogram circuit block diagram.**

This tutorial teaches the user how to:

1. Model state machines using the Boolean Equation interface.
2. Generate input signals using temporal and label equations.
3. Use the simulation log to find design entry errors.
4. Simulate incrementally by temporarily modeling outputs as drawn inputs.
5. Enter direct HDL code for simulated signals.

6. Use external HDL source code models.
7. Model tri-state gates using the conditional operator.
8. Model n-bit gates using reduction operators.
9. Model transparent latches.
10. Debug Verilog source code using \$display statements.
11. Control length of simulation time using a Time Marker.
12. Edit an external HDL file with WaveFormer's Report window.

Before you begin the tutorial you may wish to view Figure 3 in Section 13 which shows a completed version of the diagram that we will generate. File `tutsim.btim` included in the product directory is a finished tutorial file. You will not use this file during the tutorial itself, but you can always refer back to this file if you encounter any problems during the tutorial.

### Circuit Operation

A histogram is a graph displaying the count of same 12-bit values received from the ADC. To store the histogram count values we will use a 4K SRAM (2<sup>12</sup> storage cells) to hold a count for each possible 12-bit value that the ADC can generate. The width of the SRAM depends on how many data values we will accumulate from the ADC. In the worst case, the ADC could generate the same value for the entire histogram accumulation, so the SRAM must be able to store a value of up to 4K. Thus we will use 2 8-bit wide SRAMs (2<sup>16</sup> = 64K > 4K).

When the circuit starts operation, the SRAM should contain zeros at every address. Each time a data value is generated by the ADC, that data value is used as an address to look up the current count for the data value in the SRAM. The count is incremented by one and the new value is written back to the SRAM. This continues until the circuit has r

## (TD) 6.1 Set up a New Timing Diagram

Create a new timing diagram to model the histogram circuit:

1. Select the **File > New Timing Diagram** menu option to create a new diagram.
2. **Minimize** the *Parameter* window. It is not used in this tutorial.
3. Select the **Window > Tile Horizontally** menu option. This will provide us with optimal viewing by rearranging the *Diagram* window and the *Report* window (if either of these windows is not visible, select the menu option **Window > Diagram** or **Window > Report** to make it visible).


Now that we have a new diagram to work with, we are ready to model the components of our circuit.

## (TD) 6.2 Generate the Clock, Draw Waveforms, & Use Waveform Equations

The histogram circuit has a system clock, CLK0, and three signal inputs, POWER, START and ADDR. We will create the waveforms for each of these signals using three different methods: generating from clock parameters, drawing waveforms by hand, and automatically generating waveforms from temporal equations.

### 2.1 Automatically generate the CLK0 system clock

Add a clock named CLK0 with a period of 100 ns:


1. Click the **Add Clock** button  to open the *Edit Clock Parameters* dialog.
2. Verify that the default values are: **name = CLK0**, **period = 100 ns**, and **duty = 50%**. If not then make the necessary adjustments.

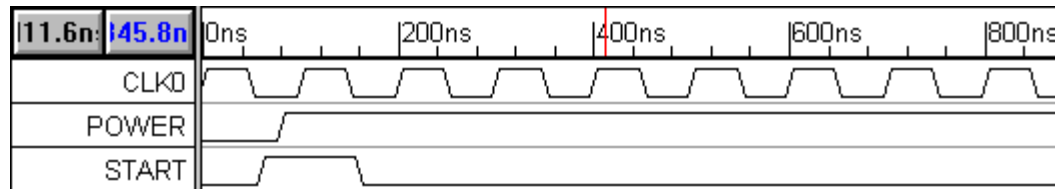


3. Press **OK** to accept the default values for the clock.

## 2.2 Graphically draw the POWER and START signal

The **POWER** signal is a power-on reset signal that we will use to set the initial state of our state machine. The **START** signal is an external input to the system that pulses high to initiate acquisition in the histogram circuit. The **POWER** and **START** waveforms are relatively simple, so we will draw them with the mouse.

1. Click on the **Add Signal** button  twice to add two signals.
2. Double-click on a signal name to open the *Signal Properties* dialog. Use this dialog to change the names of the signals to **POWER** and **START**.
3. Draw the **POWER** signal so that it is low for 80ns, then high for 2000ns.
4. Draw the **START** signal so that it is low for 60ns, high for 100ns, and then low for 800ns:
5. Verify that the timing diagram looks like:



Waveform drawing and editing techniques can be found in *Chapter 1: Signals and Waveforms* in the online help.

## 2.3 Use Temporal and Label Equations to model ADDR (A/D converter's output data)

We will model the A/D converter just as a data source, so all we need to do is generate a *virtual* bus signal called **ADDR** (the output from the ADC) that drives the address lines of the SRAMs. The **ADDR** waveform has a regular pattern that can be described easily using an equation, but would be tedious to draw by hand.

Add a virtual bus signal called **ADDR**:

1. Add a signal and change the name to **ADDR**. Leave the *Signal Properties* dialog open for the rest of the section.
2. Set the signal's **Radix** to **hex** and the **MSB** to **11**. Changing the MSB and Radix defines **ADDR** as a 12-bit signal that display its values in hexadecimal format.

Radix:	hex	Bus MSB:	11	LSB:	0
--------	-----	----------	----	------	---

The A/D converter is driven by a clock that is 1/2 the frequency of the state machine clock **CLK0**, so the **ADDR** value should change every other clock cycle (this maintains the same address for the read out of each RAM cell's count data and its write back after it is incremented). The **ADDR** signal should be **unknown for 170ns** then it should have **twenty valid states, each 200ns** in duration. Use the **Waveform Equation** interface of the *Signal Properties* dialog to generate the **ADDR** waveform:

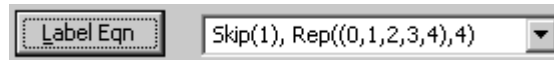
1. Enter the following equation into the edit box next to the **Wfm Eqn** button: **170=X (200=V)\*20**

Wfm Eqn	170=X (200=V)*20
---------	------------------

2. Press the **Wfm Eqn** button to apply the waveform equation. Notice that the waveform drew itself. If the waveform didn't draw, a syntax error was made when typing in the equation. To determine what the error was, look at the file `waveperl.log` displayed in the Report window. This file will show you which part of the equation could not be parsed. Fix the error, and press the **Wfm Eqn** button again.

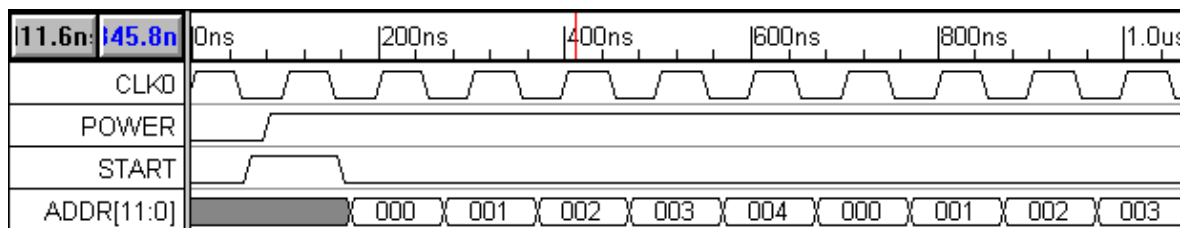
Next, we will label the states of the *ADDR* bus using a **Label Equation**. Each state could be labeled individually using the extended state field of the HEX dialog box, but labeling twenty states would take a long time. Instead, we will write an equation to label all the states at once. *Chapter 11* covers all the different state labeling functions.

1. Enter the following equation into the edit box next to the **Label Eqn** button **Skip(1), Rep((0,1,2,3,4), 4)**.



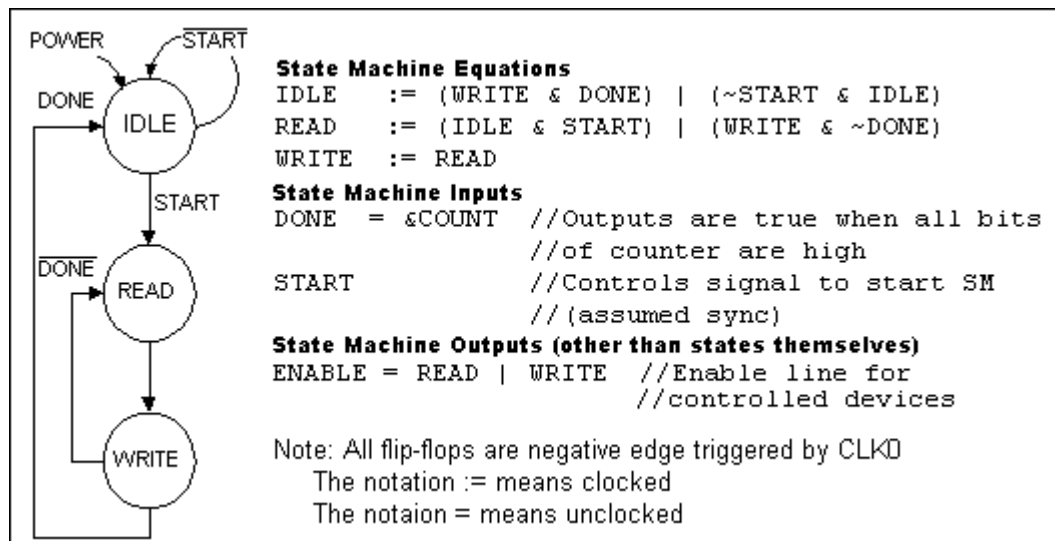
2. Press the **Label Eqn** button to apply the equation.

This equation will generate a hex count from 0 to 4, and then repeat it 4 times. The **Skip(1)** means start labeling after the first state (which we defined to be an invalid state using our waveform equation). Your timing diagram (at the appropriate zoom level) should now resemble the diagram below.



## (TD) 6.3 Modeling State Machines

We will use a simple one-hot state machine to control the circuit, and we will model it using Boolean Equations. A one-hot state machine uses a single flip-flop for each state. At any given time, only the flip-flop representing the current state will contain a 1, the other flip-flops will be at 0 (hence the name one-hot).



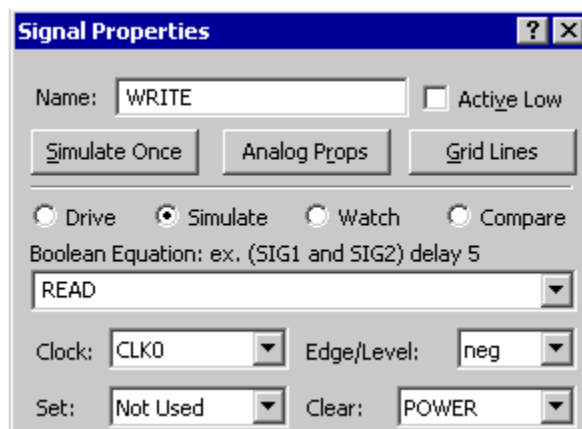
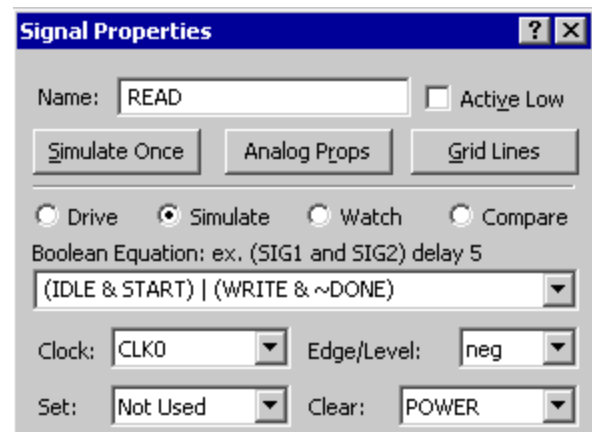
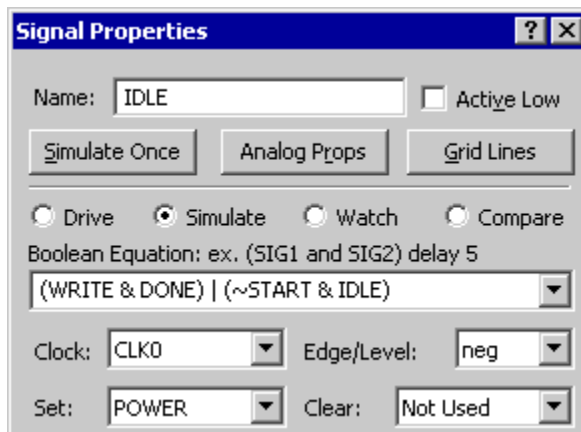
**Figure 2: State diagram and design equations for the histogram controller state machine**

The state machine (SM) initializes to the **IDLE** state. On the negative edge of the clock after **START** goes high, the SM will enter the **READ** state and look up the current count for the current address value being output by the A/D converter. This value will be incremented by a simple fast-increment circuit. On the next clock, the SM will enter the **WRITE** state, latching the

incremented value into a transparent latch called DBUS\_INC and initiating the write back of the incremented data to the SRAM. The state machine will continue to toggle between the **READ** and **WRITE** state until the desired number of data values have been histogrammed (determined by the size of the binary counter called COUNT), at which point the SM will return to the **IDLE** state. Figure 2 shows the SM that we will model.

The state machine is modeled in WaveFormer using one signal for each state. Next we will enter the equations for the state machine, however these signals are **not simulated until Section 5** because signal **DONE** has not yet been defined.

1. Add 3 signals and name them **IDLE**, **READ** and **WRITE**.
2. For each signal, enter state machine Equation, select Simulate button, setup the clock and trigger edge, and setup the set and clear signals as shown in the following pictures:



Notice the display **Compile Error** in the bottom right hand corner and notice that the state machine signal names turned gray. This is because the IDLE and READ equations reference a signal called DONE. This signal has not been defined so if you try to simulate you get errors. In the next section we will investigate the different ways to detect and fix simulation errors.

## (TD) 6.4 Checking for Simulation Errors

If you check the simulator log file, **simulation.log**  in the *Report* window, you will

see an error message reporting that DONE is not declared. The log file also reports the lines in the WaveFormer-generated Verilog source code file where this error occurred. The WaveFormer-generated source file will have the same filename as your diagram, but with a file extension of **.v** instead of **.btim** (so if your diagram is untitled.btim, the source code file is untitled.v). This source file is automatically opened by the *Report* window whenever WaveFormer Pro generates this file (by default this occurs every time you make a change to your design while simulating signals).

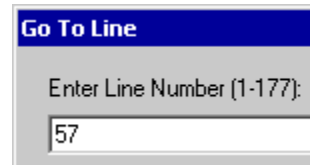
View the HDL lines where the errors occur:

1. Check the log file for the line number at which the error(s) occurred. In the *Report* window, click on the **simulation.log** tab. When we ran the simulator, our error occurred at line number 57 (your run may be different), as indicated by the error message:

**C:\SynaptiCAD\UNTITLED.v: L57: error: 'DONE' not declared**

2. Click on the tab for the \*.v file at the bottom of the Report window. This will open your source file in the *Report* window.

3. Click inside the *Report* window, and press **<Ctrl>-G**. This brings up the *Go To Line* window. Enter 57 as the line number you wish to jump to, and press **OK**.



4. As expected, these lines show the HDL code that simulates the IDLE and READ signals.

NOTE: Do not make changes in this source file as your changes will automatically be overwritten the next time a simulation is performed; instead, we will make the appropriate changes in the *Diagram* window and *Signal Properties* dialog.

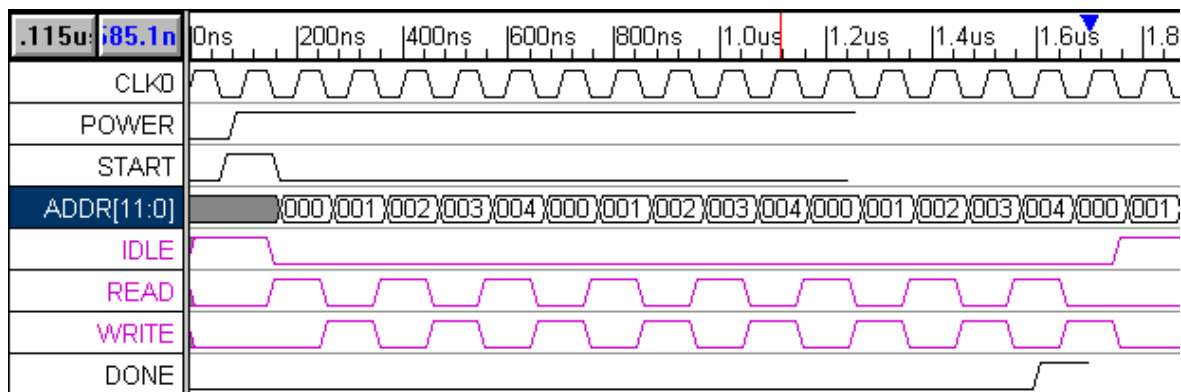
## (TD) 6.5 Incremental Simulation

One common problem in simulating and debugging digital systems is that large parts of the design have to be entered before testing can begin because the parts provide input to each other. One solution is to break a design up into pieces and test each piece with test vectors that represent the output of the other pieces. However, generation of the test vectors can be time consuming.

SynaptiCAD products provide a very simple and quick method for testing small parts of a design: graphically draw the signals for the missing parts of the design to test the design at its current state of development. Then later add the design information that models these signals (in other words, we temporarily model simulated outputs as drawn inputs).

We will now use this method below to verify the operation of our state machine before we enter the HDL code that generates the DONE signal:

1. Add a signal called **DONE**.
2. **Draw** a **low** segment for 1.6 us, followed by high pulse that lasts for at least one clock cycle. Click on **Apply** to run the simulation.
3. The diagram should now show the simulation output from your state machine. The simulated signals are pink to distinguish them from graphically drawn signals.



Make sure everything is working properly:

1. First make sure that the simulation status indicators read **Simulation Good** Simulation Good. If the indicators still show an error, then the **simulation.log** file will help you to pinpoint the error in your diagram.
2. Next, check your diagram against the figure above to verify that your state machine is simulating correctly.
3. If the simulation succeeded and there are still discrepancies in the output, check your design equations and the input stimulus you've drawn (START and DONE signals).

Once you have the circuit simulating properly, let's see what happens if the *START* pulse gets too small:

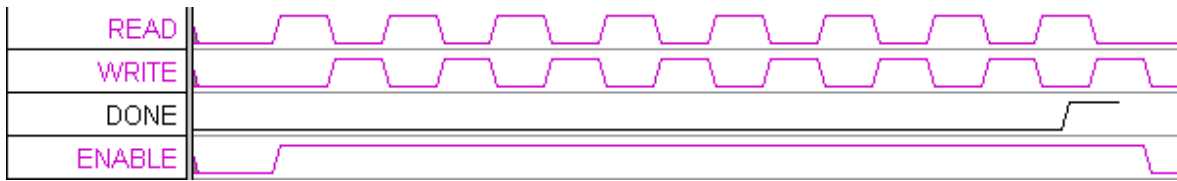
1. **Drag** the falling edge of the *START* pulse back to approximate **140 ns** (before the falling clock edge at 150 ns). This step causes the state machine to stay in the IDLE state (the IDLE signal stays high).
2. **Double click** on the falling *START* edge and enter a time of **160** into the *Edge Properties* dialog to restore proper operation.

## (TD) 6.6 Modeling Combinational Logic

In addition to the state signals, the state machine has one other output signal called *ENABLE* that is used to enable the SRAM, the *DONE* counter, and the ADC. *ENABLE* is just the output of an OR gate with the *READ* and *WRITE* signals as inputs. In Section 3 we used the Boolean Equation interface to model the flip-flops of the state machine. We will use the same interface to model combinatorial logic. To do this choose the default clock called *unclocked*. If a signal other than *unclocked* is selected, then the Boolean Equation interface models registers or latches depending on the type of Edge/Level trigger selected. Chapter 12 covers the advanced features of the Boolean Equation interface including the min/max delay features.

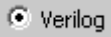
### *Model the Enable logic:*

1. Create a new signal called **ENABLE**.
2. Enter the equation: `READ | WRITE` into the **Boolean Equation** edit box in the *Signal Properties* dialog.
3. Check the **Simulate** radio button.
4. Verify that *ENABLE* is the **OR** of *READ* and *WRITE*. If *ENABLE* did not simulate, use the techniques found in section 4 to find your error. Remember that signal names are case-sensitive.
5. Click **OK** to close the dialog.



## (TD) 6.7 Entering Direct HDL Code for Simulated Signals

For simplicity, the counter output **COUNT** is modeled using a simple block of behavioral HDL Code instead of using Boolean equations. It would take a large number of Boolean equations to model the counter and the equations would be difficult to modify if the counter operation had to be changed. For this tutorial we will create a 4-bit counter to test our system. This counter could be easily modified later to make it 12-bit (to acquire 4K worth of data). To enter direct HDL code for the **COUNT** signal:

1. Create a signal called **COUNT**.
2. In the *Signal Properties* dialog, set the Radix to **hex**, its MSB to **3**, and check the **Simulate** radio button.
3. Press the **Verilog** radio button  to switch from the Equation view to the HDL Code view/editor.
4. **Enter** the Verilog code below in the HDL Code editor of the *Signal Properties* dialog (comments begin with `//` and can be skipped during code entry). You can copy and paste the text into WaveFormer instead of typing it (Select and copy to clipboard the source code below, then click into the HDL Code window in WaveFormer and press **<Ctrl>-V** to paste the text):

```
reg [3:0] COUNTER;    //declare a 4-bit register called COUNTER
always @(negedge CLK0)    //on each falling edge of CLK0
begin
    if (ENABLE)
        COUNTER = COUNTER + 1;    // count while ENABLE is high
    else
        COUNTER = 0;    // synchronous reset if ENABLE is low
end
assign COUNT = COUNTER;    //drive wire COUNT with reg COUNTER value
```

5. Click the **Simulate Once** button to simulate the **COUNT** signal.

Note: All signals in WaveFormer are modeled as wires, so the assign is required at the end of the HDL code block to drive the **COUNT** wire with the value of **COUNTER** (which must be a register in order to remember its value).

To increase the size of the counter to acquire 4K data values (**do not do this now**), we could change the MSB of **COUNT** to 11 and change the declaration of **COUNTER** in the HDL code to:

```
reg [11:0] COUNTER;    //example only, don't do in this tutorial
[****]
```

## (TD) 6.8 Modeling n-bit Gates

Next we will model the **DONE** signal that we originally drew as an input to the state machine. The **DONE** signal is generated by performing a bitwise AND of the **COUNT** signal (we are done whenever all the counter bits are high).

**To model the DONE Signal:**

1. Double click on the **DONE** signal name to open the *Signal Properties* dialog box.
2. Enter the following equation in the Boolean Equation edit box: `&COUNT`
3. Check the **Simulate** radio button. The resulting signal should look like the hand drawn signal except that it is a purple simulated signal.

The `&` operator when used as a unary operator is called a *reduction-AND* operation. A *reduction-AND* indicates that all the bits of the input signal should be ANDed together to generate a single bit output. This is equivalent to the following equation: `COUNT[0] & COUNT[1] & COUNT[2] & ...`

One nice benefit of using a reduction operator instead of the above equation is that it automatically scales the circuit to match the current size of the **COUNT** signal (it's also a lot easier to type)!

## (TD) 6.9 Incorporating Pre-written HDL Models into Waveformer Simulations

We will use an SRAM HDL module contained in an external file (**sram.v**) to model the SRAM. This model is fairly complex and accurately models the asynchronous interface that is commonly used by most off-the-shelf SRAMs. One special feature is that the SRAM resets all its memory cells to zero when it first starts up. In a real circuit, we would need to add extra logic to iterate through the addresses, writing zeros at each one. A full description of the Verilog modeling of this SRAM is outside the scope of this tutorial, but let's take a quick look at it inside the *Report* window:

1. Select the **Report > Open Report Tab** menu option and open the file **sram.v** (located in the **SynaptiCAD\lib\Verilog** directory). Verify that you can view the file in the *Report* window. Keep this file open because we will be referring back to this file later in the tutorial.

### 9.1 Including an external SRAM Verilog model file into WaveFormer

To add the SRAM model to our design we need to modify the **wavelib\_exact.v** file that contains the models used by WaveFormer. The SRAM model code cannot be entered into a signal's HDL code window because the model declares a module and modules cannot be nested in Verilog (WaveFormer puts all the HDL code from signals into a single module called testbed). All user-written Verilog modules should be declared in **wavelib\_exact.v** (or preferably, included from separate files into **wavelib\_exact.v** using the include directive as will be doing). In this case, the source code for the SRAM is already contained in a separate file called **sram.v** and we only need to add an include statement to **wavelib\_exact.v** to let WaveFormer know about it. To modify the **wavelib\_exact.v** file:

1. Select the menu option **Report > Open Report Tab** and open the **wavelib\_exact.v** file in the **SynaptiCAD\hdl** directory.
2. Add the following line to the beginning of the **wavelib\_exact.v** (it may already be there depending on which SynaptiCAD product you are using): ``include "lib\verilog\sram.v"`
3. Select the **Report > Save Report Tab** menu option to save your change.

### 9.2 Instantiating the SRAM component models

To drive the data bus **DBUS**, we need to instantiate two instances of the SRAM model:

1. Create a new signal called **DBUS**.
2. Set the Radix to **hex**, set the MSB to **15**, check the **Simulate** radio button, and select the **Verilog** radio button.
3. Enter the following HDL code into DBUS's HDL code window:

```
wire CSB = !ENABLE;
sram BinMem1(CSB,READ,ADDR,DBUS[7:0]);
```



```
sram BinMem2(CSB,READ,ADDR,DBUS[15:8]);
```

The first line creates an internal signal that is an inverted version of the ENABLE line (the SRAM is active low enabled). The next two lines instantiate two 4Kx8 SRAMs and connect up their inputs and outputs (the first SRAM contains the low byte of the count and the second contains the high byte).

COUNT[3:0]	0	1	2	3	4	5	6
DBUS[15:0]	ZZZZ	0000	ZZZZ	0000	ZZZZ	0000	

## (TD) 6.10 Modeling the Incrementor and Latch Circuit

In Section 3 we used the Boolean Equation interface to model the state machine using negative edge-triggered registers. Now we will use the same interface to generate level-triggered latches used to model the increment-and-latch circuit. The value stored in the SRAMs is placed on DBUS and the incrementor circuit takes that value, adds one to it, and latches the incremented value:

1. Create a new signal called **DBUS\_INC**.
2. Enter the following equation into the **Boolean Equation** edit box: **DBUS + 1**
3. Choose the **READ** signal from the **clock** drop-down list box.
4. Choose **high** from the **Edge/Level** drop-down list box. This selects the type of latch to be used.
5. Set Radix to **hex**, MSB to **15**, and check the **Simulate** radio button.
6. Press the **Simulate Once** button and verify that DBUS\_INC is an incremented version of DBUS. If DBUS\_INC did not simulate, use the methods in section 4 to determine the error.

COUNT[3:0]	0	1	2	3	4	5	6	7
DBUS[15:0]	ZZZZ	0000	ZZZZ	0000	ZZZZ	0000		
DBUS_INC[15:0]	XXXX	0001	0001	0001	0001			

## (TD) 6.11 Modeling Tri-State Gates

There are 2 possible drivers for DBUS: the SRAMS which we modeled in section 9, and the tri-stated output of the DBUS\_INC signal. All the drivers for a bus should be included in the code for the bus.

**To add the tri-state gate to DBUS:**

1. Double click on the **DBUS** signal name to open the *Signal Properties* dialog box.
2. In the direct HDL code edit box add a 4th line of HDL code to DBUS:

```
assign DBUS = WRITE ? DBUS_INC : 'hz;
```

COUNT[3:0]	0	1	2	3	4	5	6	7	8	
DBUS[15:0]	ZZZZ	0000	0001	0000	0001	0000	0001	0001	0002	0001
DBUS_INC[15:0]	XXXX	0001	0001	0001	0001	0001	0002	0002	0002	0002

Line 4 models the tri-state gates that follow the latches in the histogram circuit. These tri-state gates are enabled whenever the WRITE signal is high. We use the conditional operator (condition ? x : y) which acts like an if-then-else statement (if condition then x else y). If WRITE is high, DBUS is driven by DBUS\_INC (the incremented version of DBUS that we latched), else the tri-state drivers are disabled ('hz means all bits are tri-stated).



## (TD) 6.12 Debugging External Verilog Models

Verilog contains two system tasks (commands), **\$display** and **\$monitor**, that can be included in Verilog source files for debugging purposes. **\$display** acts like a C-language **printf** statement which prints to the simulation log file **simulation.log** whenever it is executed by the Verilog simulator. **\$monitor** is similar, but it automatically prints to the log file whenever any of the signals listed in this command change state. The SRAM model file **sram.v** contains two **\$display** statements that output the address and data values for the SRAM whenever the SRAM is read from or written to (you can view the **\$display** commands in **sram.v** in the *Report* window). You can see the output of the **\$display** commands by viewing **simulation.log** in the *Report* window. Each time the SRAM performs a read or write a message is sent to the log file.

```
780 Reading syncad_top.BinMem1 ABUS=001 DATA=01
780 Reading syncad_top.BinMem2 ABUS=001 DATA=00
950 Writing syncad_top.BinMem1 ABUS=002 DATA=02
950 Writing syncad_top.BinMem2 ABUS=002 DATA=00
```

## (TD) 6.13 Verify the Histogram Circuit

At this point we have modeled the entire histogram circuit, so your diagram should resemble the figure below. If it doesn't, check the **simulation.log** for errors and correct as necessary. The output of the **\$display** commands will be particularly useful if you are getting x's on your **DBUS** signal which indicates unknown data is being read from your RAMs. One thing to check for is that your diagram is never performing a write to an unknown address (an address containing x's) in your RAM bank. If you write a value to an unknown address, the memory model has no way of knowing which memory location has been changed. Therefore, all the memory locations in the entire address space of the RAM bank may or may not have been changed. The memory model is forced to represent this unknown state by setting all memory locations in the SRAM to x!

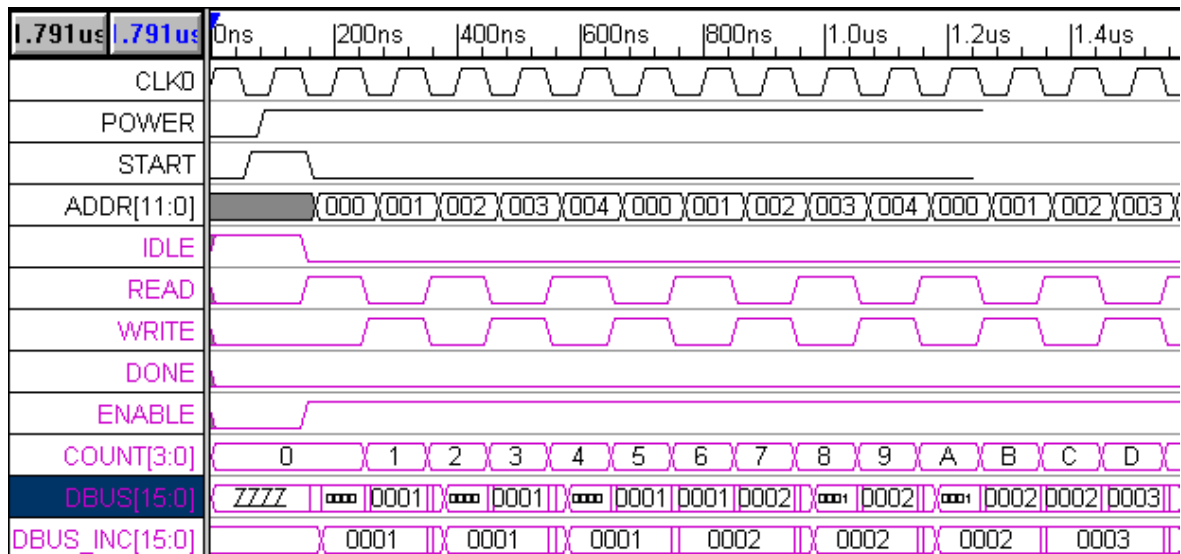


Figure 3: Completed Timing Diagram

## (TD) 6.14 Controlling the Length of the Simulation

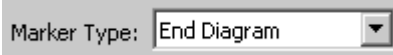
By default, WaveFormer simulates to the last drawn signal edge. You can also use a time marker to control the length of the simulation. To place a time marker:

1. Click the **Marker** button  found on the button bar. This turns the Marker button red which

indicates that right clicks in the *Diagram* window will add marker lines.

2. Right click at about **1us** in the *Diagram* window. A new time marker line will appear.
3. Double click on the marker to open the *Edit Time Marker* dialog.

4. Set the marker type to **End Diagram**.



5. Click **OK** to close the dialog, then **drag** the marker on the screen. As you move the marker, the simulator will automatically resimulate the design up to the time location of the marker.

## (TD) 6.15 Editing Verilog Source Files

To demonstrate how to make changes to a Verilog source file inside WaveFormer, we will edit the SRAM model file `sram.v` in the Report window:

1. Change **line 18** from: `ram[i] = 0;` To `ram[i] = 8;`  
This causes the SRAM cells to be initialized with 8 instead of zero.
2. Select the **Report > Save Report Tab** menu option to save your change.  
Let's see the effect of this change:
3. Press the **Simulate Once** button in the *Signal Properties* dialog, or move an input edge. Either of these steps initiates a resimulation.

You may have anticipated that DBUS would now show 8 (we did when we first did this tutorial!), but it is correct in showing 808 because our DBUS is a 16-bit value composed of the data in two parallel SRAMs each initialized with 08 (hence 0808 = 808).

4. Reset the line back to `ram[i] = 0;`

## (TD) 6.16 Simulating Your Model with Traditional Verilog Simulators

The Verilog model of your system created by WaveFormer can also be simulated by traditional Verilog simulators. The complete verilog model simulated by WaveFormer is composed of (1) the verilog file generated by WaveFormer (**untitled.v** for this tutorial), (2) the WaveFormer library file **wavelib.v**, and (3) any external model files you have included (e.g. **sram.v** for this tutorial). Follow the instructions of your Verilog simulator to simulate these files together.

## (TD) 6.17 Summary

This concludes the advanced simulation tutorial. Other simulation features not covered in this tutorial that you may wish to experiment with are: flip-flop timing characteristics (clock to output propagation delay and continuous setup and hold time checking) in the *Signal Properties* Dialog and the global simulation options in the **Options > Simulation Preferences Dialog**.

## Test Bench Generation 1: VHDL-Verilog Stimulus

This tutorial describes how to generate Verilog and VHDL stimulus files using WaveFormer Pro, VeriLogger Pro and TestBench Pro. This tutorial is important because it describes exactly how the waveforms of a single timing diagram will be exported. It also covers advanced data types that are used in VHDL generation.

TestBench Pro customers should also work through the on-line TestBench Tutorials, which cover the sample parameters that generate the self-testing code, and modifying the template files used to generate multi-diagram test benches.

This tutorial covers how the following objects are exported into VHDL and Verilog:

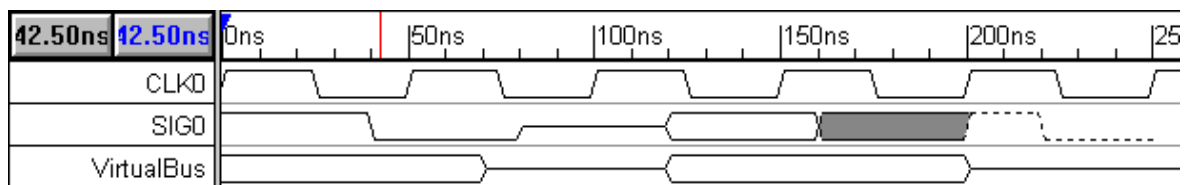
- clocks & signals
- graphical waveform states (high, low, tristate, valid, invalid, weak high, weak low)
- virtual buses with hex, binary, and other data values
- VHDL user-defined types and integer types

### (TBench) 1.1 Getting Started

Get a Full Version License

If you are evaluating WaveFormer Pro, VeriLogger Pro or TestBench Pro you need to upgrade the evaluation version by obtaining a two week trial license. This license will turn your evaluation version into a full version for two weeks. To obtain a two week evaluation license, complete the form under **Help > Request License**, or contact our sales department.

1. Select the **File > Open** menu option and load file **tuthdl.btim** from the **SynaptiCAD\Examples\TutorialFiles\AdvancedHDLStimulusGeneration** directory.
2. Select the **File > Save As** menu option and save the project as **test.btim** (this will keep the original file intact).



The first signal, **CLK0**, is a clock with a period of 50ns. The second signal, **SIG0**, is a waveform that contains all of the graphical states available in WaveFormer Pro. The third signal, **VirtualBus**, is a waveform drawn with valid and tristate segments.

### (TBench) 1.2 Default Mappings: Hex and Binary Translations

WaveFormer Pro supports a language independent bus format for hexadecimal and binary bus values.

During the translation to Verilog or VHDL, the extended state value of a segment is evaluated to determine if it is a hexadecimal or binary number. If the extended state 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. 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.

To demonstrate the hex and binary translations, we will edit the signal **VirtualBus** so that it will correctly export as an 8-bit bus. We will also use the **'b** and **'h** values to set the segment values

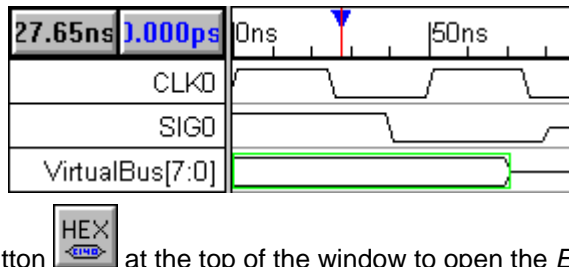
and compare how they export in VHDL and Verilog. Later in the tutorial we will generate the Verilog and VHDL code.


#### Setting the size of a virtual bus to 8 bits:

1. Double click on the **VirtualBus** signal name to open the Signal Properties dialog box.
2. Type **7** into the **Bus MSB** edit box.
3. Type **0** into the **Bus LSB** edit box.
4. Click **OK** to close the dialog box.

#### Setting the values in a virtual bus waveform:

1. **Select** the first waveform segment of *VirtualBus* by clicking on it. A selected segment has a box around it.



2. Click on the **HEX** button  at the top of the window to open the *Edit Bus State* dialog box. The *Edit Bus State* dialog box can also be opened by **double-clicking** on the selected segment.
3. Type **'b11101110** into the **Virtual** edit box. This is an 8-bit binary number.
4. Press **ALT-N** (or press the **Next** button) two times to advance to the next valid segment.
5. Type **'hA** into the **Virtual** edit box. This is the 8-bit hexadecimal number A (00001010 in binary). The program automatically left pads missing bits with zeros.
6. Click the **OK** button to close the *Edit Bus State* dialog.

## (TBench) 1.3 Generating Verilog Code

Next we will demonstrate how to generate Verilog stimulus vectors from timing diagrams.

1. Choose the **Export > Export Timing Diagram As** menu option to open the *Export* dialog.
2. In the **Save as Type** list box in the lower left corner of the dialog, choose the **Verilog (\*.v)** script. This indicates that the timing diagram will be exported to a Verilog code file with a default file extension of ".v".
3. Choose **test.v** as the file name and click the **Save** button to close the dialog. WaveFormer Pro will produce a Verilog file named test.v.
4. The file **test.v** is automatically displayed in the Report window. If you cannot see the *Report* window, select the **Window > Report Window** menu option to bring the window to the top.



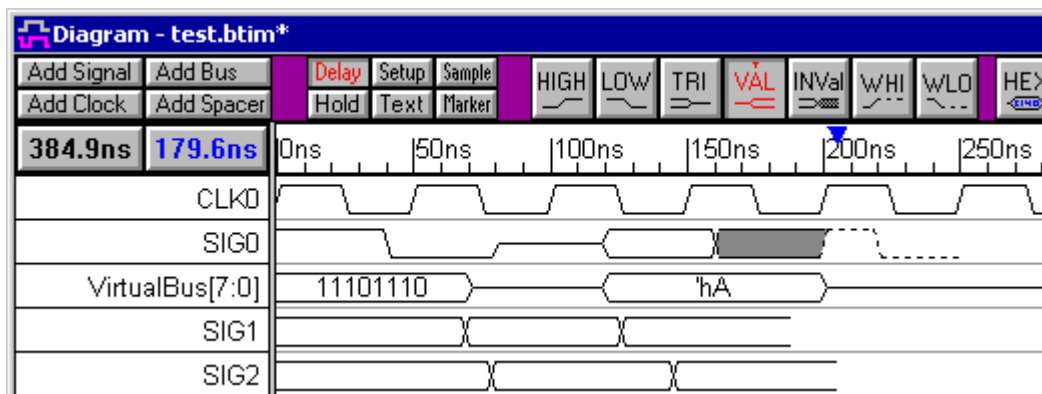
Look at the resulting file by clicking on the **test.v** tab on the bottom of the *Report* window. Notice how CLK0 uses a while loop to produce its transitions and how SIG0 uses assignment statements. Also note, values for the VirtualBus assignments have a 8' in front which indicates that VirtualBus is an 8-bit vector. The first segment of VirtualBus has a value of, 8'b11101110, which is the correct Verilog syntax for an 8-bit bus with a binary value of 'b11101110. The next segment has a value of 8'bzzzzzzzz which is the value for an 8-bit tri-stated bus. Next value is 8'b00001010 which is a zero padded translation of the hexadecimal value 'hA.

## (TBench) 1.4 VHDL - Advanced Data Types

WaveFormer Pro supports both simple **std\_logic** signals and complex user-defined data types for VHDL test benches. By default all signals are assumed to have a type of **std\_logic** and a direction of **out** (CLK0, SIG0, and *VirtualBus* will use the defaults for this tutorial). In this section you will add SIG1 and SIG2 to demonstrate signals with a standard integer type and a user defined type.

### Add SIG1 and SIG2

1. Click on the **Add Signal** button two times to add two signals.
2. Double click on the **VAL** button. The first click selects Valid as the initial graphical state, and the second click selects Valid as the toggle state (as indicated by the red T). This causes the VAL button to stay selected when you draw waveform segments.
3. Sketch some valid waveforms for **SIG1** and **SIG2** similar to those in the figure below.



Change the type of **SIG1** to integer and the type of **SIG2** to MyColor.

1. Double-click on the **SIG1** signal name to open the *Signal Properties* dialog box.
2. Choose **integer** from the **Signal Type** drop down list box. You may have to scroll down to find the entry.
3. Choose **dec** from the **Radix** drop-down list box to indicate that the values you will be entering into the virtual state are decimal values.
4. Click the **Next** button to move to **SIG2**. Make sure that you moved to **SIG2** and not to the previous **VirtualBus** signal.
5. Type in **MyColor** into the **Signal Type** drop down list box. *MyColor* is the name of the user defined type that we will use.
6. Click the **OK** button to close the dialog.

Add Extended State information to the waveforms of **SIG1**.

1. Double-click on the **first** waveform segment on **SIG1** to open the *Edit Bus State* dialog box.
2. Type **25** into the **Virtual** edit box and move to the next segment (**ALT-N** or the **Next** button).
3. Enter **integer values** for each segment (we used **25, 50, 47**).

Add Extended State information to the waveforms of **SIG2**.

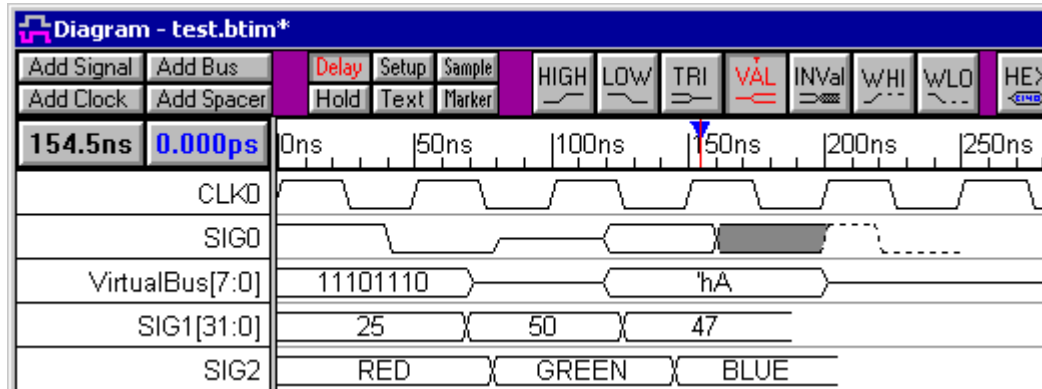
1. Select the **first** waveform segment on **SIG2**. Notice that the *Edit Bus State* dialog changes to display state information about the current segment.
2. Type **RED** into the **Virtual** edit box and move to the next segment (**ALT-N** or the **Next** button).
3. Enter color values for each segment (we used RED, GREEN, BLUE).

For a real VHDL simulation, the color value would have to match the enumerated values

defined for MyColor in your original circuit model. For instance, MyColor might be defined as: **enum MyColor = {RED, GREEN, BLUE, BLACK};**

4. Click **OK** button to close the *Edit Bus State* dialog and exit the HEX mode.

Your timing diagram should resemble the figure below.



## (TBench) 1.5. Exporting to VHDL

Export to a VHDL stimulus file:

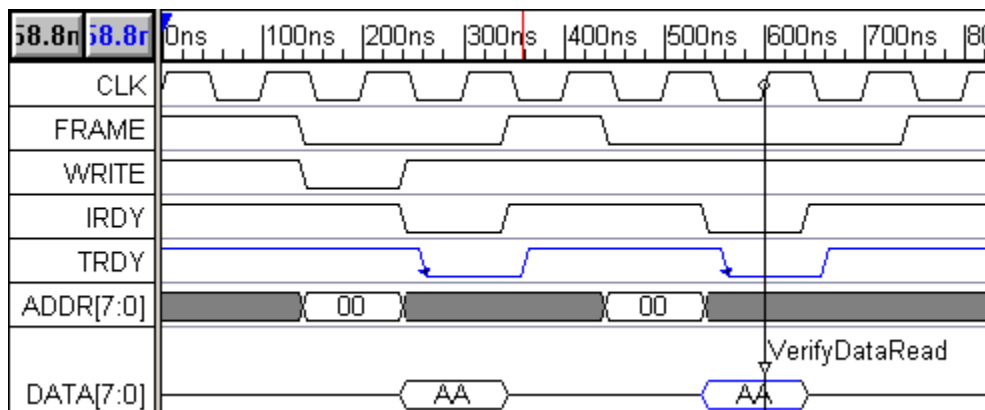
1. Choose the **Export > Export Timing Diagram As** menu option to open the *Save As* dialog.
2. Choose **VHDL (\*.vhd)** script using the **Save File as Type** list box in the lower left corner of the *Save As* dialog.
3. Click **Save** to close the edit box and generate the VHDL transport stimulus file.

View the file **test.vhd** inside the *Report* window. Notice the entity and architecture structures and the types of all the signals. CLK0 uses a while loop to calculate its value. **SIG0** shows how the graphical states are exported. **VirtualBus** is defined as an 8-bit logic vector. SIG1's values are exported as integers. SIG2's values are exported as RED, GREEN, and BLUE.

Congratulations, you have now completed the HDL stimulus generation tutorial.

## Test Bench Generation 2: Reactive Test Bench Option

The Reactive Test Bench Generation Option can be added to WaveFormer Pro, DataSheet Pro, VeriLogger Pro, and BugHunter Pro. The features are included in TestBencher Pro, so it is a good introduction for creating a single timing transaction in TestBencher Pro, also. With Reactive Test Bench Generation, users can draw "expected" waveforms on the MUT output ports and add "samples" to the waveforms to test for specific state values. During simulation, the code generated by the samples would watch the output from the model under test and compare it to the drawn waveform states. 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.



All of the relevant files for this tutorial can be found in the **SynaptiCAD\Examples\TutorialFiles\ReactiveTestBench** directory. At the end of this tutorial, you will have created one timing diagram that uses many different reactive features. There are also pre-made diagrams for each completed step, allowing you to start at any step of the tutorial desired. These completed diagrams can be found in the **ReactiveTestBench\CompletedDiagrams** directory.

### (TBench) 2.1 The Model Under Test

We will use a simplified version of a PCI slave device as the model to be tested. The model is contained in **mymut.v** and the module is named **mymut**. No experience with PCI is required to perform and understand this tutorial. There is no arbitration, the MUT responds to all addresses, and the only valid commands are single reads and writes. It contains a memory that can be written to and read from and has the following ports (all control signals are active low):

- **CLK (input)**: device is clocked on the negative edge
- **FRAME (input)**: indicates start of transaction.
- **WRITE (input)**: indicates write transaction.
- **IRDY (input)**: stands for **initiator ready**. Indicates when the master device is ready for transaction to complete (the master will be the test bench in this case).
- **TRDY (output)**: stands for **target ready**. During a write, this indicates that the MUT has finished writing data to its memory. During a read, this indicates that the MUT has read the data from memory and put it on the DATA bus.
- **ADDR (output)**: Address to write to or read from.
- **DATA (inout)**: Data to write to memory or data that is read from memory.

Each transaction consists of an address cycle and data cycle. During the address cycle, the

**WRITE** and **ADDR** signals must be valid. During a write data cycle, the **DATA** signal must be valid before **IRDY** is asserted. Then the MUT indicates that it is finished storing the data by asserting **TRDY**. During a read data cycle, the MUT must drive **DATA** before asserting **TRDY**. Then, the master asserts **IRDY** when it is finished reading the data. Once **IRDY** or **TRDY** is asserted, they must remain asserted until the transaction is finished which is indicated by the de-assertion of **FRAME**.

## (TBench) 2.2 Create Signals

### A) Create a Project


First, create a project file to hold the model under test. This will allow WaveFormer to extract the information necessary to instantiate the model under test inside the testbench and extract the input and output ports from the model under test.

#### Non-Libero users:

- Choose **Project > New Project** menu function to open the *New Project Wizard* dialog.
- In the **Project Name** box, type in **reactivetut** as the name of the project. This will become both the name of the project and the directory where the project and associated files are stored.
- Click the **Finish** button to create the project and close the dialog.
- Inside the **Project** window, right click and choose **Copy New Source Files** from the context menu. This will open a file find dialog.
- Select **mymut.v** from the **SynaptiCAD\Examples\TutorialFiles\ReactiveTestBench** directory.
- The mymut.v file will be copied to the project directory and listed in the project window.

**Actel Libero users only:** Create a project inside Libero following the steps in the Libero documentation, add the source file **mymut.v** from the **SynaptiCAD\Tutorials\Reactive Test Bench** directory, and launch WaveFormer Lite. This will automatically create a WaveFormer project file and add your source files to that project.

### B) Extract MUT ports into Diagram

- Next, press the , **Extract MUT ports into Diagram** button to extract the top level ports from **mymut.v** and dump them into a timing diagram window.

### C) Create Clock Waveform

Once the ports are extracted, convert the signal named **CLK** to a Clock:

- Right-click on the CLK label name and select **Signal <-> Clock** from the context menu. This will draw a clock waveform with a default frequency of **10MHz**.

### D) Set Default Clocking Signal and Edge

Next set the **Clock** signal and **Edge** for all of the signals in the diagram so that the test bench will be cycle-based instead of time-based (this means the test bench stimulus will change after waiting on clock transitions instead of time delays).

- Right-click in the signal name list in the diagram window and select **TestBencher Diagram Properties** context menu to open a dialog of the same name.
- In the **Default Clock** drop down, select **CLK** as the default clocking signal to use and set the **Edge** dropdown as **pos** to specify positive-edge clocking.
- Click the **Update Existing** button to set the clocking signal for existing signals. Press **OK** to close the dialog.



- The diagram window will have all of the signal names and the clock waveform. In the next section you will draw the rest of the waveforms.

Following is an example of the difference between a cycle-based and time-based test bench. Both of these code segments were exported from the diagram you will be drawing in the next step. The example on the left is time-based and the example on the right is cycle-based.

Signals with the same clocking signal and edge type will be driven by a common process in the generated code. We call a process of this type a Clocked Sequence. All the unclocked signals are driven by the Unclocked Sequence. This means that when a diagram contains signals with different clocking signals, a separate sequence process will be created for each clocking signal/edge type. Clocked Sequences are named based on the clocking signal and the edge type, so for this example, the clocked sequence that contains the code on the right in the example below will be called **CLK\_pos**.

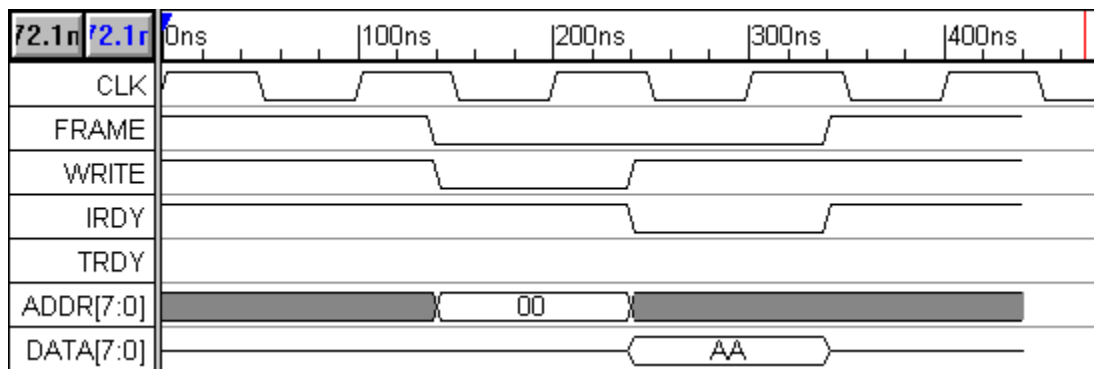
```
#137;
FRAME_driver <= 1'b0
#3;
WRITE_driver <= 1'b0
ADDR_driver <= 8'h00
#100;
WRITE_driver <= 1'b1
IRDY_driver <= 1'b0;
ADDR_driver <= 8'hxx
DATA_driver <= 8'hAA
#100;
FRAME_driver <= 1'b1
IRDY_driver <= 1'b1;
DATA_driver <= 8'hzz
#101;
```

```
repeat (2)
begin
    @(posedge CLK);
end
FRAME_driver <= 1'b0;

WRITE_driver <= 1'b0;
ADDR_driver <= 8'h00;
@(posedge CLK);
WRITE_driver <= 1'b1;
IRDY_driver <= 1'b0;
ADDR_driver <= 8'hxx;
DATA_driver <= 8'hAA;
@(posedge CLK);
FRAME_driver <= 1'b1;
IRDY_driver <= 1'b1;
DATA_driver <= 8'hzz;
@(posedge CLK);
```

## (TBench) 2.3 Draw Single Write (without waiting on TRDY)

Draw the write transaction shown below. This transaction could be used as a simple test bench that just drives the input ports of the MUT, but it ignores the **TRDY** signal and doesn't verify that the data was actually written successfully to the MUT. We will add this functionality in the next couple of steps. If you have trouble drawing the waveforms, please review the **Basic Drawing and Timing Analysis Tutorial** before continuing with the rest of this tutorial.



## (TBench) 2.4 Export Diagram as an HDL Test Bench

To Generate a Verilog or VHDL test bench:

- Select the menu option **Import/Export > Export Timing Diagram As** to open the *Export As* dialog.
- In the **Save As Type** drop-down, select either:
  - **Verilog w/ Top Level Test Bench** to create a test bench that includes an instantiation of the model under test, or
  - **Verilog** to generate just the model that includes the testbench code
- Press the Ok button to generate the test bench.
- You can view the generated code in the **Report** window, under the tab of the saved file. (If you cannot see the Report window, choose the menu **Window > Report** to bring it to the front).

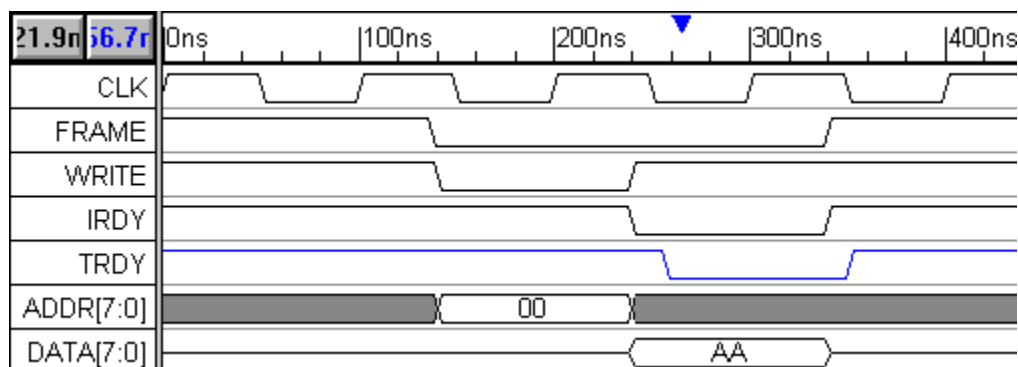
As you proceed through this tutorial, you will periodically perform the steps above to see how the generated code changes as the timing diagram is edited.

## (TBench) 2.5 Add Wait for TRDY Assertion

There are two ways to perform this step. One method uses the **Sensitive Edge** feature and will wait indefinitely for **TRDY** to assert. The other method uses a **Sample** instead, so that a timeout can be specified. Both methods are explained below.

### Draw the Expected Waveform for the TRDY signal

- First draw the expected **TRDY** waveform shown below. Notice that the waveform is blue because it is an input to the diagram, so the data shown is predicted data, not data to be driven. The direction of the signal was imported during the extract ports from mut step.



### A) Wait Indefinitely Using Sensitive Edges

A 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. Here we will edit TRDY's properties so that the test bench will wait until TRDY has a falling edge before continuing to supply stimulus to the model under test.

- Double-click on **TRDY** to open the *Signal Properties* dialog.
- Check the **Falling Edge Sensitive** check box, and press **OK** to close the dialog.
- Notice that TRDY now has an arrow drawn on the falling edge, to indicate that it is falling edge sensitive.
- Make sure that the falling edge of TRDY is drawn after the falling edge of IRDY, otherwise the test bench will wait for TRDY to assert before asserting IRDY.

To see the generated code, export the test bench to your preferred language and view the code in the Report tab that contains the generated code. Use the search tool in the button bar to locate the CLK\_pos sequence (in verilog, search for "task CLK\_pos", in VHDL search for "CLK\_pos :"). Then search down for "Sensitive" to view the code generated for the sensitive edge. You should see code similar to the following:

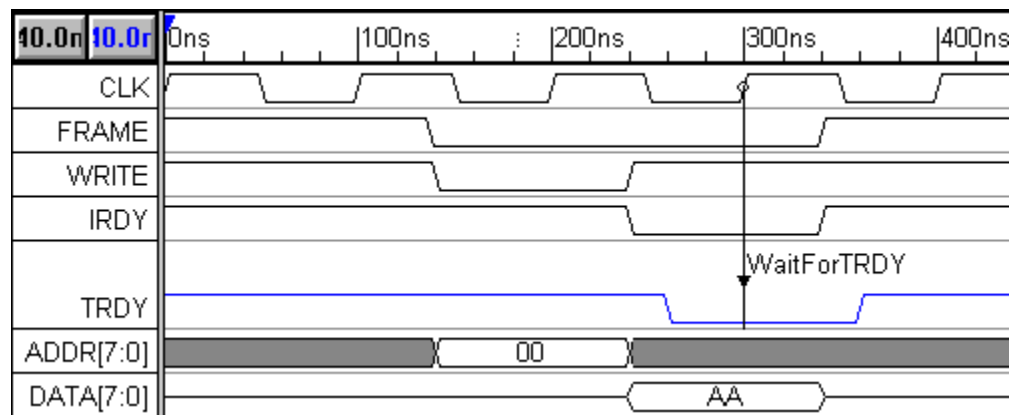
```
VHDL:          -- Sensitive Falling Edge on signal: TRDY
                wait until falling_edge(TRDY) or (tb_DgmAborted);
Verilog:       // Sensitive Falling Edge on signal: TRDY
                @(negedge TRDY);
```

Using this method, the transaction will wait indefinitely for TRDY to assert. If you want the transaction to timeout if TRDY doesn't occur after a given time, use the sample approach below instead to wait for TRDY.

### B) Wait with a Timeout Using a Sample

An alternate approach to waiting on an incoming edge is to use a graphical sample. If you have already performed the steps in section A, uncheck the sensitive edge check box to cancel the indefinite wait that you created in the previous section before performing this step.

- Press the **Sample** button at the top of the diagram window to enter into sample mode.
- **Left-click** on the rising edge of **CLK** at **300ns** to select the edge, then **right-click** on **TRDY** at **300ns**. This will add a sample to the diagram.
- Double-click on the new sample's name to open the *Sample Properties* dialog.
- Change the name to **WaitForTRDY**, then click on the **HDL Code** button in the *Sample Properties* dialog to open the *Code Generation Options* dialog. Here is where you can control the behavior of the Sample once it is triggered to run. Make the following changes:
  1. Uncheck the **Full Expect** check box to indicate TRDY only has to assert at some time during the sample's execution.
  2. Specify **100** for the *Multiplier* to make the sample wait for up to 100 cycles of the clock since TRDY is a clocked signal.
  3. Check the **Blocking** check box so that the sample blocks the transaction until the sample finishes.

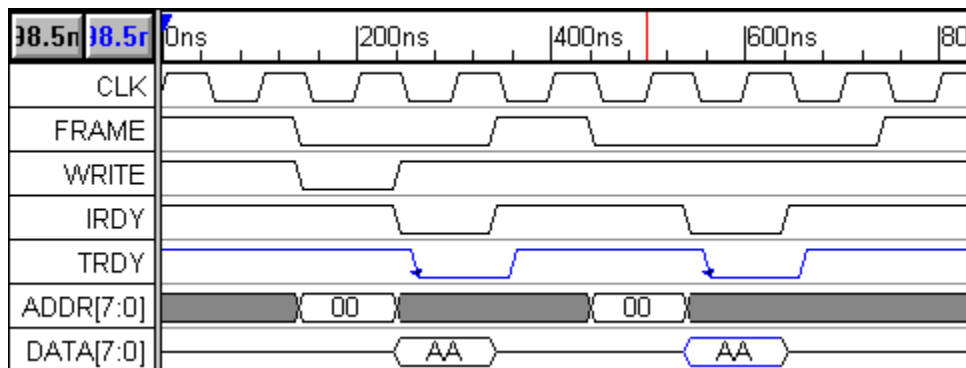


To see the generated code, export the test bench and view the code in the Report tab that contains the generated code. Locate the CLK\_pos sequence, then search down for "WaitForTRDY" to see the code generated for the sample.

## (TBench) 2.6 Draw Single Read

### A) Draw the Waveforms for the Read cycle

Draw a read transaction following the write transaction. Below is a picture of what the waveforms should look like. The read cycle starts at 400ns. The picture will vary if you did the sample version instead of the sensitive wait waveform.



### B) Disable Drive for the DATA Segment

To avoid bus contention, the test bench must not drive the DATA bus during the read cycle, because the MUT will be driving that bus. Since the DATA bus is a bi-directional signal, you can specify which parts of the waveform are driven by the test bench and which are not. One-way to do this is to draw a tri-state waveform. However, in this case we need to specify the expected data on the bus, so we will have to disable the drive on the expected value segments.

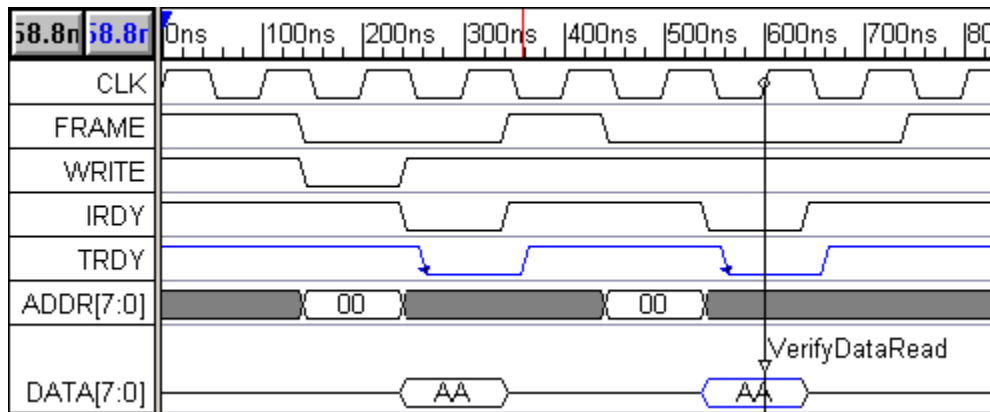
- Double-click on the waveform segment of **DATA** that happens during the read cycle to open the *Edit Bus State* dialog.
- Uncheck the **Driven** check box and click **OK**.
- Notice that the segment will be drawn in blue now, indicating that the **DATA** signal will *not* be driven by the test bench during this time period (just like the entire **TRDY** signal).

## (TBench) 2.7 Add a Sample to Verify Data Read From MUT

Samples can be used to verify data that is generated by the model under test during simulation. Here we will add a Sample to the Data bus to verify that the read cycle worked correctly.

- Press the **Sample** button at the top of the diagram window to enter sample mode.
- **Left-click** on the positive **CLK** edge at **600ns**, then **right-click** on the **DATA** segment directly below it. This will place a Sample that will trigger at that clock edge and verify that the data read from the MUT is what we expect (indicated by the waveform drawn under the Sample).
- Double-click on the Sample name to open the *Sample Properties* dialog.
- Change the sample name to **VerifyDataRead**
- Press the **HDL Code** button to open the *Code Generation Options* dialog.
- The **If condition** should be set to **Sample State Matches**, which is the default behavior for a sample
- Set the **Then Action**, to **Display Message** so that each time the sample passes it will generate a log messages stating that it passed.
- Click **OK** to close these dialogs.

Here is what the diagram should look like after adding the Sample:



To see the generated code, export the test bench and view the code in the Report tab that contains the generated code. Locate the CLK\_pos sequence, then search down for **VerifyDataRead** to see the code generated for the sample.

## (TBench) 2.8 Drive Data Using a "Test Vector Spreadsheet" File

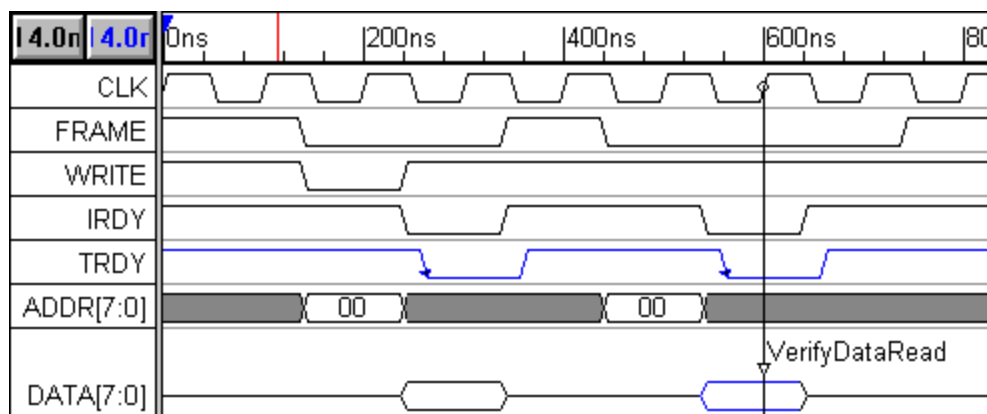
This step will use an ASCII input file to drive the **DATA** bus during the write cycle so that we can model a memory array. The basic idea is to create a user-defined array variable that is initialized from a file containing the values stored in the memory, then drive the DATA bus using the values of this array (using the current address value as the index for the array). Here are the steps to create the array variable.

- Click the **View Variables** button in the diagram to open the *Variable List* dialog.
- Click the **New Variable** button, then click on the name and change it to **inputData**. This name is important because it must match a column name in the input file that we choose.
- Under the *Structure* column select **array**.
- Set **Size** to **256** to indicate that we are creating an array of 256 elements.

- Set *Data Type* to **2\_state** then change *MSB* to **7** (this indicates we're storing 8-bit values in the array).
- Check the **Initialize Structure With File** checkbox near the bottom of the dialog.
- Browse to the **SynaptiCAD\Examples\TutorialFiles\ReactiveTestBench\inputData** directory and select **inputData.txt**.
- Hit **OK** to close the dialog

Now that the variable is created, the next step is to refer to this array to drive and verify data. So, both of the **AA** states need to be changed. For the two **AA** states, do the following:

- Double-click on the state to open the *Edit Bus State* dialog.
- Set the *Virtual State* to **@inputData[address]** and click **OK**. The @ symbol is used to refer to a variable defined in the *Variable List* dialog.



To see the generated code, export the test bench and view the code in the Report tab that contains the generated code. To see the code that initializes the inputData array, search for "inputData". Next, look at the CLK\_pos sequence, and search for "address" to find the assignment statement that drives the DATA bus with a value from inputData indexed by the address variable (the address variable will be created in the next step).

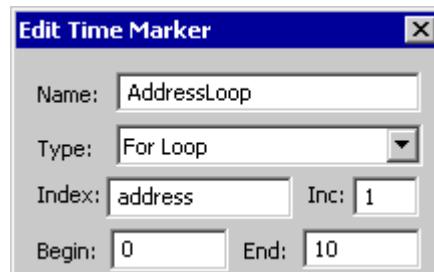
## (TBench) 2.9 Create For-Loop to Perform Multiple Writes and Reads

This step sets up the diagram to perform multiple writes and reads.

**Note:** If you are creating a TestBencher transactor then the next step should be performed, *TestBencher Pro Transactor - Add Address Argument*. Perform this step if you are unsure as it is also valid for TestBencher transactors.

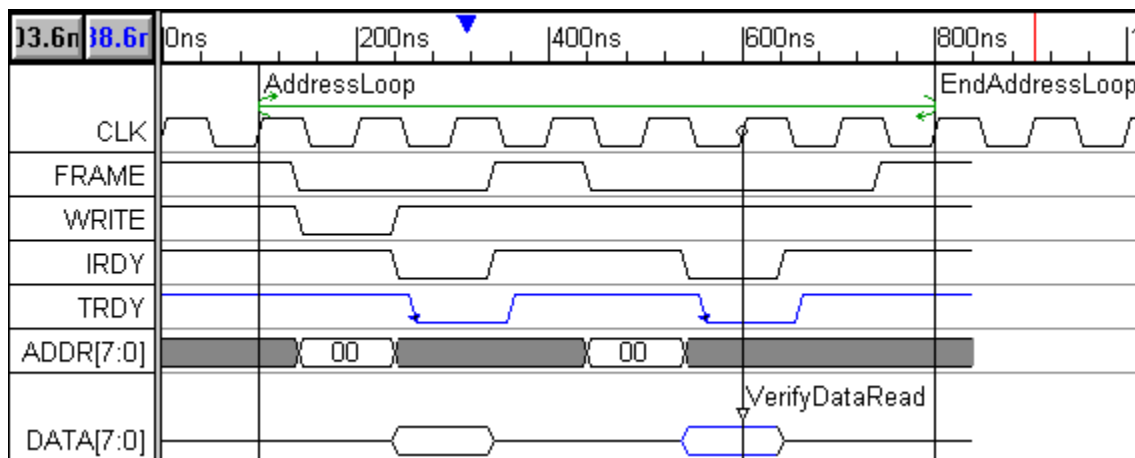
- Depress the **Marker** button, left-click the positive clock edge at **100ns**, then right-click to place the Marker.
- Place another marker at the positive clock edge at **800ns**.

- Double-click on the first Marker to open the *Edit Time Marker* dialog. Change the name to **AddressLoop**, set the type to **For Loop**, set *Index* to **address**, and set end to **10**. Then press **OK** to close the dialog



- Double-click the second Marker to open the *Edit Time Marker* dialog. Change *Type* to **Loop End** and then press the **OK** button to close the dialog

The two markers should now be connected graphically as shown below.



## (TBench) 2.10 TestBencher Pro Transactor - Add Address Argument

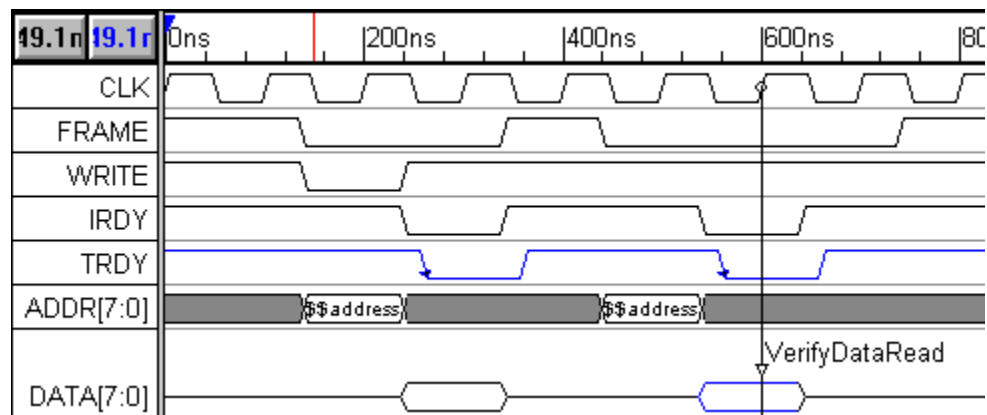
This step is optional and should only be performed if you are creating a TestBencher transactor. In this case, the for-loop can be omitted from the diagram and an argument can be set up for the address (i.e. the address can be passed in via the diagram apply call). It's not invalid to create a for-loop as performed in the previous step, but avoiding the for-loop gives the transactor greater flexibility.

**Note:** The primary purpose of this tutorial is to demonstrate various features available to all Reactive TestBench users. So, there are several steps that may not make as much sense for TestBench users. For instance, two transactors could have been created instead of one: one for the write cycle and one for the read cycle. Also, the data could have been passed in as an argument to the diagram apply call (a function call that causes the transactor to perform a transaction with a given set of transaction arguments).

To add an **address** argument, do the following for the two address states (which currently are set to **00**):

- Double-click on the state to open the *Edit Bus State* dialog.
- Enter **\$\$address** for the state value and click **OK**.

Here's what the final transactor should look like:



Now when an apply call is inserted for this transactor in the sequencer process, you will be able to specify which address to use.

## (TBench) 2.11 Alternatives

### *A) Consecutive Writes followed by Consecutive Reads*

If you want to perform multiple writes concurrently, followed by multiple concurrent reads, then two for-loops are needed. The array of data can be referenced in each loop in the same manner already demonstrated.

### *B) Random Data*

In Verilog, you could use `$random()` as the state value for **DATA** during the write transaction. A user-defined function can also be embedded into the generated test bench using the *Class Methods* dialog which could be used to generate data values. In both of these cases, you would need to modify the state value under the **VerifyDataRead** sample since the **inputData** array is no longer used. A Sample must be placed on the driven **DATA** segment to capture the expected data. For example, you could create a Sample named **ExpectedData** that is triggered from the clock edge at **300ns**. Then the state under the **VerifyDataRead** Sample would be set to **ExpectedData** instead of `@inputData[address]`.



## Test Bench Generation 3: TestBencher Pro Basic Tutorial

In less than 30 minutes you will create a reusable test bench that can apply different stimulus and verify the results of a clocked SRAM. Below is a schematic of the different components that you will construct. First you will create the Project file that controls the generation of the interface model (test bench). Next you will draw the different transaction diagrams that are needed to communicate with the SRAM. And then you will edit the sequencer process to apply the transactions to the model under test. Finally you will simulate the design and verify the operation of the SRAM model.

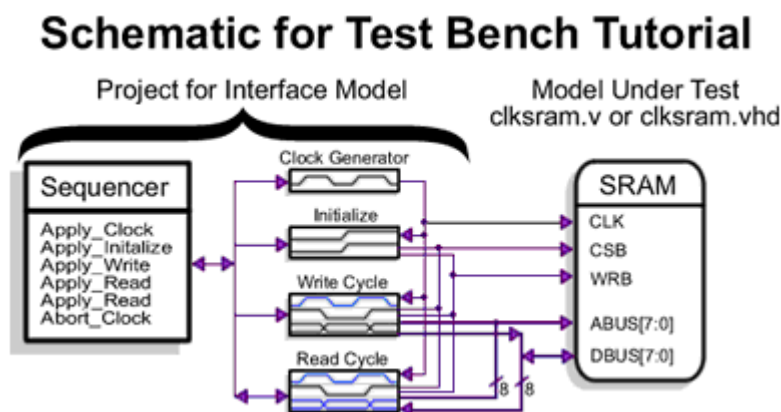


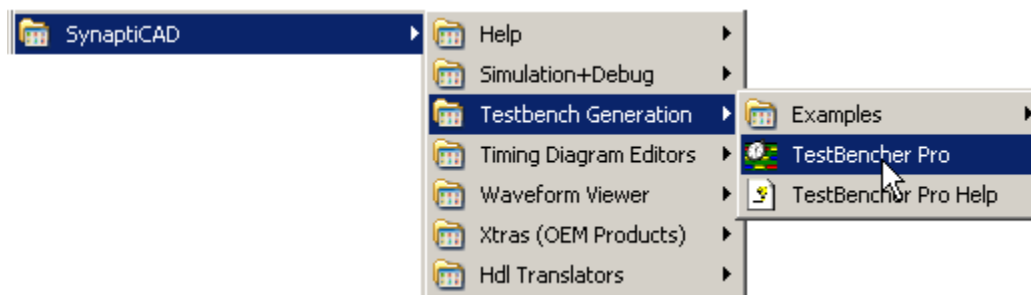
Figure 1: Tutorial Schematic

This tutorial assumes that you are familiar with the SynaptiCAD timing diagram editing environment. If you would like more information on the drawing environment then work through the short **Help > Tutorial > Basic Drawing and Timing Analysis** tutorial.

### (TBench) 3.1 Run TestBencher Pro

This tutorial requires a full version license for TestBencher Pro or an evaluation license. If you are evaluating then you can obtain a license by completing the form under the **Help > Request License** menu item and contacting our sales department. To check that you have a good license, verify that you can save a timing diagram.

- Run TestBencher Pro from the Start Menu.

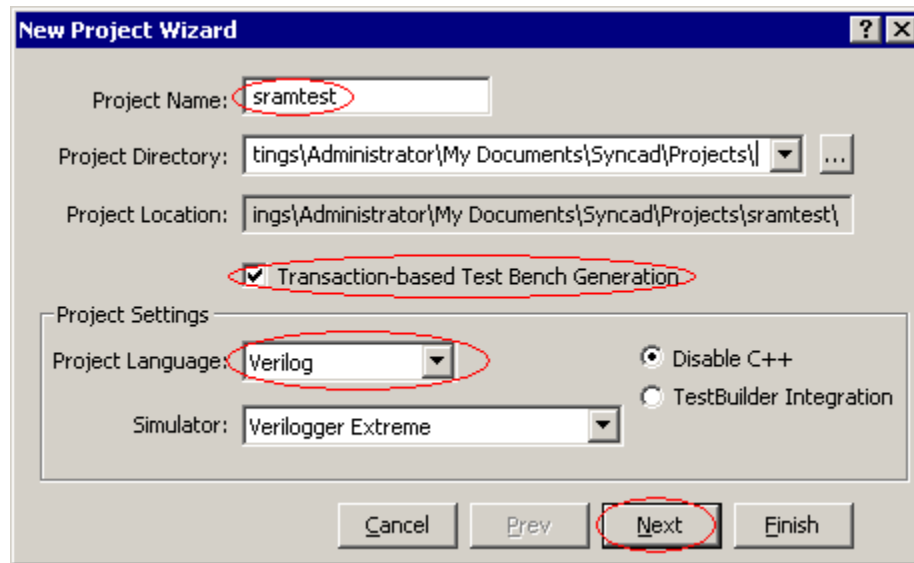
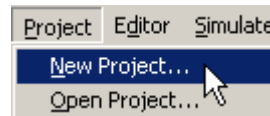


### (TBench) 3.2.Create a Project

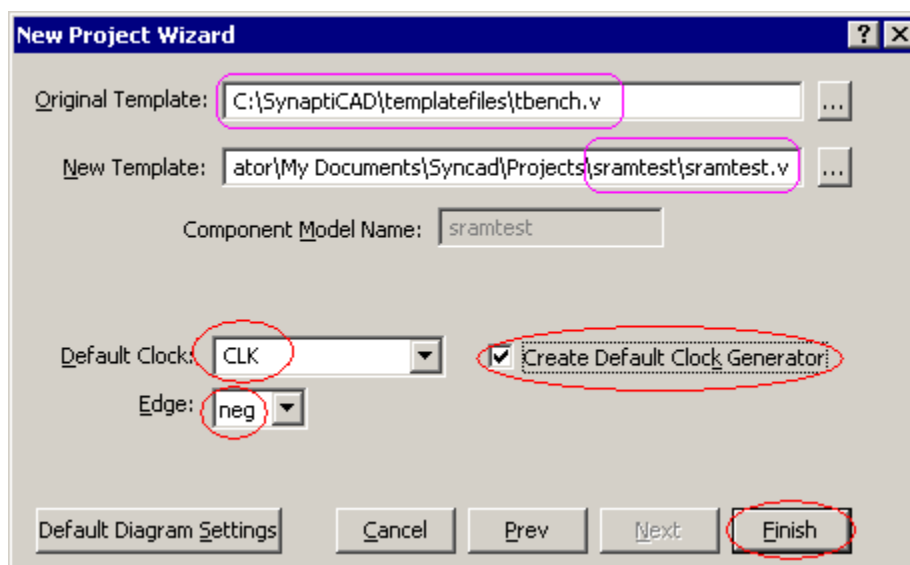
TestBencher Pro uses a project file to represent and to control the generation of a bus-functional model (BFM) component. The information in the project file is displayed in the Project window and context sensitive menus (right-click menus) provide a list of actions that can be performed for the elements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

Projects are created using New Project Wizard dialog. This dialog helps setup the project directory, the generated language, and the clocking signal for the project.

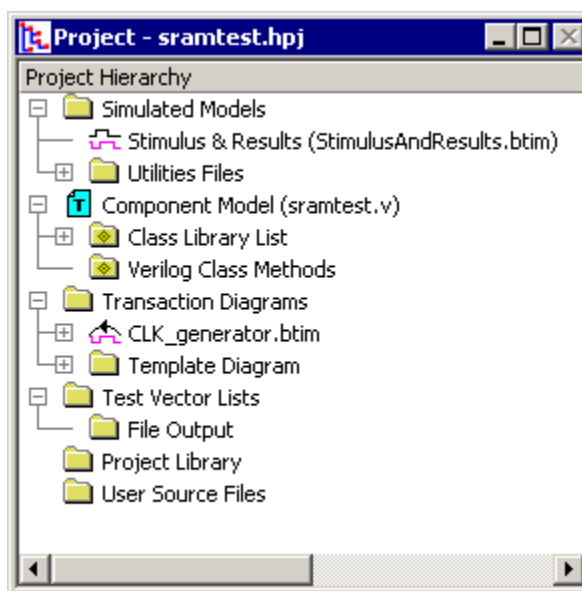
- Select the **Project > New Project** menu option to launch the *New Project Wizard* dialog.



- Enter **sramtest** in the **Project Name** edit box. This will be both the name of the project and the subdirectory where project files are located. The subdirectory will be placed underneath the *Project Directory* path. **Unix users** need to make sure that you have read/write access to the directory specified in the *Project Directory* edit box.
- Check the **Transaction-based Test Bench Generation** checkbox.
- Use the **Project Language** box to select the code generation language. This tutorial can be used to generate Verilog, VHDL, and TestBuilder code. Sometimes a file name will be written as filename.<language extension>. This means that the file extension will be different depending on the language used: Verilog \*.v, VHDL \*.vhd, and TestBuilder \*.cpp.
- Use the **Simulator** box to select your simulator. If you are evaluating, use the **Verilog** and **VeriLogger Extreme** combination, because the simulator is already set to work with TestBencher Pro. If you do not see your simulator pick one at random, then after the project is created read through the Chapter 1, Section 7 Setting Up Simulators in the TestBencher Pro Manual to setup the simulator manually.
- Press the **Next** button to move to the second page of the *New Project Wizard*.



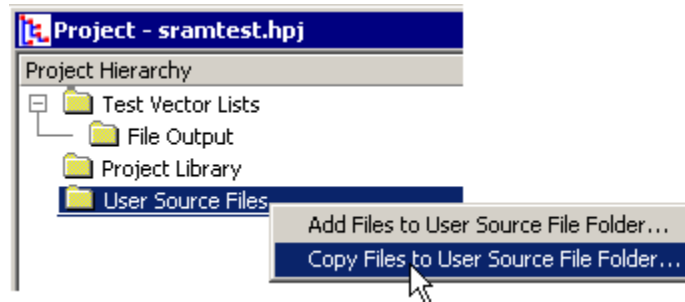
- Note that the name of the *New Template* is **sramtest** (the name of the project). TestBencher will use this file to generate the top-level module of the test bench. The *Original Template* is a file whose contents are copied into the new template file. Typically this file is **tbench.v** (a default file that ships with the software).
- Type **CLK** into the **Default Clock** box, and choose **neg** from the **Edge** box. Selecting a default clock causes the test bench to be cycle-based; if no clock is specified, the test bench will be event-based.
- Check the **Create Default Clock Generator** box. This will cause TestBencher to create a slave timing diagram called **Clk\_generator.btim** that will drive the default clock signal.
- Press the **Finish** button to close the *New Project Wizard*, create the project, and populate the *Project* window.



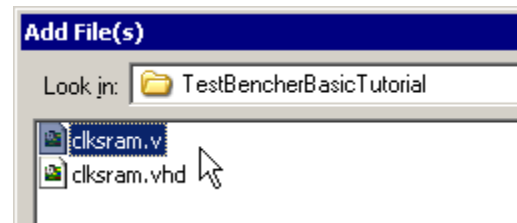
### (TBench) 3.3 Add the SRAM model to the Project

Next we will add the model under test (MUT) files to the project. TestBencher can parse the MUT files and extract the signal and port information for use in the transaction diagrams. Also, TestBencher uses the MUT information to instantiate the top-level component into the testbench model. For this tutorial we are going to test a clocked SRAM model.

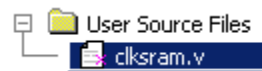
- Right-click the *User Source Files* folder in the *Project* window and select **Copy Files to User Source File Folder** from the context menu option. This will open the *Add Files...* dialog.



- Use the **Look in** box to browse to the **SynaptiCAD > Examples > TutorialFiles > TestBencherBasicTutorial** directory.
- Depending on your language type, select either the Verilog **clksram.v** or the VHDL **clksram.vhd** file.



- Press the **Open** button to close the dialog and add the file to the project.
- Double click on the **clksram** MUT file to open the file in an editor window. Glance through the code so that you have an of how the model we will be testing works. Close the editor when you are done.



```

clksram.v
module clksram(CLK,CSB,WRB,ABUS,DBUS);

input CLK;
input CSB;
input WRB;
input [7:0] ABUS;
inout [7:0] DBUS;

reg [7:0] DBUS_driver;
wire [7:0] DBUS = DBUS_driver;
reg [7:0] ram[0:255];

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

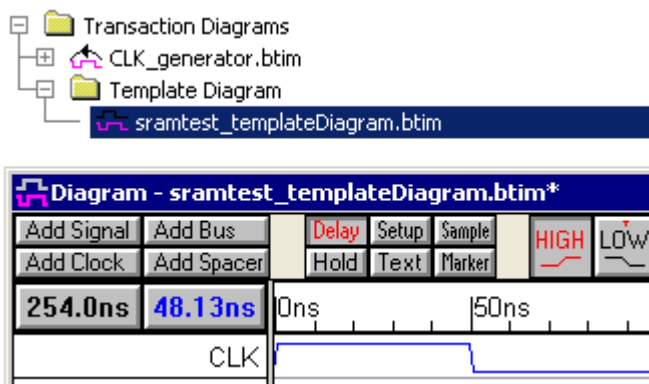
```

## (TBench) 3.4 Setup the Template Diagram

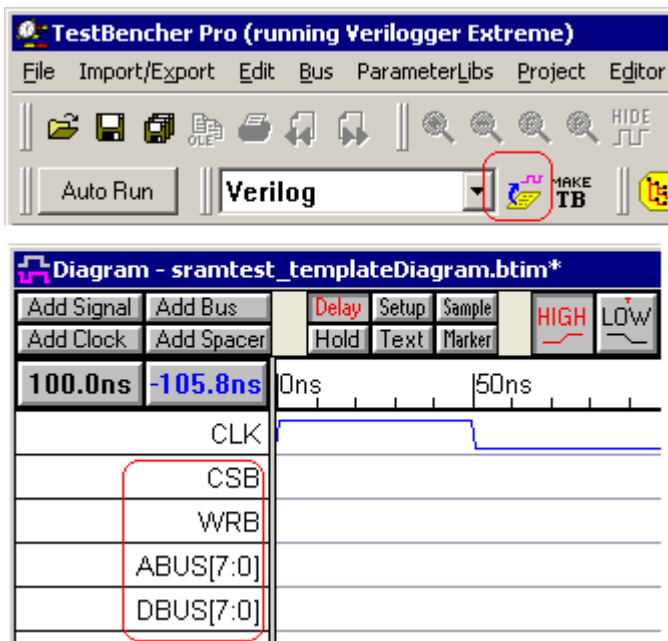
When TestBencher created the project it also generated a template diagram. New transaction diagrams that are created for this project will contain the same signals, waveforms, parameters, and properties as the template diagram. Currently the CLK signal is the only signal in the template diagram. You are going to add the port signals for the clocked SRAM to the template so that later when you create the timing diagrams project all of the signal information matches up.

**Extract the ports from the SRAM into the template diagram:**

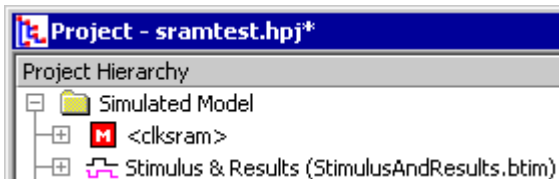
- In the *Project* window, under the *Template Diagram* folder, double click on **sramtest\_templateDiagram.btim** to open the template diagram window.



- Click the **Extract Ports from MUT** button. This will build the MUT and insert the signals for the MUT ports into the template diagram.



- Notice that **<clksram>** is now present in the *Project* window under the *Simulated Model* folder. The single angle brackets indicate that clksram is the Model Under Test. Expanding this tree will display signal, port, and component information of the MUT.



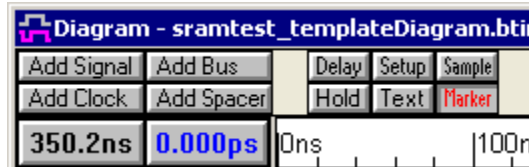
Note: If **<clksram>** was not generated as the MUT, then change the simulation preferences by

choosing the **Options > Diagram Simulation Preferences** menu. Check the **Auto-create test bench and tree** check box. Press the **Extract Ports from MUT** button to rebuild the MUT.

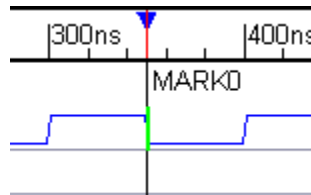
#### **Add an end diagram marker:**

The transaction diagrams use an End Diagram Marker to indicate the exact time that the transaction ends. You can add an end diagram marker to the template diagram, so all new transactions will get the marker.

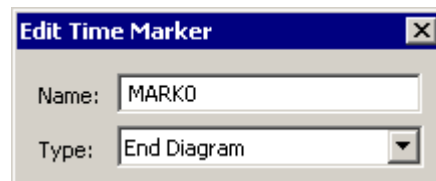
- Press the Marker button so that right clicks will add Marker lines to the diagram.



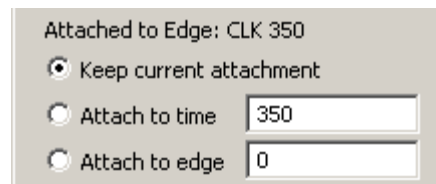
- Click on the fourth falling edge of the CLK signal (at 350ns) to select it and turn it green. Then right-click to add the marker line.



- Double-click on the marker to open the *Edit Time Marker* dialog.
- Select a Marker Type of **End Diagram** from the drop down list box. This end diagram marker will force the transaction to end at the fourth falling edge of the CLK signal.



- Notice that the Marker is **Attached to Edge** on **CLK** at 350ns. This is because you selected the edge before adding the marker. These controls can be used to changed the attachment.

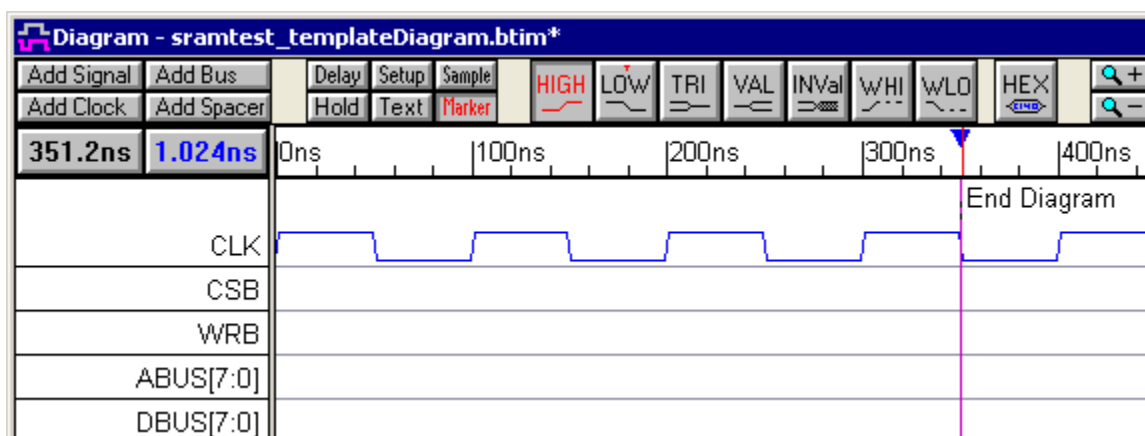


- Select **Type** from the *Display Label* list box. This will cause the marker to display its type rather than its name.



- Click **OK** to close the *Edit Time Marker* dialog.
- Use the **File > Save All Files** menu option to save the project and the template diagram.

The completed template diagram should look like the following:

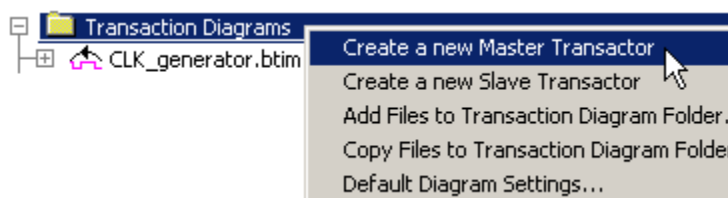


### (TBench) 3.5 Create the Write Cycle Transaction Diagram

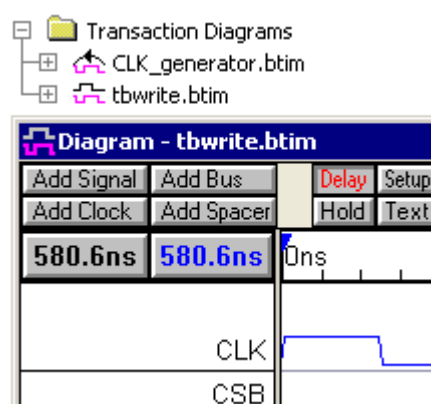
TestBencher Pro uses timing diagrams to represent reusable bus transactions. This tutorial will use two timing diagrams, **tbread.btim** and **tbwrite.btim**, to represent the read and write cycles used in testing the memory module. First, draw the write cycle diagram and then create variables for the data and address busses so that new values can be passed to the timing diagram each time it is called by the sequencer. Variables are also used to provide comparison values for runtime testing, and this will be demonstrated in the read diagram.

#### Create the Write diagram from the Template:

- In the *Project* window, right click the *Transaction Diagrams* folder and select **Create a new Master Transactor** from the context menu. This will cause the *Save As* dialog to open.

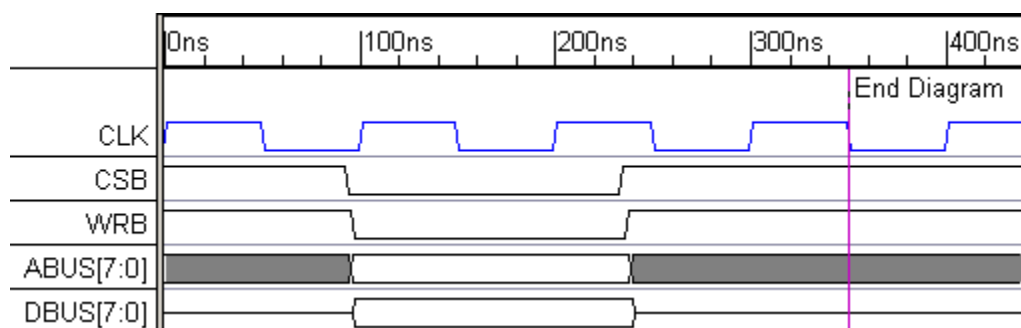


- Name the file **tbwrite** and press the **Save** button. This creates a new timing diagram using the information in the template file and lists the file in the *Transaction Diagram* folder.



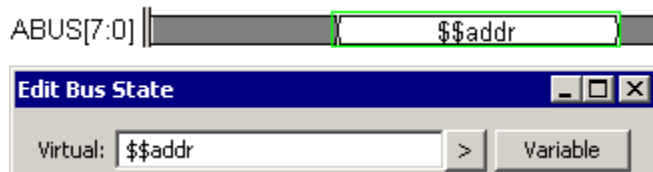
#### Draw the waveforms for the Write diagram:

Sketch the waveforms as shown on the diagram below. Since this is a negative edge clocked diagram the exact placement of signal edges is not important (unless it is near a negative clock edge). If you need a help drawing, refer to the Basic Drawing and Timing Analysis Tutorial sections **1.5 Drawing Signal Waveforms** and **1.6 Editing Signal Waveforms**.



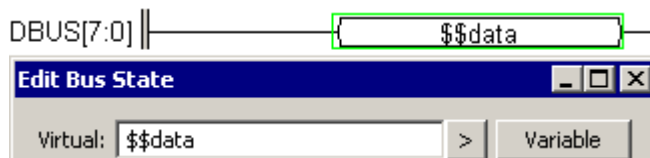
### Add the variables:


- Double click on the valid segment in the center of *ABUS* to open the *Edit Bus State* dialog.



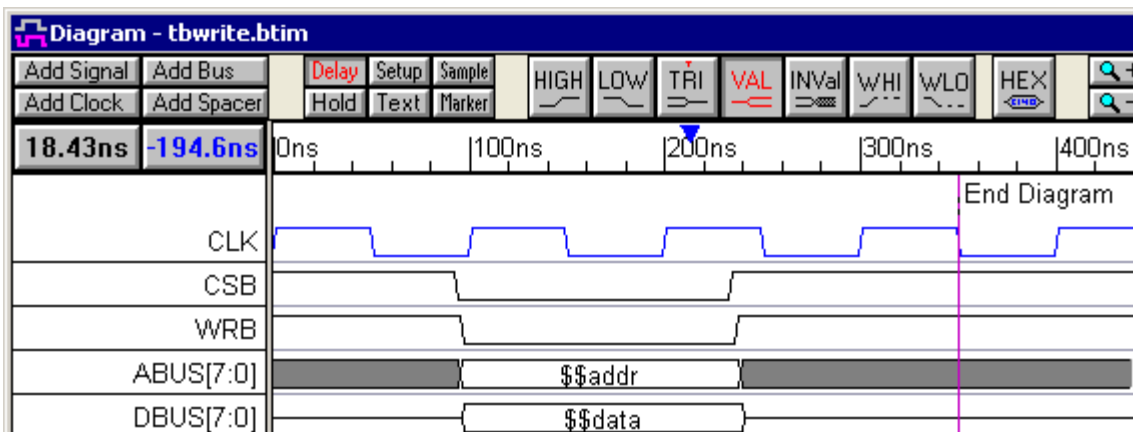
- Type **\$\$addr** into the **Virtual** edit box. The "\$\$" in front of the variable name indicates that this is a state variable. If the "\$\$" is missing, TestBench Pro will assume that this is the value of the address rather than a variable that will accept a value at a later time.

- Click on the valid segment in the center of *DBUS* to move the focus of the *Edit Bus State* dialog to the new segment.



- Type **\$\$data** in the **Virtual** edit box, then press the **OK** button to close the *Edit Bus State* dialog. The two edited segments will display the state variables.
- Click the diskette icon  on the main toolbar to save the timing diagram.

Below is the completed write transaction. When the chip select (CSB) and the write enable (WRB) are low, the address and data busses will be driven with the current values of the **\$\$addr** and **\$\$data** variables.





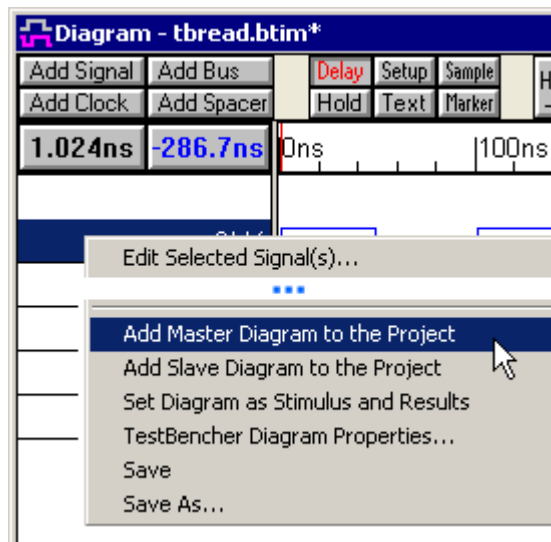
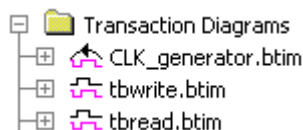
## (TBench) 3.6 Create the Read Cycle Transaction Diagram

The read cycle will initiate a read transaction with the clocked SRAM and also monitor the data bus to verify the result of the read. Since the signals for the read diagram are so similar to the write diagram, we will start by copying the write diagram and then modify the waveforms (instead of starting with the template).

### Copy the *tbwrite* diagram to make *tbread*:

- Click on the **tbwrite** diagram window bar so that it is the active window, then choose **File > Timing Diagram As** menu to open the Save As dialog
- Name the file **tbread**, and press the **Save** button.

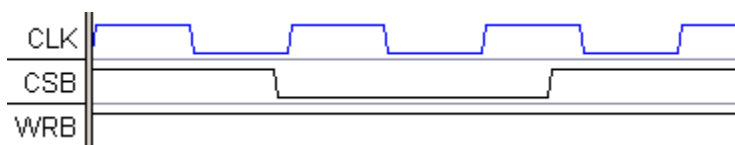
- Right-click in *tbread*'s Label window, and select **Add Master Diagram to Project** from the context menu. This will add *tbread* to the *Transaction Diagrams* folder in the *Project* window.



### Modify the WRB Signal:

The write control signal, WRB, should stay high (inactive) for the duration of the read.

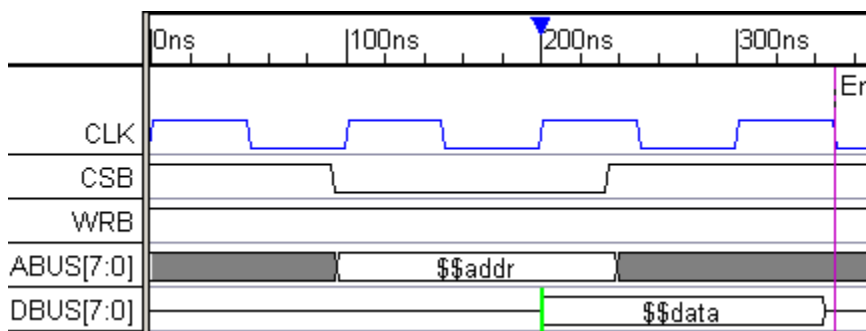
- Select the center segment and press the delete key to remove the low signal segment.



### Modify the DBUS Signal:

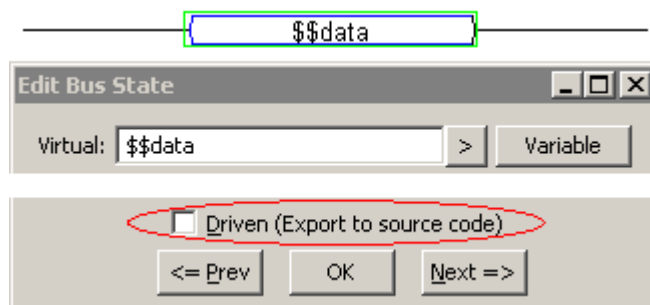
Since our SRAM is clocked the data comes out on the clock cycle after the chip select signal, CSB, goes active. You can drag each edge of DBUS individually, or use the following technique to shift the whole signal.

- Shift the start of the DBUS data segment to 200ns. Hold down the <2> key (the number 2 key) on the keyboard, while dragging the starting transition to 200ns. The <2> key causes transitions to the right of the selected edge to move with the dragged edge.



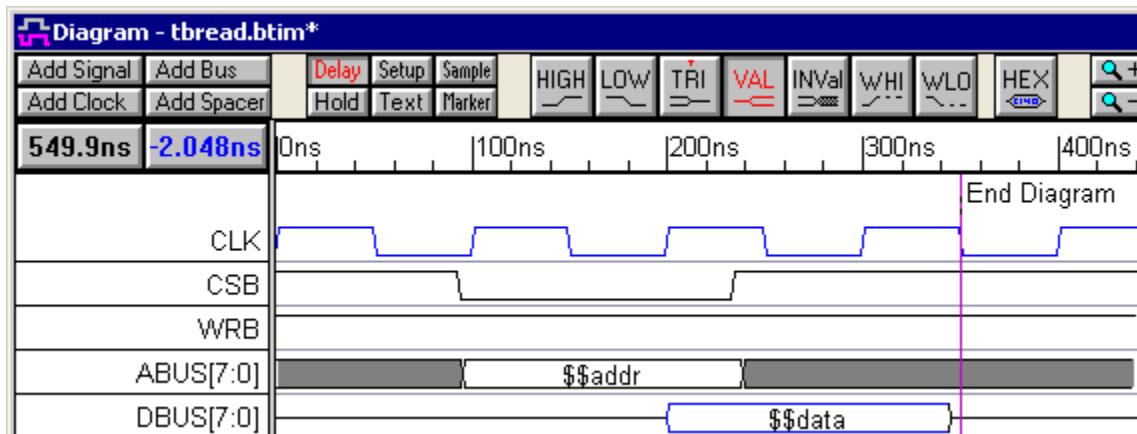
During the read transaction, the DBUS signal will be driven by the SRAM so it will be an input signal to the test bench (not an output like in the write cycle).

- Double click on the data segment to open the *Edit Bus State* dialog.
- Uncheck **Driven (Export to source code)** checkbox. This will cause the segment to be displayed in blue.



- Click the diskette icon  on the main toolbar to save the timing diagram.

The completed read diagram looks like the following:



### (TBench) 3.7 Add a Sample to Verify Data

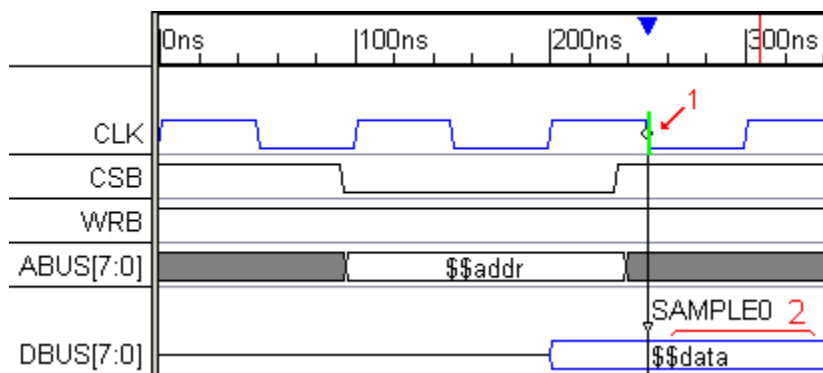
Next a Sample will be added to the read timing diagram. Samples compare the actual state value of an input signal to the expected state value, and conditionally react to the results of the comparison.

**Add the sample:**

- In the **tbread** diagram, press the **Sample** button so that right clicks will add delays.



- First, left-click on the **third falling edge** (250ns) of **CLK** to select the edge.
- Then, Right-click near the end of the **blue valid segment** on **DBUS**.

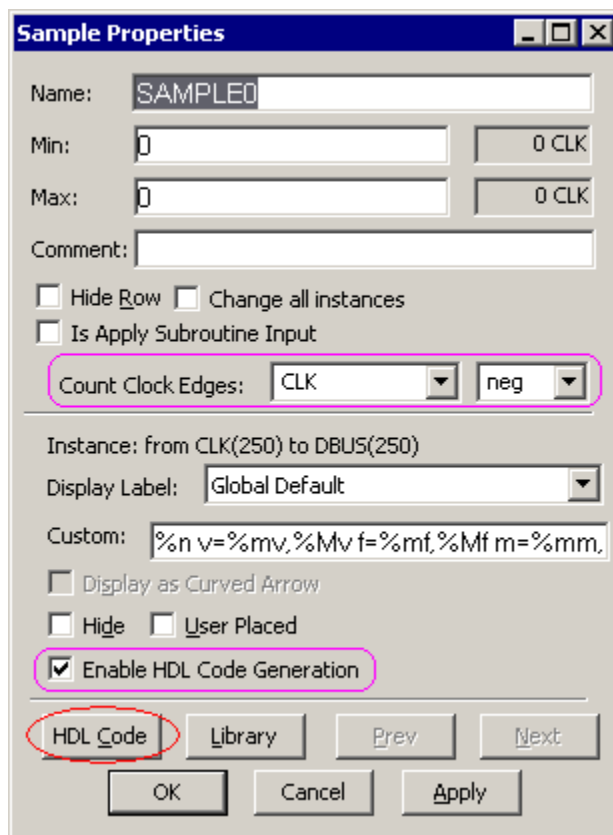


- This adds a Sample parameter named **SAMPLE0** that lines up with the third neg edge of the CLK signal. Refer the image in the previous section.

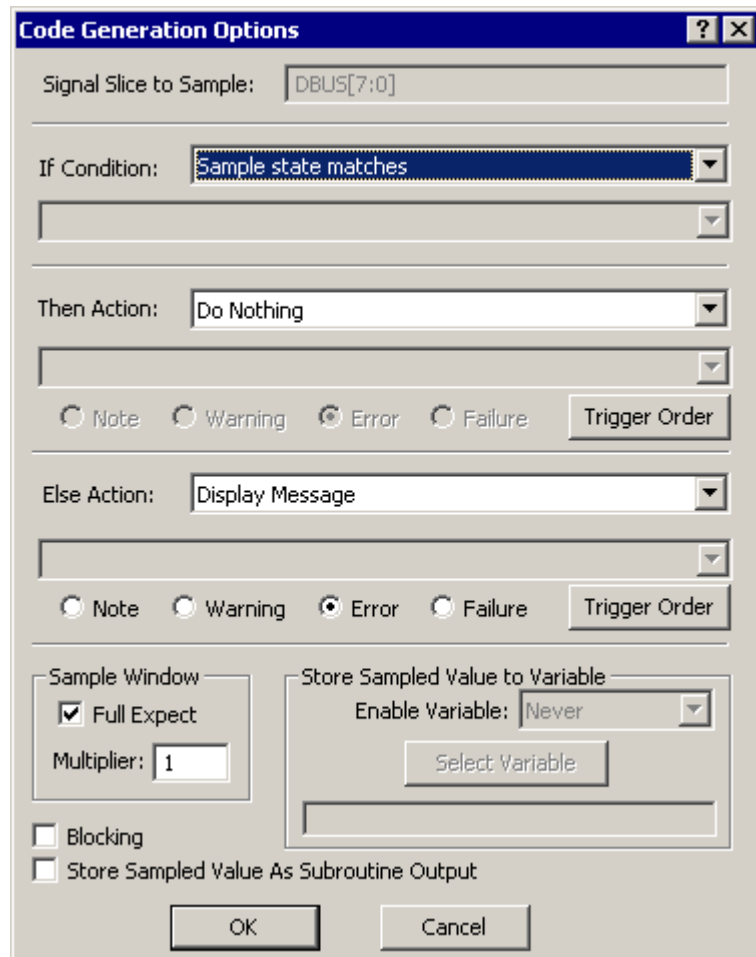
#### Investigate the sample code generation features:

The default behavior of the sample compares the run time value with the drawn value (\$\$data) and throws an *Error* if they are different. This is the behavior that we need for the tutorial. The next few steps show you the HDL code generation dialog and how to control the generated code. You do not need to make any changes to the dialog defaults.

- Double-click on the sample name **SAMPLE0** in the drawing window to open the *Sample Properties* dialog.
- Notice that this dialog controls all of the display feature for the Sample. The sample can be offset from the triggering edge. Also notice that the HDL Code Generation is enabled.
- Press the **HDL Code** button to open the *Code Generation Options* dialog.



- The Sample generates an code in the form of if/then/else.
- Look through the *condition* and *action* drop-downs to see the built-in behavior choices.
- Make sure to return the dialog to the shown state so that error messages will be generated if the Sampled data does not match the variable.



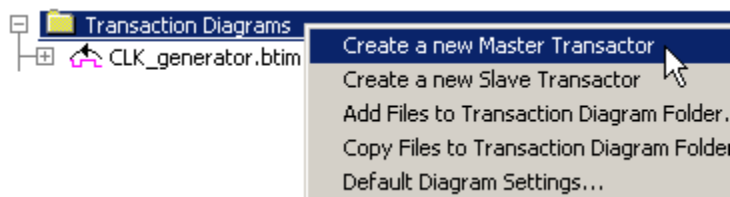
- Press **OK** to close the *Code Generation Options* dialog. Then press **OK** to close the *Sample Properties* dialog.
- Save the timing diagram by selecting **File > Save Timing Diagram** from the main TestBench menu.

### (TBench) 3.8 Create the Initialize Transaction Diagram

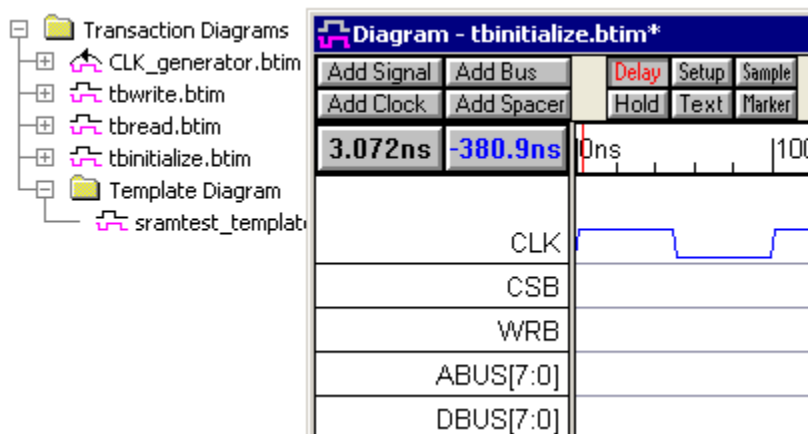
When drawing the waveforms for a transaction diagram it is important to remember that transactions do not automatically include an event at time zero and that only the drawn events are driven. This is a feature that allows transactions to be reused any time during simulation without implying any initialization information. In our example the clocked SRAM control signals, CSB and WRB, need to be initialized before the read and write cycles are applied to the model. We will draw a simple initialization diagram that will drive the control signals to high (inactive).

### Create the Initialization diagram from the Template diagram:

- In the *Project* window, right click the *Transaction Diagrams* folder and select **Create a new Master Transactor** from the context menu. This will cause the *Save As* dialog to open.

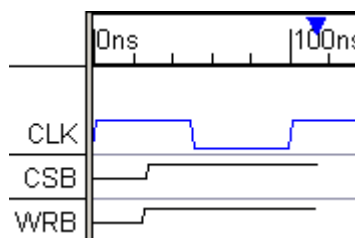


- Name the file **tbinitialize** and press the **Save** button. This creates a new timing diagram using the information in the template file and lists the file in the *Transaction Diagrams* folder.



### Edit the Waveforms:

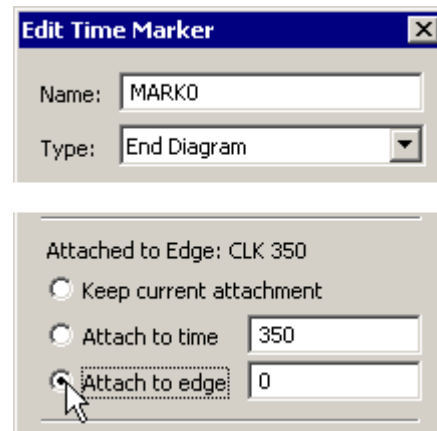
- Remove the **ABUS** and **DBUS** signals, because the tri-state bus signals do not need to be initialized. Select the **ABUS** and **DBUS** signals by clicking on them, and then press the **<delete>** key to delete the selected signals.
- Draw the following waveforms as shown. If you need a help drawing, refer to the Basic Drawing and Timing Analysis Tutorial sections **1.5 Drawing Signal Waveforms**.



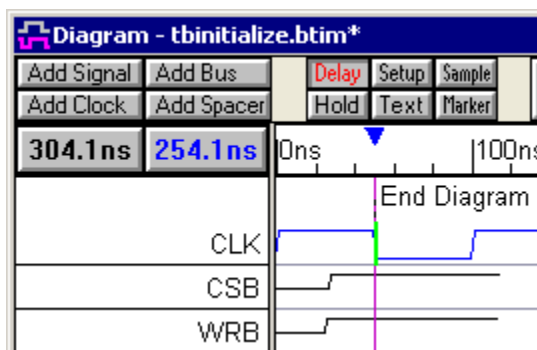
### Move the End Diagram Marker:

The initialization timing diagram will only need one clock cycle to initialize the control signals. Therefore, the **End Diagram** marker can be moved to the 1st negative clock edge.

- Double-click on the marker to open the *Edit Time Marker* dialog.
- Select **Attach to Edge** from the radio buttons.
- Click **OK** to close the *Edit Time Marker* dialog. This will put TestBencher into a special select mode.



- As you move the cursor around in the diagram a green bar will jump to the closest edge to remind you that you are in the Attach to edge mode. Click on the first negative clock edge (at 50ns) to attach the marker to that edge.



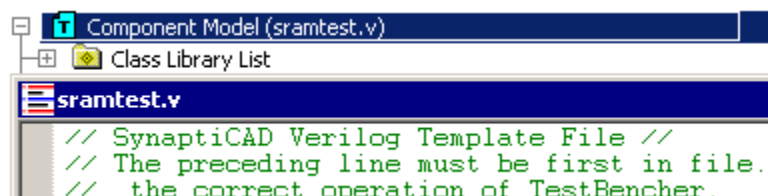
- Click the diskette icon  on the main toolbar to save the timing diagram.

## (TBench) 3.9 Add Transaction Calls to the Sequencer Process

Inside the primary template file for the project is a *Sequencer Process*. This process is the place in the top-level test bench that defines the order in which the timing transactions are applied to the model under test. By using the *Insert Diagram Calls* dialog, you can construct the testbench by writing little to no code.

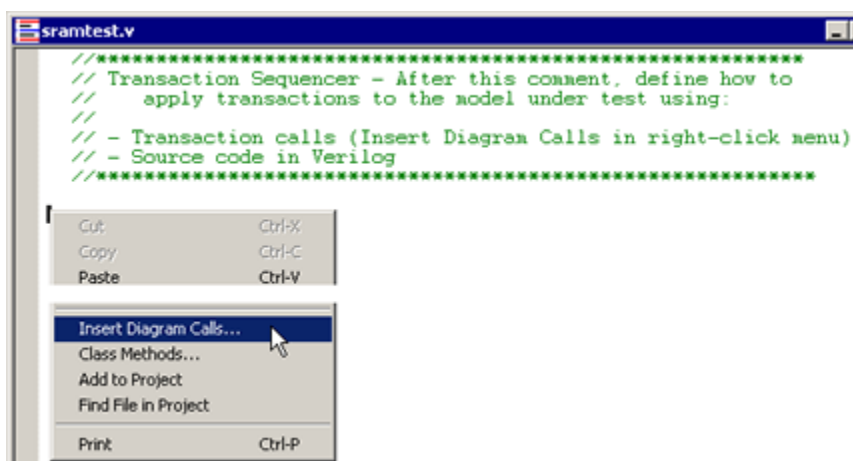
### Open the Component Model for the main Test Bench:

- In the *Project* window, double click on the *Component Model* folder to open an editor window with the **sramtest** template file.



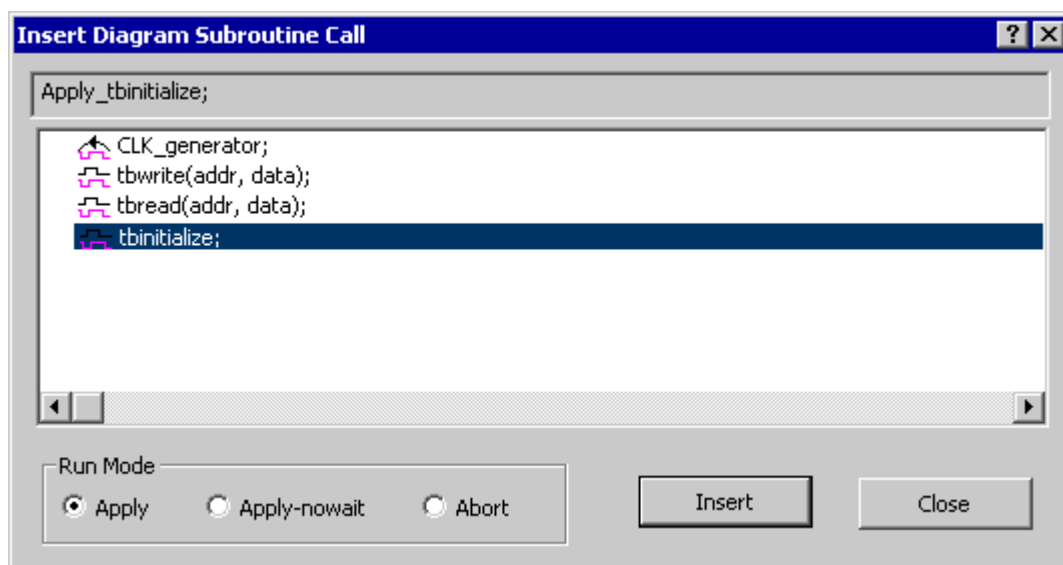
### Find the Transaction Sequencer in the Component Model:

- Scroll down in the **sramtest** editor window near the end of the file until you find the **Transaction Sequencer** comment block. The comment changes depending on the code generation language.



#### Open the Insert Diagram Calls Dialog:

- Click in the **sramtest** editor window below the Transaction Sequencer comment (as shown above) so that the blinking cursor is in the place where the apply statement should be added.
- Right-click in the editor window and select **Insert Diagram Calls** to open the *Insert Diagram Subroutine Call* dialog.

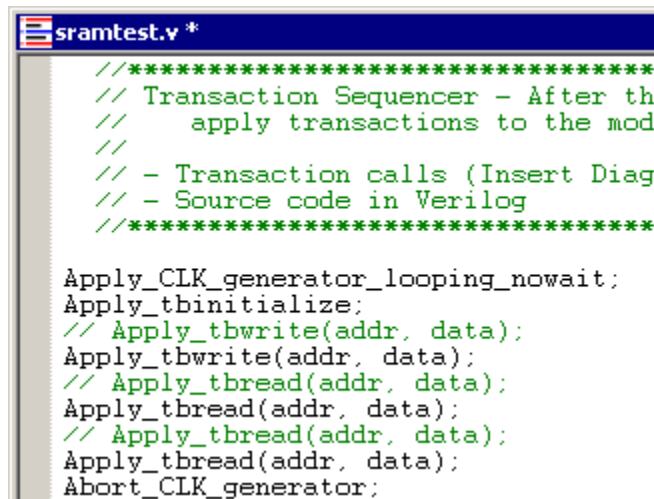


- Arrange the windows so you can see the editor and the dialog at the same time.

#### Insert the Apply calls:

The *Insert Diagram Subroutine Calls* dialog generates diagram apply calls so you do not need to memorize the function syntax. Each timing diagram can generate one of three task calls: Apply, Apply-nowait, and Abort. Apply runs the transaction in a blocking mode, and Apply-nowait runs the transaction concurrently with other transactions. The Master/Slave *Diagram Setting* determines how many times a transaction executes. Master Transactors, like the Read, Write, and Initialize diagrams run once and stop. Slave Transactors like the Global Clock Generator run in a looping mode until an Abort call is received. You will first start the clock, initialize the control signals, write to the SRAM, the read from the SRAM twice, and then abort the clock.

- Double click on the **CLK\_generator** insert the statement into the Sequencer Process. Since this is a slave diagram (indicated by the black arrow), the default state is **Apply-nowait**, because most of the time slave diagrams will run concurrently with other diagrams.
- Double click on the **tbinitialize** entry. Since this is a master diagram, the default state is **Apply**, because usually Master diagram run in blocking mode.
- Double click on the **tbwrite** entry.
- Double click on the **tbread** entry TWO times to insert the code to add two read calls.
- Select **CLK\_generator** entry, choose **Abort** radio button, and then press the **Insert** button to insert the code. This will add the abort call to stop the clock generator.
- The apply call should look similar to the following code block. Different languages may have extra parameters.



```

//*****
// Transaction Sequencer - After th
//   apply transactions to the mod
//
// - Transaction calls (Insert Diag
// - Source code in Verilog
//*****

Apply_CLK_generator_looping_nowait;
Apply_tbinitialize;
// Apply_tbwrite(addr, data);
Apply_tbwrite(addr, data);
// Apply_tbread(addr, data);
Apply_tbread(addr, data);
// Apply_tbread(addr, data);
Apply_tbread(addr, data);
Abort_CLK_generator;

```

#### **Edit the State Values of the Write and Read Apply calls:**

Edit the write and read Apply code lines and replace the state variable names with actual variables that will be passed into the timing diagrams. The comment lines are there to document the parameter variable names. **Note:** The code to be entered is **bold**.

- For Verilog type:

```

Apply_tbwrite('hF0, 'hAE);
Apply_tbread('hF0, 'hAE);
Apply_tbread('hF0, 'hEE);

```

- For VHDL type:

```

Apply_tbwrite(tb_Control, tb_InstancePath, x"F0", x"AE");
Apply_tbread(tb_Control, tb_InstancePath, x"F0", x"AE");
Apply_tbread(tb_Control, tb_InstancePath, x"F0", x"EE");

```

- For OpenVera type:

```

tb_tbwrite.ExecuteOnce('hF0, 'hAE);
tb_tbread.ExecuteOnce('hF0, 'hAE);
tb_tbread.ExecuteOnce('hF0, 'hEE);

```

Notice that the **tbwrite** apply statement writes the hex value AE to memory cell F0. The **tbread** diagram calls will then read the value from the same memory cell. The data values provided in the **tbread** diagram calls will be used to compare with the actual value. The first call to **tbread** will expect to find a value of hex AE in the address F0. The second call to **tbread** will expect to find the



hex value EE instead. This will cause the sample to report an error during the second execution of **tbread**.

- Save the top-level template file by right-clicking in the editor window and selecting **Save**.

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

An alternative method to placing transaction calls in the sequencer process is to create a file external to the bus-functional model with transaction calls and during simulation read the transaction calls from a file (see *Section 9.4: Transaction Manager and Test Reader* in the online TestBencher Manual).

## (TBench) 3.10 Setup the Simulator

At this point all the timing diagrams have been created and you have edited the Sequencer process. Next we will generate the test bench and simulate the entire design, but we should first check to see if the simulator is setup properly.

TestBencher can control external simulators and compilers or use its built-in Verilog simulator to compile and simulate the design. If you are using the built-in simulator, skip ahead to next section. *Section 10.3: External Program Integration* in the online manual has a complete list of instructions for working with remote simulators and for setting up a compiler for TestBuilder.

### *To configure a third-party simulator:*

1. Choose the **Options > Simulator and Compiler Settings** menu option. This will open the *Simulator and Compiler Settings* dialog.
2. From the **Simulator and Compiler tools** dropdown select the appropriate simulator.
3. Enter the directory that contains the simulator executable in the **Simulator Path** edit box.
4. Press **Compile Syncad Libraries** to build libraries required by the simulator in order to compile TestBencher projects. **IMPORTANT:** If you omit this step, you will get compile errors when you attempt to compile your test bench source files.
5. Click **OK** to close the *Simulator and Compiler Settings* dialog.


### *Select the third-party simulator:*

1. Select the **Project > Project Settings** menu option. This will open the *Project Settings* dialog.
2. Select the tab for the language you are working with.
3. Select the desired simulator from the **Simulator Type** dropdown.
4. Click **OK** to close the *Project Settings* dialog.

## (TBench) 3.11 Generate the Test Bench and Simulate



Once the simulator is setup you are ready to generate the test bench and simulate the design.

### *To generate the test bench:*

- Click on the **Make TB** button  on the main TestBencher toolbar. This will expand the macros in the template file and pop up a dialog that says "Finished generating test bench. Please check waveperl.log for errors." Close this dialog by clicking the **OK** button.
- In the *Report* window, check the **waveperl.log** tab to see if TestBencher encountered any errors during the test bench generation. If it did, fix the error and regenerate the test bench. (If you can not see the *Report* window, choose the **Window > Report** menu to bring it to the

front.)

### To simulate the design:

- Click the yellow **Compile Model and Test Bench**  button. This builds (parses) the project using the tools specified in the *Project Settings* and *Simulator and Compiler Settings* dialogs.
  - In the bottom right corner, a yellow **Simulation Built** status message indicates the build was successful and that you are ready to simulate.
  - If the status indicates an error, the *Report* window *Errors* tab displays the compile errors. If there are errors then fix them, regenerate the test bench, and recompile.
- Click the green run button  on the simulation button bar. This will simulate the design and display the results in the *StimulusAndResults* diagram and the *Report* window *simulation.log* tab.
  - In the bottom right corner, a **Simulation Good** status message indicates that the simulation has reached a successful end.

## (TBench) 3.12 Examine Report Window Results

The *Report* window **simulation.log** tab displays the default log file for the simulator. TestBench automatically writes a message to the log file each time a transaction starts and stops. The clocked SRAM contains code to display a message each time it performs a read or write. We also added a sample parameter to the Read Cycle, and set it to generate an error message when the data from the SRAM does not match the expected value.

Examine the log file:

- In the *Report* window, open the **simulation.log** tab and display the following results:

```
Running...
TB> Note: In "sramtest_CLK_generator" at 0.000ns: Executing LOOPING
TB> Note: In "sramtest_tbinitialize" at 0.000ns: Executing ONCE
TB> Note: In "sramtest_tbinitialize" at 50.000ns: Execution DONE
TB> Note: In "sramtest_tbwrite" at 50.000ns: Executing ONCE
In clksram at 150.000ns: Writing ae to address f0
TB> Note: In "sramtest_tbwrite" at 350.000ns: Execution DONE
TB> Note: In "sramtest_tbread" at 350.000ns: Executing ONCE
In clksram at 450.000ns: Reading ae to address f0
TB> Note: In "sramtest_tbread" at 650.000ns: Execution DONE
TB> Note: In "sramtest_tbread" at 650.000ns: Executing ONCE
In clksram at 750.000ns: Reading ae to address f0
TB> Error: In "sramtest_tbread" at 850.000ns: Sample SAMPLE0_process sampled
        signal: DBUS expected: ee ; detected: ae
TB> Note: In "sramtest_tbread" at 950.000ns: Execution DONE
TB> Note: In "sramtest_CLK_generator" at 950.000ns: Execution DONE
0 Errors, 0 Warnings
Compile time = 0.01000, Load time = 0.02000, Execution time = 0.05000

Normal exit
```

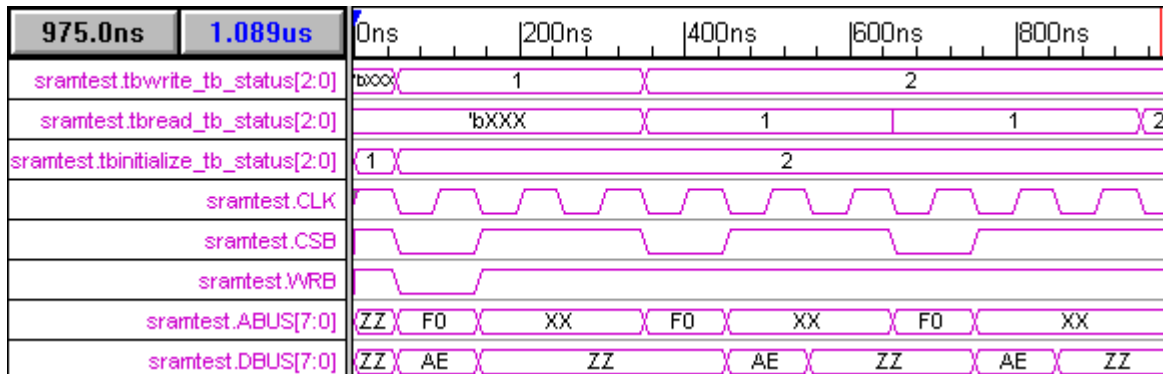
- Notice that the clock generator starts at time zero and continues until the end of the simulation when the abort call is issued.

- The initialization diagram also starts executing at time zero and blocks the next transaction until it is complete.
- The write diagram starts next and writes a value to the SRAM. The SRAM acknowledges that it is writing the value to the specified address.
- The first read diagram executes successfully.
- The second read diagram throws a warning because the expected value did not match the value from the MUT. We purposely passed in a bad expected data value so we could see how the sample throws the error.
- Next the abort call to the clock stops the clock transaction and ends the simulation.

### (TBench) 3.13 Examine the Stimulus and Results Diagram

After simulation the Stimulus and Results diagram will contain all of the top level signals of the project, the driver signals, and status and trigger signals for each transaction.

- Hide some of the signals in the Stimulus and Results diagram by selecting the signal names and choosing **View > Hide Selected Signals** until the diagram looks like this:



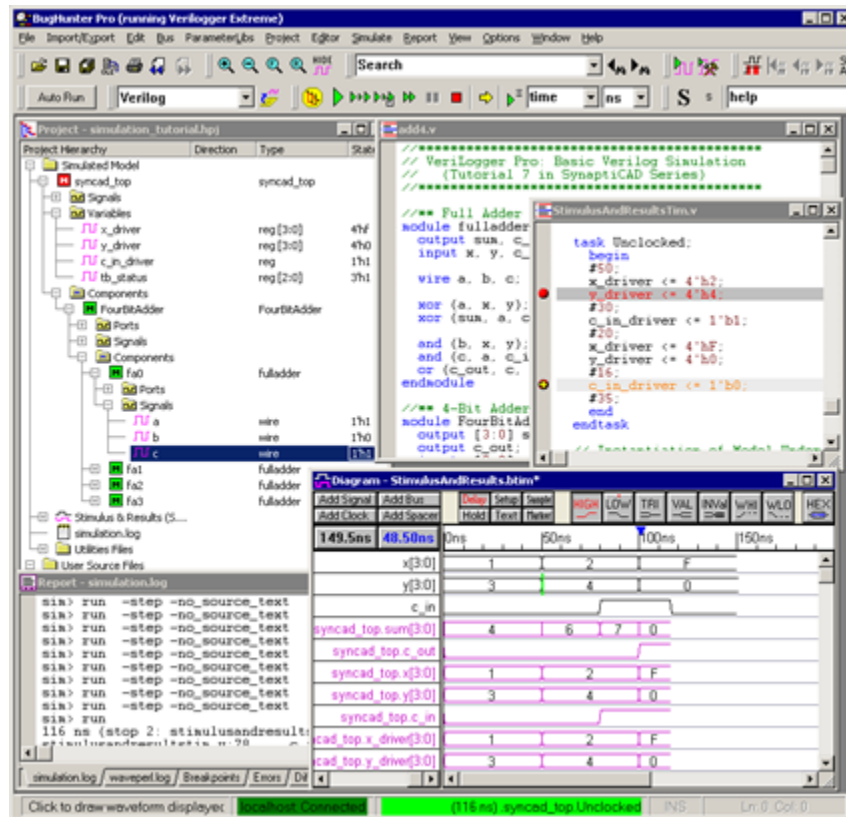
- A status signal of <1> indicates the transaction is running. You can see that the initialization diagram runs followed by the write cycle and two read cycles.
- During the write cycle, the data AE is written to address F0. When comparing the simulated write cycle to the drawn transaction, remember that this is a negative clock edge diagram.
- The read cycles read back the data from the memory.

### (TBench) 3.14 TestBencher Pro Basic Tutorial Summary

Congratulations! You have completed the TestBencher Pro Basic tutorial. In just a few minutes you created four transaction diagrams, modified the sequencer process, and generated a full bus functional model. This is a very basic example of how TestBencher Pro can work. More advanced features include the ability to randomly generate values for transactions or read them in from files. You also extend TestBencher's capabilities by writing functions and calling them within a transaction diagram. The next step is to read through the first page of each of the TestBencher Pro Manual's chapters to see the types of functions that are available.

## Simulation 1: VeriLogger Basic Verilog Simulation

This tutorial demonstrates the basic simulation features of the VeriLogger simulators (simx and vlogcmd) and the graphical debugger (BugHunter Pro). It teaches you how to create and manage a project and how to build, simulate, and debug your design. It also demonstrates the graphical test bench generation features that are unique to BugHunter Pro.



This is a stand alone tutorial which you should be able to complete without reading any of the other tutorials. However, if you plan to make extensive use of the graphical stimulus generation features then you may also want to perform the Basic Drawing and Timing Analysis tutorial which covers more of the drawing features of the timing diagram editor.

### (Sim) 1.1 Simulator Choices

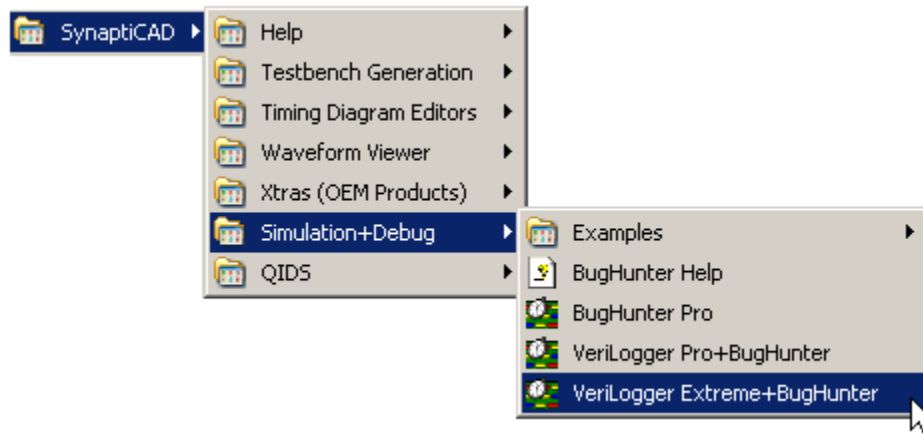
SynptiCAD offers two different Verilog simulators: VeriLogger Extreme (simx) and VeriLogger Pro (vlogcmd). VeriLogger Extreme is a high-performance compiled-code Verilog 2001 simulator that offers fast simulation of both RTL and gate-level simulations with SDF timing information. VeriLogger Pro is an interpreted Verilog-95 compliant simulator with a low memory footprint, but it does not support strengths.

VeriLogger Extreme is the faster simulator for large designs and simulating in **debug run mode** (the standard mode for simulators). However, since VeriLogger Pro is interpreted and does not need to compile the code, it is faster for smaller designs and for **auto run mode** where the user is graphically changing the test bench and kicking off automatic simulations.

When you purchase a simulator, you get SynptiCAD's graphical debugger, BugHunter Pro, in addition to the command line version of the simulator. Instructions for running the command line versions are found in the **BugHunter and VeriLogger Manual Chapter 5: Command Line**

*Simulators.* For this tutorial, run BugHunter so that you can experiment with the graphical debugging interface.

- From the Start Menu, choose one of the VeriLogger simulators running under BugHunter.



- An alternative way to launch the simulators is using the command line:
  - VeriLogger Extreme: `syncad -p bhp -s verilogger_extreme`
  - VeriLogger Pro: `syncad -p bhp -s vlogcmd`

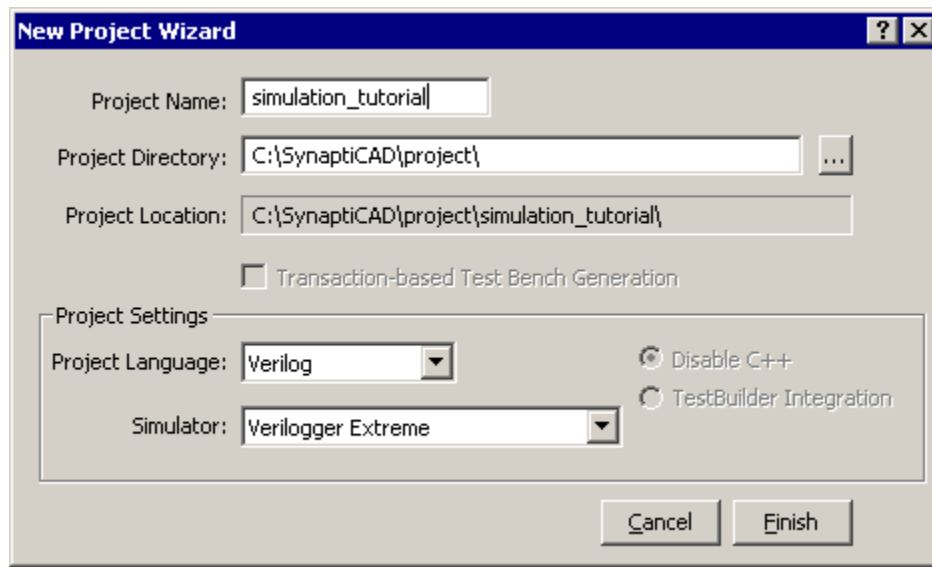
The `-p bhp` option says to run the BugHunter Pro product. The `-s` option sets the default simulator used for new BugHunter projects. Note that the simulator used by a project can be changed at any time from inside BugHunter by selecting the **Project > Project Simulation Properties** menu option and changing the the **Simulator Type** under the **Verilog** tab in the dialog that appears.

## (Sim) 1.2 Add Files to the Project

BugHunter uses a project file to list the files to be simulated and store the simulation options. Here you will create the project file and investigate the source code that is used in the tutorial.

### *Create a new Project:*

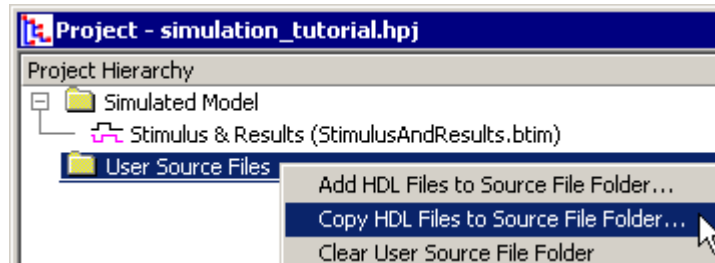
- Select the **Project > New Project** menu to open the *New Project Wizard* dialog.
- In the **Project Name** box, name the project **simulation\_tutorial**. Notice that the wizard is also creating a directory of the same name as you type in the project name.
- Notice that you can select the simulator used by the project by changing the selection in the **Simulator** drop down box at the bottom of this dialog.



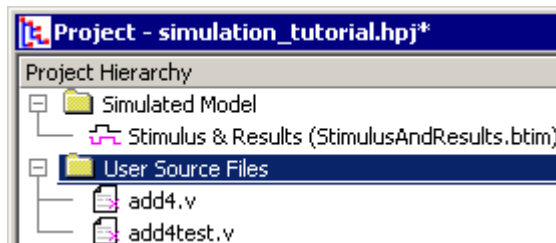
### Add the Source Files to the Project:

If you were starting a design from scratch, you could use the **Editor > Open HDL File** to open an editor window and type in the code, and then **add** the new file to the project. In this tutorial, we will just copy existing source files from an examples directory.

- Right click on the **User Source Files** folder and select **Copy HDL files to Source File Folder** from the context menu to open a file dialog.



- In the file dialog, browse to the **SynaptiCAD\Examples\TutorialFiles\VeriloggerBasicVerilogSimulation** directory and select the **add4.v** and **add4test.v** files. To select multiple files, hold down the **<CTRL>** key while selecting files with the left mouse button. Then close the file dialog.
- When the files are first added to the project they will be marked with a purple x to show that the files are not compiled.



### Investigate the code for the Tutorial:

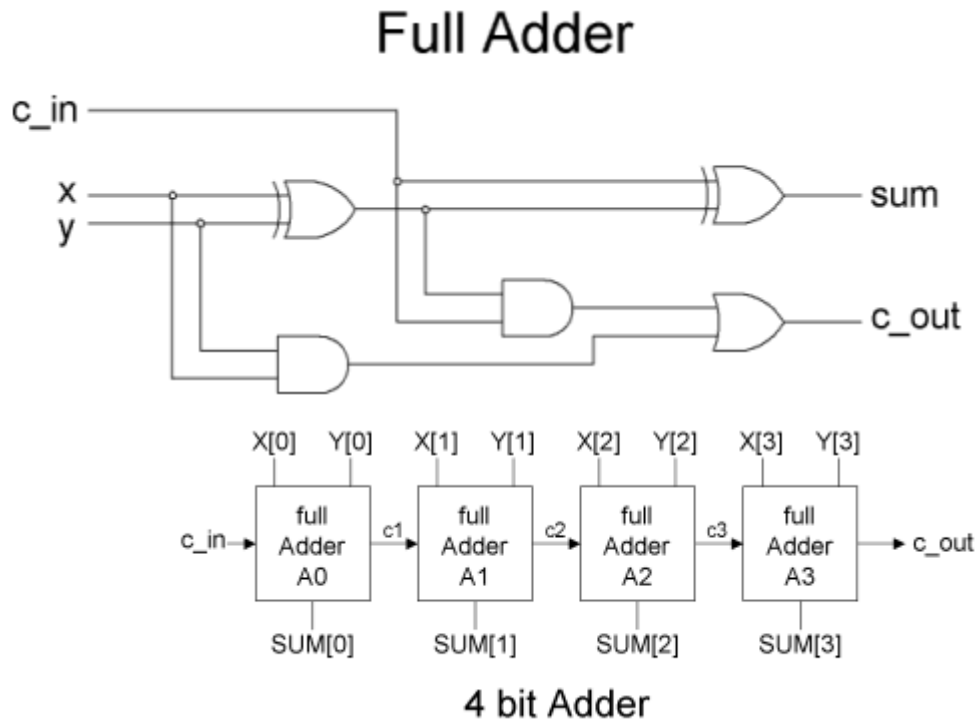
- In the project window, double click on **add4.v** to launch an editor window with the source code loaded.

```

add4.v
/** Full Adder *****/
module fulladder(sum, c_out
    output sum, c_out;
    input x, y, c_in;

```


- Look through the code and compare it to the following schematics. We will be simulating a 4-bit adder circuit which adds the **x** and **y** inputs together and outputs the answer on the **sum** and **c\_out** lines.

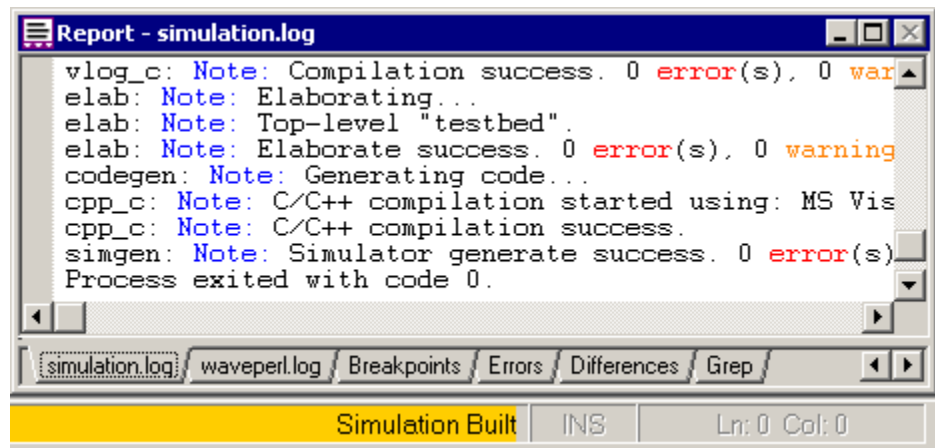


### (Sim) 1.3 Build the Tree and Investigate the Project

In this section we will build the project tree and use the tree to view the internal modules.

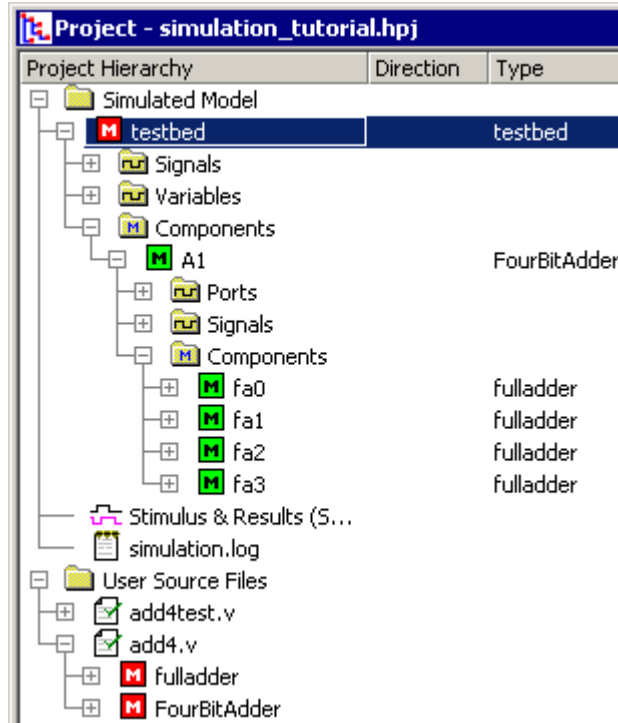
#### Three ways to build a project:

- Press the yellow **Build** button on the simulation bar, select the **Simulate > Build** menu, or press the **<F7>** key. 
- When the build is successful, a yellow **Simulation Built** message will display in the lower right hand corner of the program.
- The **Report** window **Simulation Log** tab shows the build results, and the **Error** tab would display a hyper-linked list of errors if there were any.



### Setting Top-level instances after the first Build:

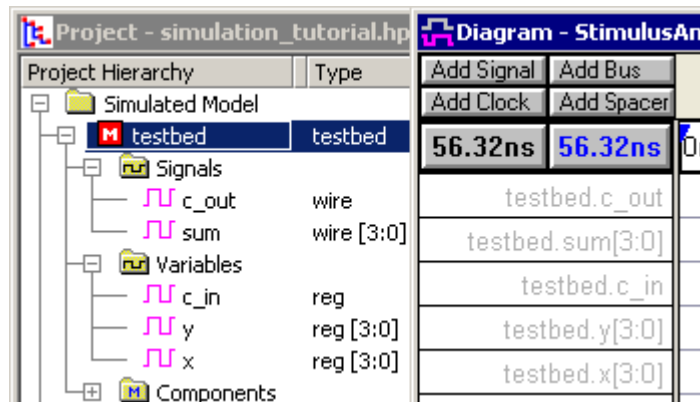
- By default, BugHunter identified **testbed** as the only top-level instance, because all other modules are instantiated under it. For this tutorial, this is the desired top level instance.
- After the first build, you can optionally right click on other modules and choose **Set as a Top Level instance** to force instantiation as top level instances.
- 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.



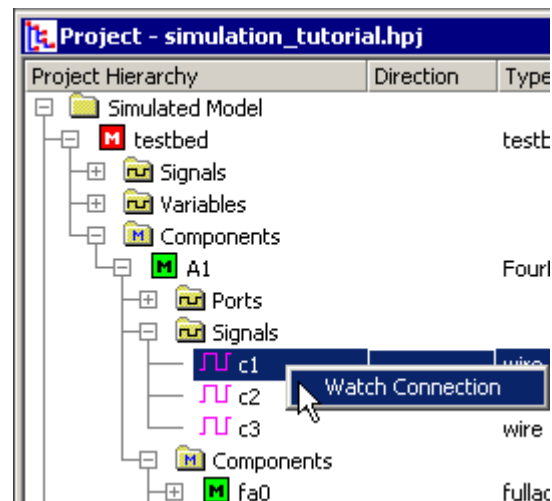
### Set Watch Signals after the first build:



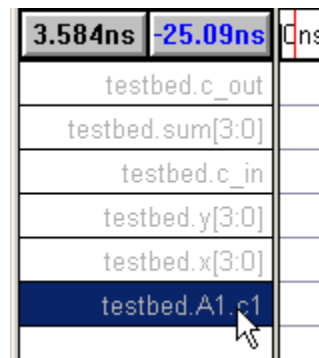
- BugHunter by default sets *watches* on all the signals and variables of the top-level module. This means that the signal names are displayed in the **Stimulus and Results** diagram and the waveforms will be displayed during simulation. Event history is only maintained for these signals.



- Open the project tree until you find **component A1** and **signal C1**, then right click and choose the **watch** menu option. This causes C1 to be added to the **Stimulus and Results** diagram.
- You can also set watches on entire components, blocks or variables using the same technique.



- Remove any extra watch signals that you added in the last step. First left click on the signal name in the diagram window to select it, then press the **<delete>** key.



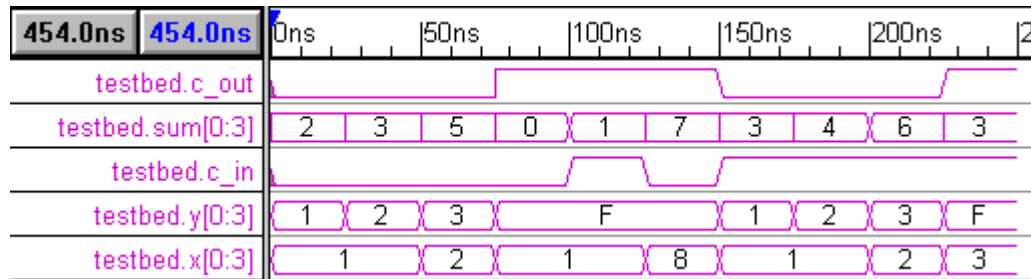
## (Sim) 1.4 Simulate the Project

The green buttons on the Simulation button bar cause the simulator to run. The smaller buttons are for single stepping through the code. The one with the hourglass will simulate for a specified period of time.



### Simulate the project:

- Press the large green run button to simulate until the end of the simulation or until a breakpoint is encountered.
- Notice that the **Stimulus and Results** diagram has displayed the simulation results. Verify that **sum** and **c\_out** are correctly being computed as  $x + y + c_{in}$ .



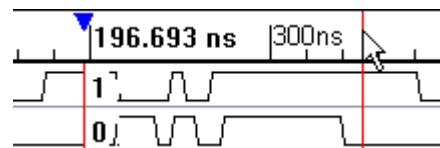
- If there were errors in the code, they would be indicated by the status bar in the lower right hand corner, and listed in the *Report* window's **Error** tab.

For the rest of this section, just play around with the zooming, scrolling and searching capabilities of the waveform window.

### Zoom using buttons or the mouse:

- To zoom in and out quickly, hold down the **<Shift>** key while using the **scroll wheel** on your mouse.
- To zoom in over a visible section, drag and drop inside the **Time Line**.
- The zoom buttons are located on the main window below the menu bar and on the diagram window button bar.
- The zoom in (+) and zoom out (-) center the zoom on the selected item, the blue delta mark, or the center of the diagram in that order.

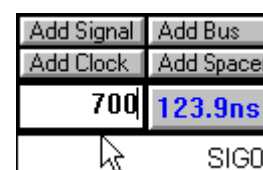
**<Shift>** and mouse **scroll wheel**



- The zoom full (F) displays the entire timing diagram on the screen.
- The zoom range (R) opens a dialog that lets you specify the starting and ending times for the zoom.

### Scroll to a specific or relative time using the Time or Delta buttons:

- Press on either the **Time** or the **Delta** button to open an edit box, and type in a time. The Time button (black) causes the diagram window to scroll to that exact time. The Delta button (blue) causes the diagram window to scroll that amount from its current position.



**Search for a specific signal name, parameter name or string:**

- Select one of the child windows in the program, then type into the **Search** box on the main window bar.



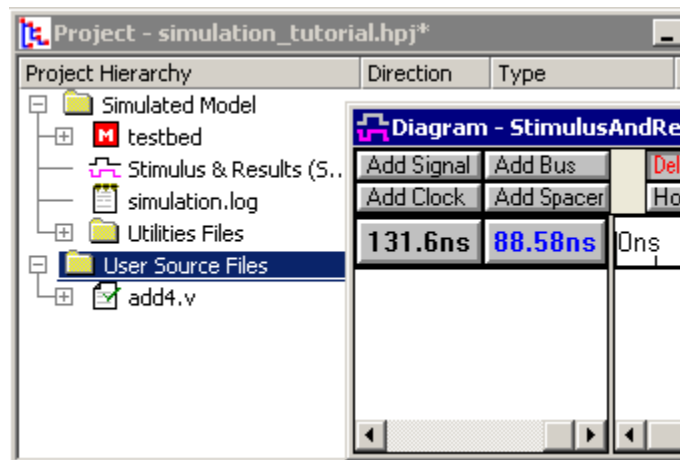
- If the Diagram window is selected, then it will search for signal names.
- If the Parameter window is selected, then it will search for parameter names.
- If the Report window is selected, then it will search for text in the selected report tab.

**(Sim) 1.5 Prepare for Graphical Test Bench Generation**

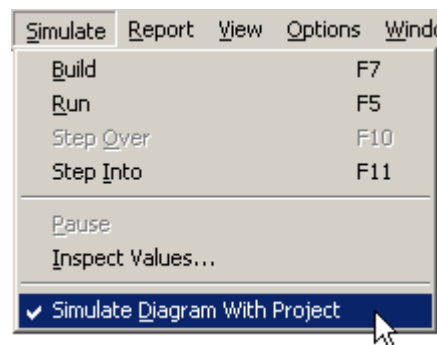
So far in the tutorial, we have created a project and simulated some code using a manually written testbench, which is the traditional design flow using a simulator. In the next few sections, you will be drawing a testbench using SynaptiCAD's graphical testbench generator. Before we start, we must prepare the project by removing the manually written testbench file and clearing out the Stimulus and Results diagram. Then extract the MUT ports into the Stimulus and Results diagram.

**Remove the Test Bench Source Code file and empty the Stimulus and Results Diagram:**

- Select the file **add4test.v** file in the project window and press the **<delete>** key.
- Delete all of the signals in the **Stimulus and Results** diagram by selecting the signal names and pressing the **<delete>** key.
- Verify that only **add4.v** is listed on the project tree, and that the diagram is empty.

**Verify that BugHunter is in the proper mode to generate a test bench:**

- Verify that the **Simulate > Simulate Diagram With Project** menu option is checked. This option lets the simulator compile both the drawn waveforms and the Verilog source code files together. If this is unchecked, then no testbench will be created.

**Extract Ports from the Model Under Test (MUT):**

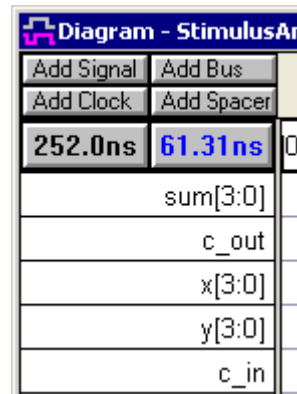
In the previous section, the **Stimulus and Results** diagram displayed only signals output by the simulator (the results). This diagram can also hold a testbench that will exercise the model under

test (the stimulus). First we will extract the ports from the model under test and later we will draw the test bench.

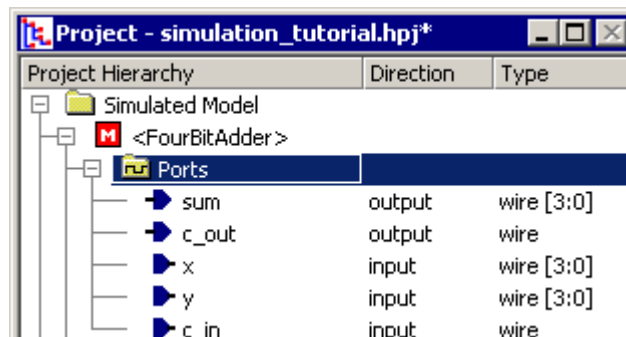
- Press the **Extract the MUT ports into Diagram** on the simulator button bar.



- Notice that the **Stimulus and Results** diagram is populated with the ports of the **FourBitAdder** module (see project tree below). These will be the signals that you will draw on in the next section.



- The Extract the MUT function makes a guess as to which model is the model under test and displays that model with single brackets, <>, underneath the Simulated Model folder. It guessed correctly for this tutorial.



- If you wanted to pick a different model under test, right click on a different model under the User Source Files list and pick **Set as Model Under Test**.

## (Sim) 1.6 Draw Test Bench in Debug Run Mode

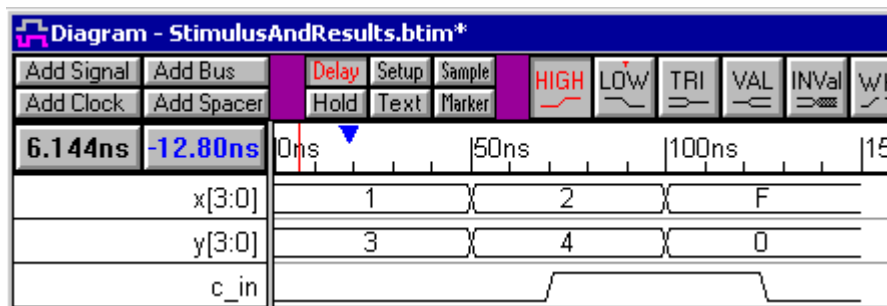
Draw the test bench on the input ports of the of the MUT. You can draw anything and see the results, however the result images shown in this tutorial will be using the timing diagram shown below.

### Draw the Test Bench

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



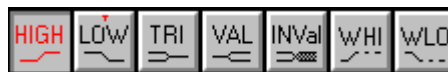
- Draw the following waveforms on the **input ports** of the MUT. Notice that the waveforms are **black** to indicate that will drive the MUT. If you have never used SynaptiCAD's drawing environment, then read the rest of the instructions in this section and practice drawing random signals before drawing the following diagram:



- Notice also that if you draw on the **output ports** (*sum* or *c\_out*) the waveforms would be **blue** to indicated that they are *expected outputs* from the MUT. This expected output is used by the *Reactive Test Bench Option* to create self-checking test benches, which is covered in the Reactive Test Bench Option Tutorial. For this tutorial, you can delete the **sum** and **c\_out** signals because we will not use them in this tutorial.

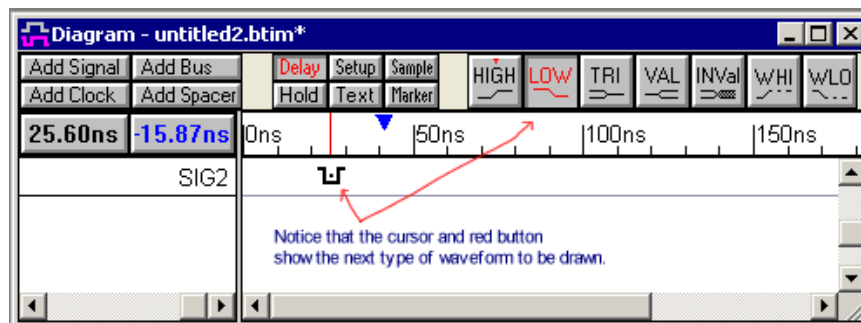
### Basic Drawing Instructions (Skip to Section 1.7 if you can draw the timing diagram)

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 and 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:

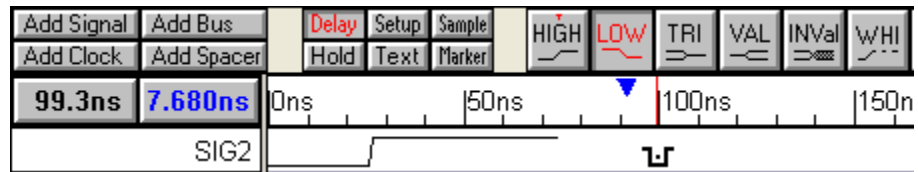
- Place the mouse cursor inside the *Diagram* window at the same vertical row as the signal name. The red state button on the button bar determines the type of waveform drawn. The cursor shape also mirrors the red state button



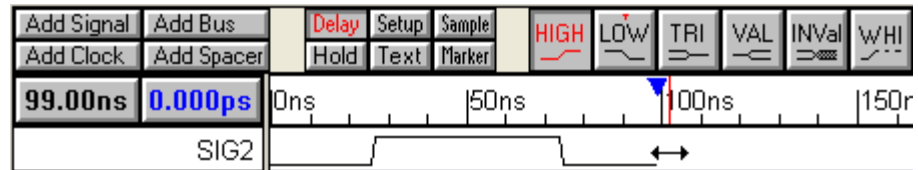
- Click the left mouse button. This draws a waveform from the end of the signal to the mouse cursor.



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



- Keep drawing from left to right across the diagram.

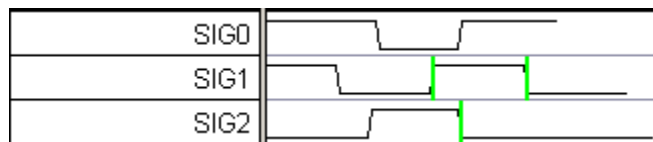
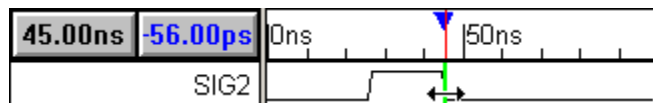
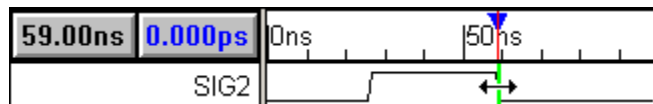


- Pressing the **middle mouse button** either toggles the state buttons or cycles through them depending on the setting in the *Design Preferences* dialog. Choose **Options > Design Preferences** menu to open the dialog.

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 (purple) signals.

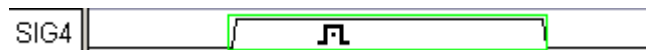
### 1) Drag-and-Drop a Signal Transitions:

- 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 right.
- To move transitions on different signals, first select the transitions by holding the <CTRL> while clicking on them. Then drag the transition to desired location.



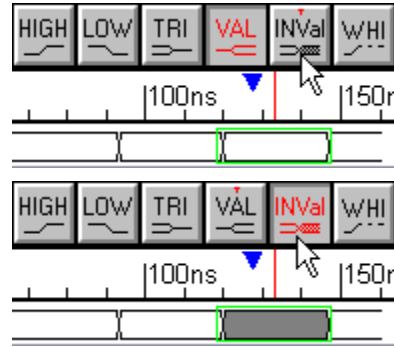
### 2) Click-and-Drag to insert a segment into a waveform:

- 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 same level as an adjacent section, the transition will turn red to preserve the edge data. This transition can be deleted if necessary.



### 4) Adding virtual state Information to a segment

- For Signals, double-click on the middle of a segment to open the *Edit Bus State* dialog, and then type in a new value into the **Virtual** edit box.
- For Clocks, press the **Hex** button and then double-click on the middle of the segment to open the *Edit Bus State* dialog. If the Hex button is not pressed, the double-click will open a different dialog to allow editing of the clock.



## (Sim) 1.7 Simulate in Auto Run Mode

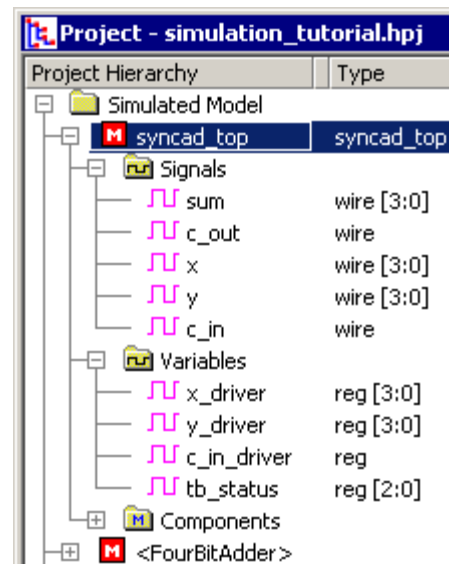
In this section, you will build and simulate the MUT with the graphical testbench. Then you will experiment with the Debug Run and Auto Run simulation modes.


### Build and Simulate the Project:

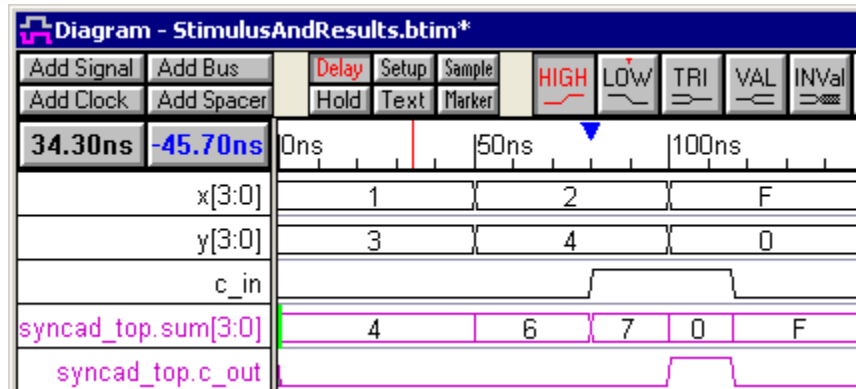
- Press the yellow **Build** button on the simulation bar.






- Notice that after the build is complete, a new model called **syncad\_top** was added to the *Simulated Models* tree. This is the top-level instance for the project and it hooks up the graphical test bench signals to the model under test (FourBitAdder). Double clicking on syncad\_top will let you view the generated test bench code.
- Also notice that all of the signals and variables of syncad\_top have been added to the **Stimulus and Results** diagram (like in section 1.3).
- For this tutorial we are concerned with the outputs of the MUT, so you can ignore all of the signals except the **sum** and **c\_out** signals.
- The **driver** and **status** signals are used by our TestBench Pro product to control the execution of multi-diagram testbenches. These are not used in the tutorial.



- Press the large green **Run** button to simulate the entire test bench. 
- Verify that **Sum** and **c\_out** equals **x + y + c\_in**. The schematic is shown in section 1.2.



### Debug Run versus Auto Run Simulation Modes:

- Currently the simulator should be in **Debug Run** mode, so that simulations are only compiled when the Build button is pushed. 
- Drag and drop an edge on **c\_in** and notice that the simulated waveforms do not change. To update the waveforms, press the **Build** button, followed by the **Run** button. 
- Press the **Debug Run** mode button to toggle the mode to **Auto Run**. 
- Drag and drop the same edge on **c\_in** and now notice that the simulated waveforms change each time an input is changed. In Auto Run mode, simulations are performed each time a waveform is moved (the Build and Run actions are automatically performed). However, if the simulator is paused in the middle of a simulation when the waveform is changed, then you must manually restart the simulation to apply the change. This keeps minor mouse clicks from prematurely exiting a debug session.
- Experiment with dragging edges and changing the values of the virtual states. If this was a low-level module that you just designed, then you could quickly check the functionality of the module without having to design a formal test bench. If you are running VeriLogger Extreme, you need to wait between changes for the simulator to compile and finish the previous simulation.

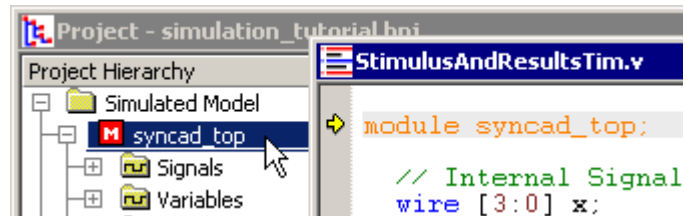
## (Sim) 1.8 Breakpoints, Stepping and Inspecting

In this section we will place a breakpoint in the generated code and take a brief look at the debugging environment.

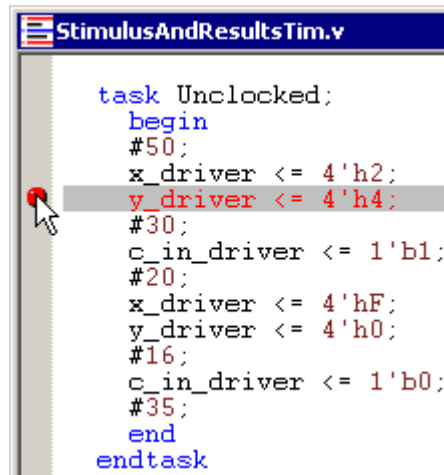


**Insert a code breakpoint and simulate**

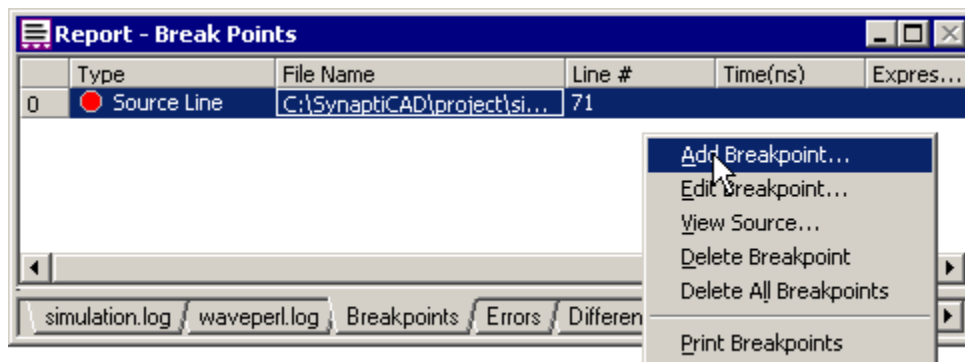
- Double click on the **syncad\_top** module in the project tree to open an editor that displays the generated code.



- Scroll down to the **task Unlocked** module, to see the stimulus code that was generated from the drawn waveforms.
- Place a breakpoint on the first change of the **y\_driver** signal by left clicking on the grey bar on the side of the editor window. This will place a red dot in the margin.



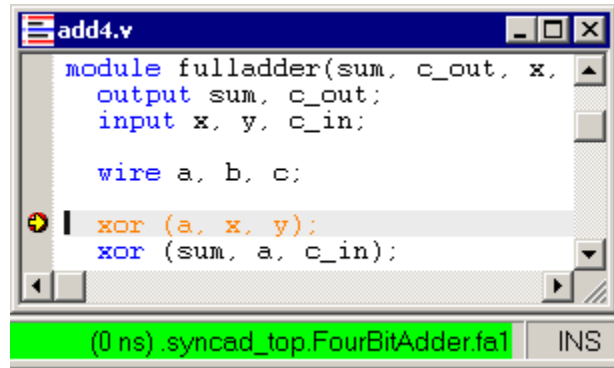
- Also note that the **Report** window **breakpoints** tab lists the source line breakpoint. BugHunter also supports time and condition breakpoints that are covered in Chapter 3 Debugging of the BugHunter and VeriLogger manual. These other kinds of breakpoints can be added by right clicking on the breakpoint tab window and choosing **Add Breakpoint** from the context menu, but the easiest way to add a condition breakpoint on a signal is to right click on it in the project tree and select the **Add/Toggle Condition Break Point...** menu option. This will cause the simulation to stop whenever the signal changes value.



- Press the large green **Run** button to simulate to the breakpoint. This particular code line will execute twice at time 50 because it is a non-blocking statement. The expression values of Non-blocking statements are evaluated when the statements are first encountered, but they only update their assigned signal at the end of the simulation cycle (as opposed to blocking statements which evaluate and immediately update the assigned signal).



- Add a breakpoint to the first **exclusive or** in the **add4.v** file, then restart the simulation.
- Press the run button a few times and watch the green status bar at the bottom. The fulladder model is instantiated 4 times inside the FourBitAdder module, so you are going to hit this breakpoint a lot. The status bar shows which instance is executing. Here, the **fa1** instance of **fulladder**, which is instantiated in module **FourBitAdder** is about to execute.



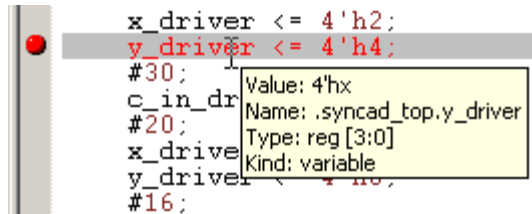
### Single step through the code

- Next press one of the single step buttons a couple of times to watch the execution of the code. Notice that **step with trace** (the middle button) leaves a trail of statements in the simulation log tab of the *Report* window. Make sure to check the status bar and compare the execution to the schematic in section 1.2.



### Use the Inspect Values

- Put the cursor over a variable that has been initialized to see its current value.



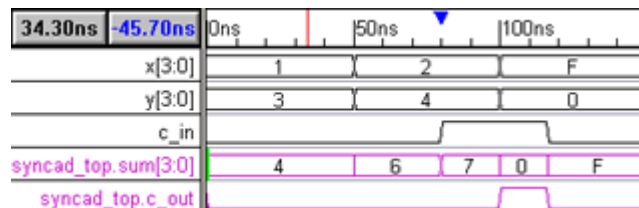
- Variables that are watched in the **Stimulus and Results** diagram can be inspected for values at previous times. See the instructions in the **BugHunter and VeriLogger** manual for using the **Simulate > Inspect Variables** menu function (to fill the window, drag signal names from the **Stimulus and Results** diagram or type them in manually).

## (Sim) 1.9 Archiving Stimulus and Results

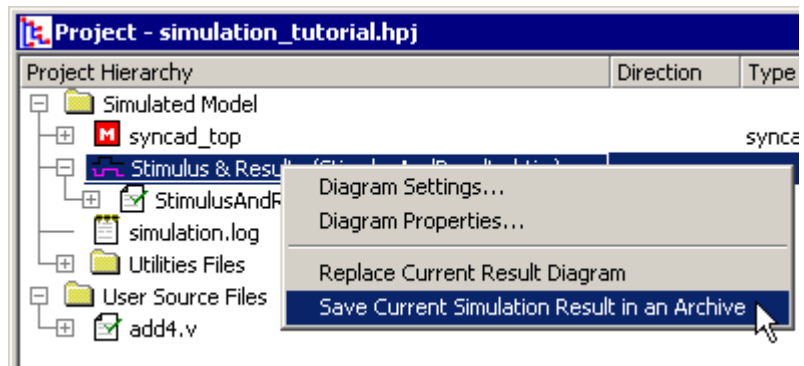
So far we have only used a single **Stimulus and Results** diagram, however multiple diagrams can be switched in and out to test different aspects of the design, or archived off to be used as comparison diagrams for later simulations.

### Archive off the simulation results:

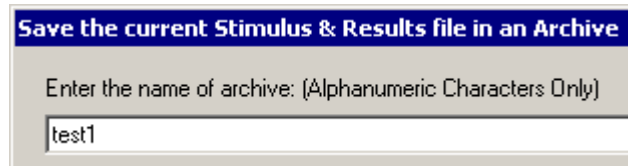
- Run the simulator to the end of the simulation so that the Stimulus and Results diagram and simulation log file are full of data.



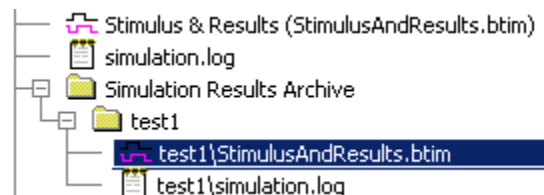
- Right click on the Stimulus and Results node and choose **Save Current Simulation Result in an Archive** from the context menu, to open a dialog.



- Name the archive **test1** and close the dialog. This will create a subdirectory called test1 under the project directory.

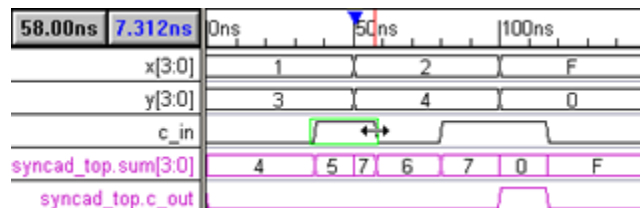


- Notice that a new node called Simulation Results Archive has been added to the tree. Both the Stimulus and Results diagram and the simulation log file have been copied to the new archive directory.

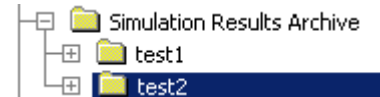


#### Create a new test and archive that off:

- Modify the stimulus and then simulate to see the changed waveforms.

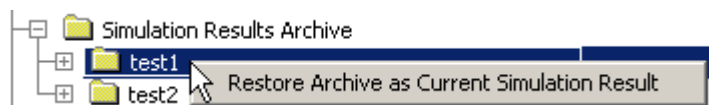


- Archive these results to an archive named **test2** using the method described above.

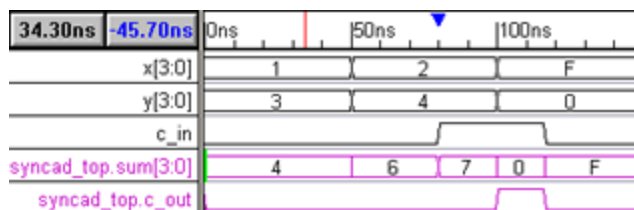


#### Switch back to the original test:

- Right click and choose **Restore Archive as Current Simulation Result** from the context menu to restore the results



- Notice that the **Stimulus and Results** diagram has been replaced with the original test1 data

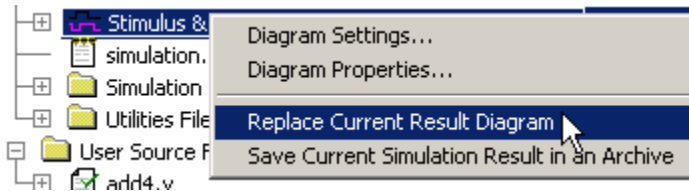


- The archive files remain untouched by the **restore** command. The data is copied from the archive into the working directory and you need to re-archive if you want to save the new data.
- If you purchase the **Compare Option**, then you can use the archive diagrams to compare against new simulations of your design.

### *Multiple Stimulus and Results diagrams:*

In the previous example, both archives used the default **Stimulus and Results** diagram name so it can get a little confusing. However, the diagram can be named any name.

- Save the **Stimulus and Results** diagram to a new name using the **File > Save Current Diagram** menu option.
- Right click on the **Stimulus and Results** node and choose **Replace Current Result Diagram** from the context menu, then choose the file that you saved.



## **(Sim) 1.10 Saving the Project files**

Whenever you build a project, the project file and any modified source files are automatically saved. You can also manually save files at any time. This section describes how the different file types can be saved manually.

### *Save the HPJ project file:*

- Choose the **Project > Save Project** menu option to save the \*.hpj file. The project saves the simulation options and the names of the files contained on the project tree. It does not save the source code or the watched signals.

### *Save the Source code:*

Each time you simulate, every open editor is queried to determine if the source code needs to be saved before the simulation starts. You can also force a save by doing the following:

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

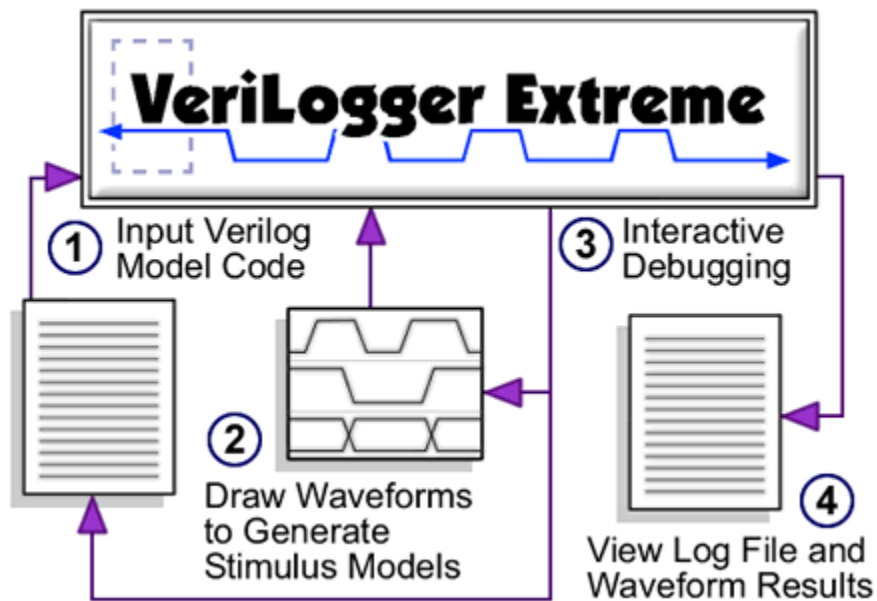
### *Saving the Stimulus and Results diagram:*

The watch signals and simulation results are saved in the current stimulus and results file.

- Click on the **Stimulus and Results** diagram window, then select the **File > Save Timing Diagram** menu option to save the diagram. Note: the evaluation version does not allow diagrams to be saved (you will need to buy a full license or get a temporary evaluation license to perform this function).

## **(Sim) 1.11 Summary of VeriLogger Basic Verilog Simulation**

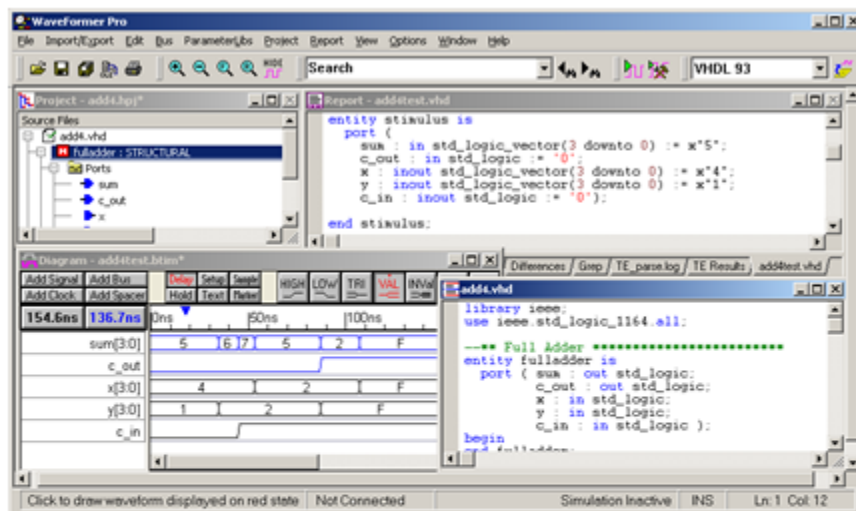
Congratulations, you have completed the VeriLogger Basic Verilog Simulation tutorial. We have demonstrated how to create a project, copy files into the project, simulate and view the results. Chapter 1 of the BugHunter Pro and VeriLogger Pro manual has a step-by-step design flow of how to set up a simulator and create a project. You may wish to read that before attempting to create a complicated project.



We have also introduced the graphical testbench generation that comes standard with BugHunter Pro. This feature generates a test bench code from a single timing diagram. There are two other levels test bench generation that can be added to BugHunter. The first is the **Reactive Test Bench Option** that generates self-checking code from the expected waveform information. The highest level is **TestBench Pro** which creates bus functional models from multiple timing diagrams and is able to apply randomized data to each transaction.

## Simulation 2: Using WaveFormer with ModelSim VHDL

WaveFormer Pro can be used to create a VHDL stimulus file for a VHDL model that needs to be tested. Then the test bench and the model under test can be simulated using an external VHDL simulator. In this tutorial we show the commands to use ModelSim, but if you are using a different simulator this should give you the basic idea for controlling the simulation. Then we will use WaveFormer Pro to compare the simulation results against expected results drawn by the user.



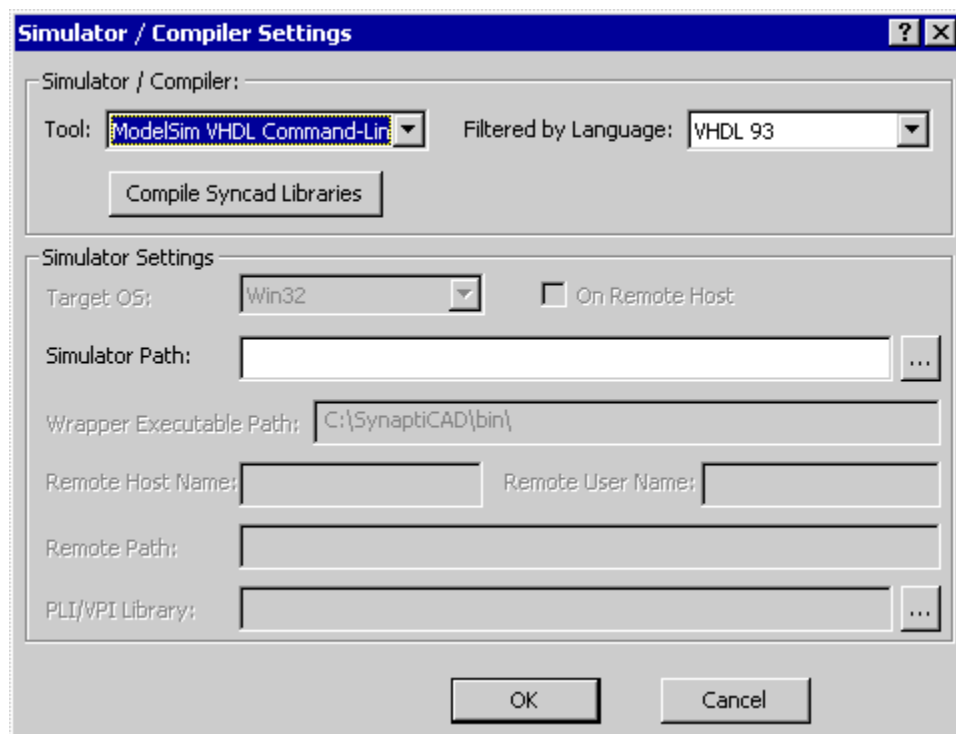
**Draw a VHDL test bench  
and simulate with an  
external VHDL simulator**

To perform this tutorial, you will need a license for WaveFormer Pro with the Comparison feature and a license for some version of ModelSim VHDL (XE, PE or SE).

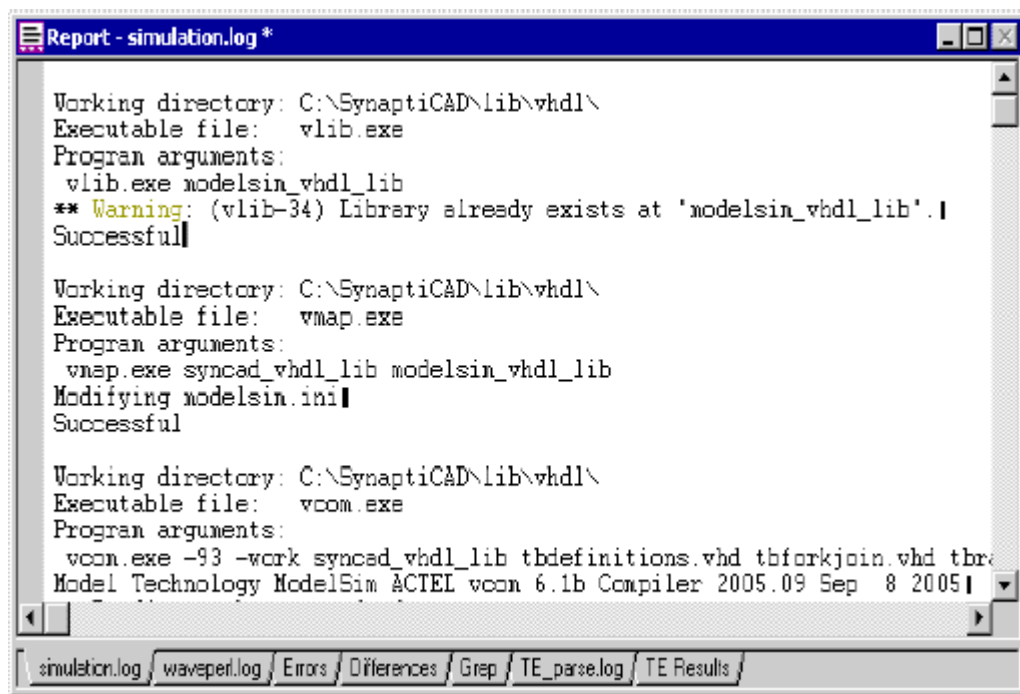
### (Sim) 2.1 Compile SynaptiCAD Library Models

**First time only step:** When you first begin using WaveFormer Pro, you will need to compile SynaptiCAD's testbench library models with ModelSim.

- Start WaveFormer and select the **Options > Simulator/Compiler Settings** menu option to open the *Simulator and Compiler Settings* dialog.



- From the **Tool** drop-down box, select the your particular ModelSim VHDL compiler. If you are not sure which version of ModelSim that you have, select **ModelSim VHDL Command-Line XE/PE**.
- Press the **Compile SynCAD Libraries** button to compile the SynaptiCAD Libraries. The first time that you set up a particular simulator or compiler you should press this button.
- You can view the results of this simulation by looking in the **simulation.log** tab of the *Report* window. If you cannot see the *Report* window, then choose the **Window > Report** menu option to bring the window to the front.



```

Report - simulation.log *

Working directory: C:\SynaptiCAD\lib\vhdl\
Executable file:  vlib.exe
Program arguments:
  vlib.exe modelsin_vhdl_lib
** Warning: (vlib-34) Library already exists at 'modelsin_vhdl_lib'.!
Successful

Working directory: C:\SynaptiCAD\lib\vhdl\
Executable file:  vmap.exe
Program arguments:
  vmap.exe synacd_vhdl_lib modelsin_vhdl_lib
Modifying modelsin.ini
Successful

Working directory: C:\SynaptiCAD\lib\vhdl\
Executable file:  vcom.exe
Program arguments:
  vcom.exe -93 -work synacd_vhdl_lib tbdefinitions.vhd tbforkjoin.vhd tbr
Model Technology ModelSim ACTEL vcom 6.1b Compiler 2005.09 Sep  8 2005
simulation.log / waveper.log / Errors / Differences / Grep / TE_parse.log / TE Results

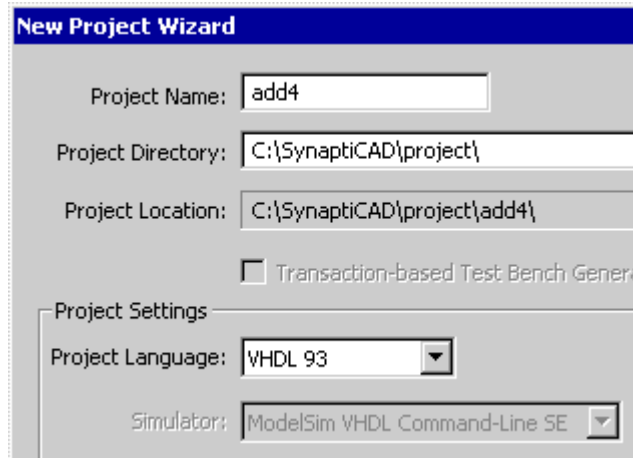
```

## (Sim) 2.2 Create a project and extract the ports

Create a project in WaveFormer Pro to hold the model under test. This will give WaveFormer access to the port information of the model

*In WaveFormer Pro create a new Project:*

- Select the **Project > New Project** menu to open the *New Project Wizard* dialog.
- Name the project **add4**. This will be both the name of the project and the directory name where the project is stored.
- Set the **Project Language** drop-down to **VHDL 93**. This tells WaveFormer's parser what language the MUT (*Model Under Test*) is written in.
- Close the dialog to create the project.

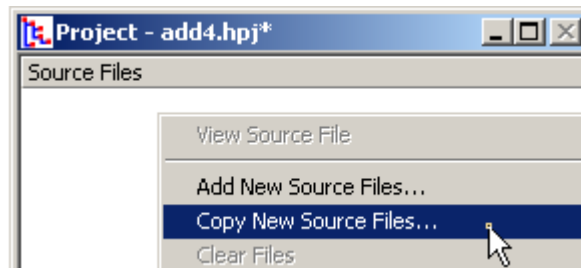


*Copy the Model Under Test File into the project:*

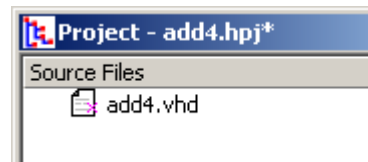
Next we will copy the file containing the VHDL model under test to the project directory and add the source file to the project file.



- Right click on the surface of the Project window and choose **Copy New Source Files** from the context menu. This will open a file dialog.



- In the above picture, the **copy** command copies the files to the project directory. The **add** command will leave the file in place and just point to the file. For this tutorial we will copy the files.
- In the file dialog, browse to the directory **C:\SynaptiCAD\Examples\** and select the **add4.vhd** file. Then press ok to close the dialog. The file should appear in the Project window.

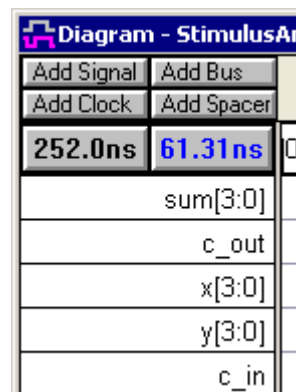


#### *Extract the port information from the Model Under Test:*

- Press the **Extract MUT Ports** button on the toolbar.



- Notice that the ports of the fulladder model have been inserted into the **Stimulus and Results** diagram. If no signals appear, make sure that language drop-down on the toolbar is set to VHDL.
- Select the **File>Save Timing Diagram As** menu to open a file dialog. Save the diagram and name it **add4test.btim**.

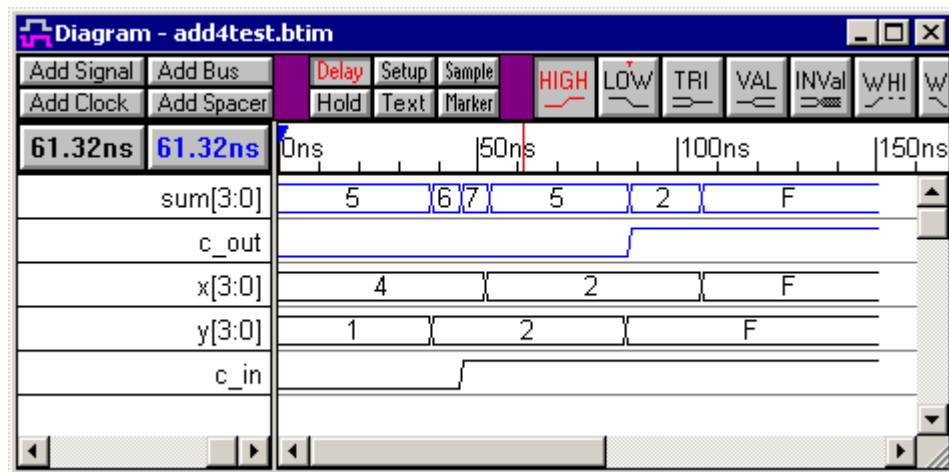


## (Sim) 2.3 Draw the test bench waveforms

Draw the stimulus on the input ports of the of the MUT and the expected output on the output signals. You can draw anything and see the results, however the rest of the images shown in this tutorial will be using the timing diagram shown below. If you do not want to draw the testbench then you can copy the **add4test.btim** file from **c:\SynaptiCAD\Examples\TutorialFiles\Waveformer2MsimVHDL** directory into the project directory.

#### *Draw the Test Bench*

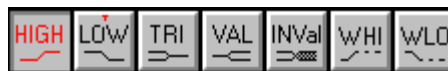
- Draw the following timing diagram. If you have never used SynaptiCAD's drawing environment, then read the rest of the instructions in this section and practice drawing random signals before drawing the following diagram:



- Notice that the **black** waveforms are the inputs and will generate code that will drive the MUT. The **blue** waveforms are outputs of the MUT and represent expected outputs of the model.
- Save the changes to **add4test.btim** using the **File>Save Timing Diagram** menu, as we will use this btim file later to compare our expected results against the VCD results file generated during simulation.

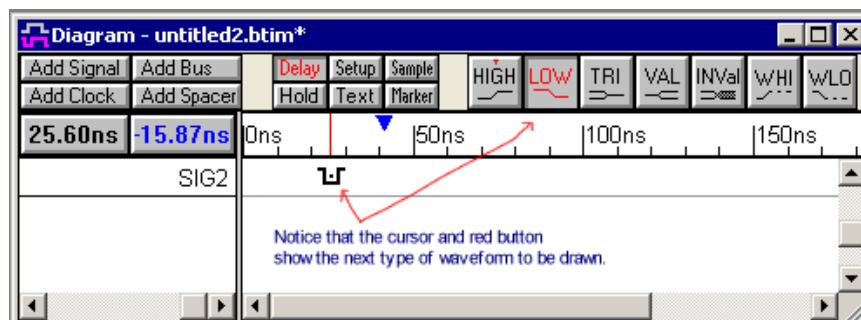
#### Basic Drawing Instructions (Skip to Section 2.4 if you can draw the timing diagram)

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 and 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.

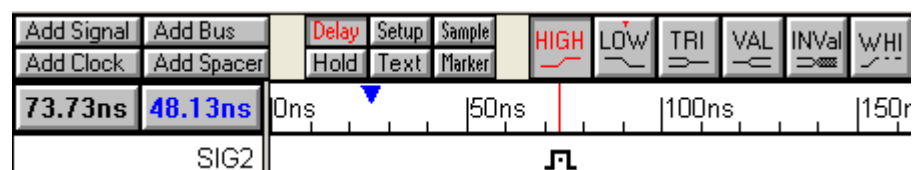


#### To draw the waveform of a signal:

- 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

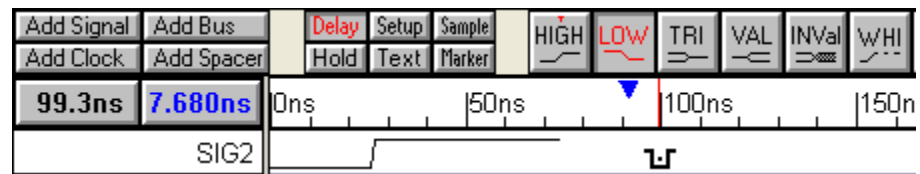


- Click the left mouse button. This draws a waveform

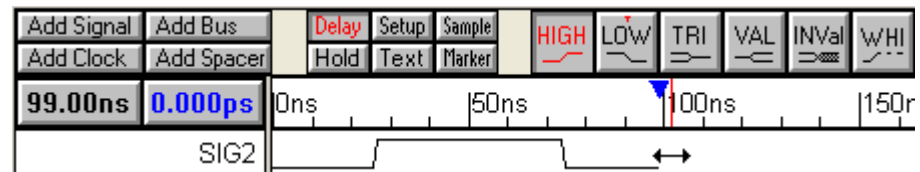


from the end of the signal to the mouse cursor.

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



- Keep drawing from left to right across the diagram.

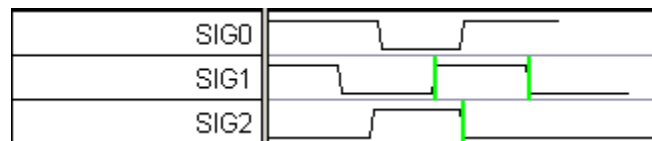
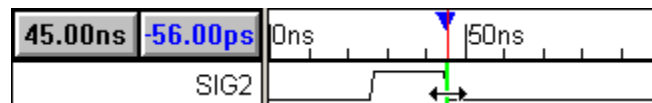
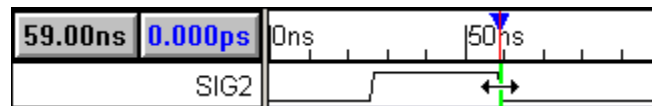


- Pressing the **middle mouse button** either toggles the state buttons or cycles through them depending on the setting in the *Design Preferences* dialog. Choose **Options > Design Preferences** menu to open the dialog.

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 (purple) signals.

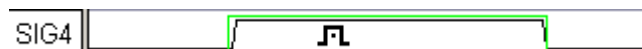
### 1) Drag-and-Drop a Signal Transitions:

- 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 right.
- To move transitions on different signals, first select the transitions by holding the <CTRL> while clicking on them. Then drag the transition to desired location.



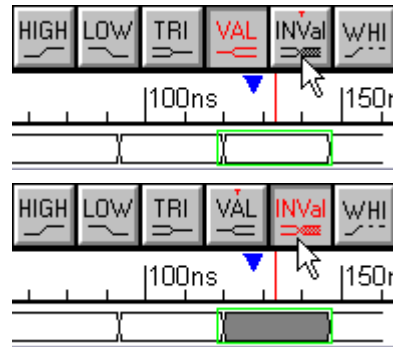
### 2) Click-and-Drag to insert a segment into a waveform:

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



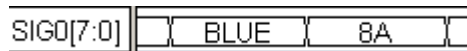
### 3) Change a segment's graphical state by selecting it and then pressing a state button:

- 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 same level as an adjacent section, the transition will turn red to preserve the edge data. This transition can be deleted if necessary.



### 4) Adding virtual state Information to a segment

- For Signals, double-click on the middle of a segment to open the *Edit Bus State* dialog, and then type in a new value into the **Virtual** edit box.
- For Clocks, press the **Hex** button and then double-click on the middle of the segment to open the *Edit Bust State* dialog. If the Hex button is not pressed, the double-click will open a different dialog to allow editing of the clock.

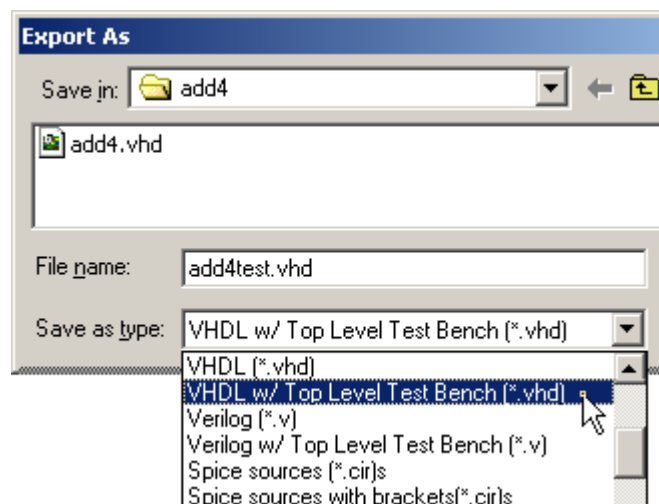


## (Sim) 2.4 Export Waveforms to VHDL

In the first steps of this tutorial, we created a project file and pressed the **Extract MUT Ports** button. This also determined the model under test module. The project must remain open during the export of the btm file in order for the model under test to be instantiated in the stimulus file. If the source files in the project contained multiple modules that could be the top level MUT instance, you would need to select the top level instance in the project window by right clicking on the desired top level instance.

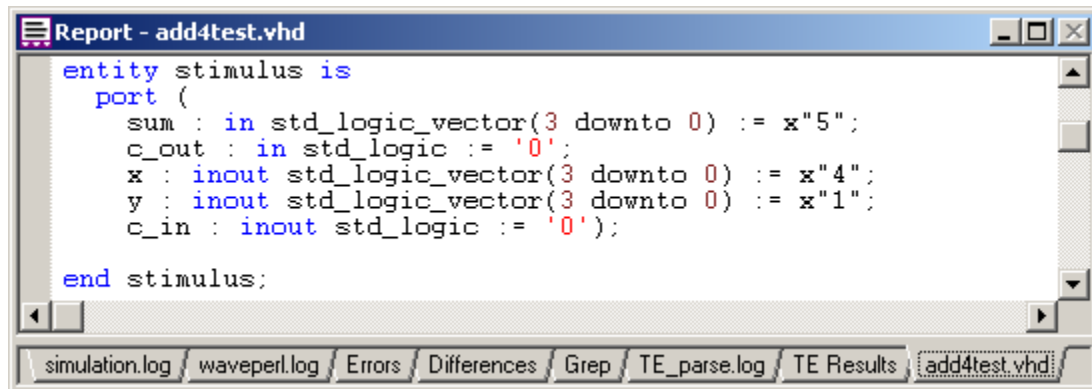
### Generate the VHDL test bench by exporting the timing diagram file.

- Choose the **Import/Export > Export Timing Diagram As** menu option to open the *Export As* dialog.
- Set the **Save As Type** drop-down to **VHDL w/Top Level Test Bench (\*.vhd)**. This creates a VHDL file called **add4test.vhd** containing the stimulus and instantiates a copy of the model under test.



- Once the file is generated it is also loaded into the *Report* window so that you can see the generated code. If you cannot see the *Report* window, then choose the **Window > Report**

menu option.



## (Sim) 2.5 Simulate VHDL test bench using ModelSim

First map the standard syncad library, syncad\_vhdl\_lib, so that ModelSim can locate it in this project (this step requires that you have previously compiled the syncad library as described at the beginning of this tutorial). This step only needs to be done once for a project. Next, run the simulation and generate the vcd results file, you can either use the ModelSim *do script* included with this tutorial or manually type the commands.

### Map SynaptiCAD library to ModelSim:

- Launch a DOS command prompt and type the following commands:  

```
cd \synapticad\project\add4
vmap syncad_vhdl_lib C:\Synapticad\lib\vhdl\modelsim_vhdl_lib
```

### ***EITHER run the do script to simulate, by typing from the command prompt:***

```
copy c:\Synapticad\Examples\TutorialFiles\Waveformer2MsimVHDL\add4test.do
.
modelsim -do add4test.do
```

### ***OR type the following individual commands below***

- To generate library files:  

```
vlib work
```
- To compile the source files:  

```
vcom add4.vhd add4test.vhd
```
- Select the top level module to simulate (this will launch the ModelSim GUI):  

```
vsim testbench
```
- In the console window of the ModelSim GUI, set the name of the vcd dump file:  

```
VSIM> vcd file add4.vcd
```
- Specify the signals to dump to the vcd file (top level signals in the design):  

```
VSIM> vcd add /testbench/*
```
- Simulate the design+stimulus  

```
VSIM> run -all
```
- Exit the simulator (the vcd file will be created by ModelSim at the end of this step):  

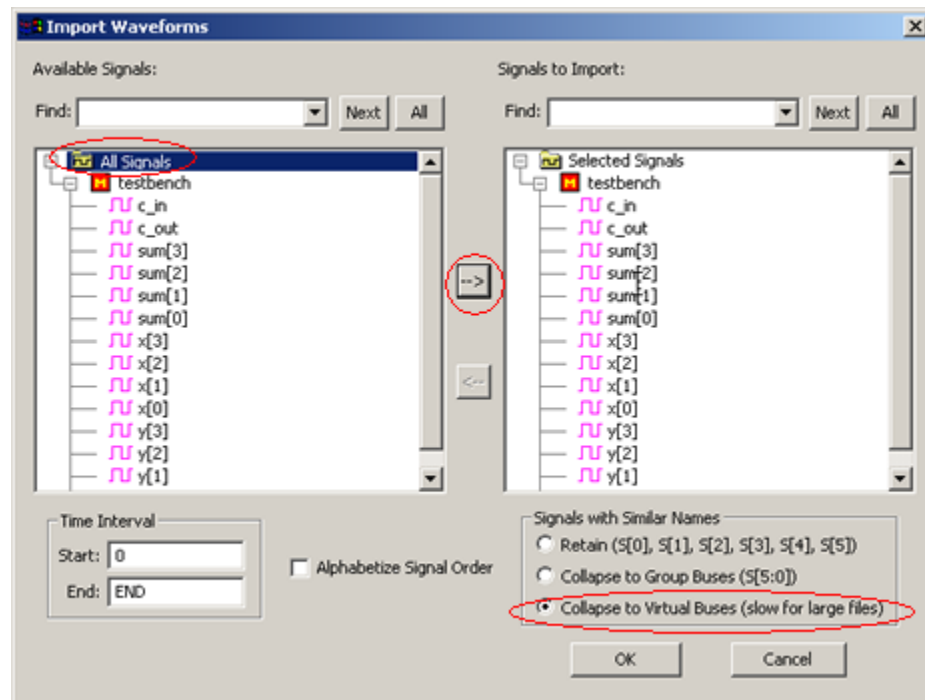
```
VSIM> quit -f
```

## (Sim) 2.6 Compare simulation results against expected results

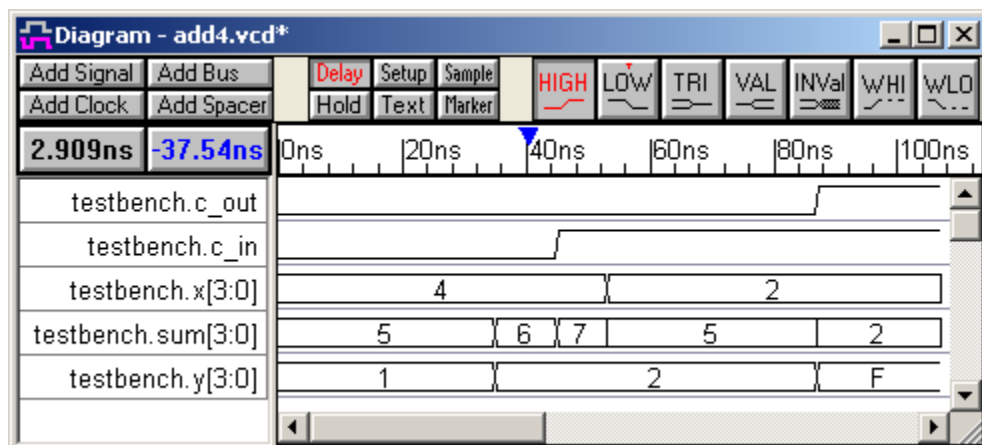
In this step we will compare the simulation results against the expected waveforms that we drew in an earlier section. The compare feature is an option that can be added on to WaveFormer Pro.

### A) Load the Simulation Results file into WaveFormer Pro

- Select the **Import/Export > Import Timing Diagram From** menu option and load the **add4.vcd** file created in the previous step to open the *Import Waveforms* dialog.



- Check the **Collapse to Virtual Buses** box, so that the signals will be imported as buses instead of as individual bits.
- In the left hand pane, select the **All signals** node then press the => button to move the signals to the right hand pane. Then press **OK** to import the diagram.

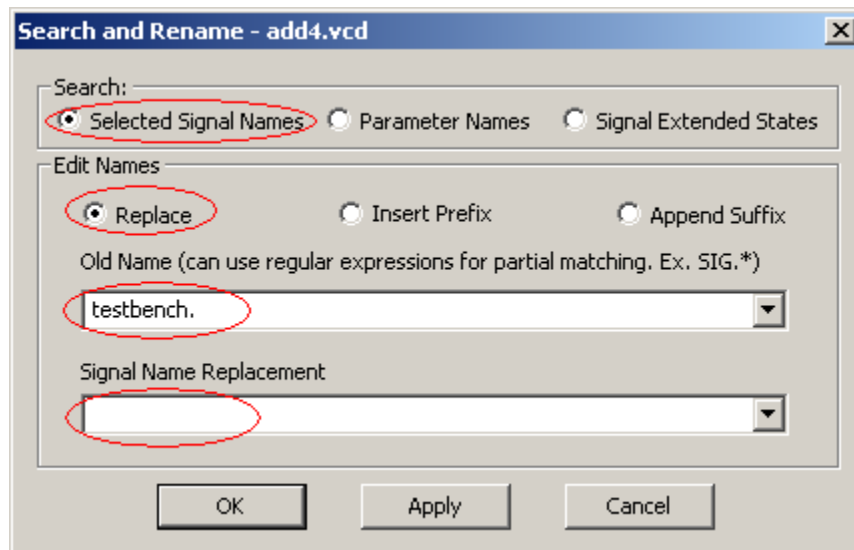


### B) Strip out the Simulation Model Names

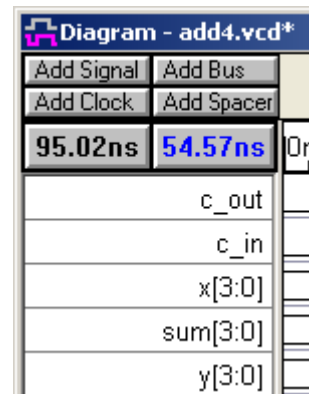
Before we can compare the VCD file to the expected results btim file, we must make sure that the

signal names are the same. During simulation, the model name "testbench." prefix was added to each of the VCD's signals. We can either strip out the name from the simulation file or add the prefix to the expected waveforms. Either way, we can use the *Search and Rename Signals* dialog to strip or add a prefix. Here we will strip the prefix from the simulation vcd file.

- Select the **Edit > Search and Rename Signals** menu to open the *Search and Rename Signals* dialog.



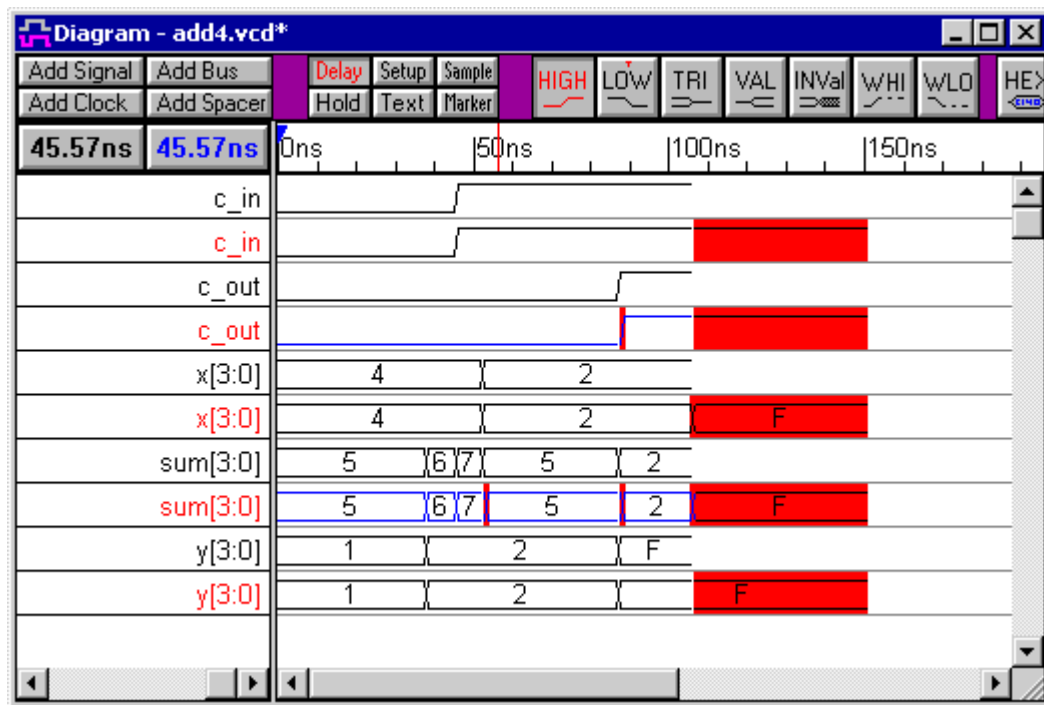
- Check **Selected Signal Names** to make the dialog operate on the signals. Since we did not have any signals selected in the diagram, this will operate on all the signals by default.
- Check **Replace** to make the dialog do a replace. If we had decided to add "testbench." to the names in the timing diagram, we would have to use *insert prefix* to add the prefix.
- Set the **Old Name** pattern to "testbench." making sure that you include the period after testbench.
- Leave **Signal Name Replacement** blank, because we just want to strip out the name.
- Press **OK** to strip the prefix from the signal names.



### C) Compare The Timing Diagrams

To do this step, you will need to have a license for the Compare Feature. This is normally turned on in the evaluation version, but must be purchased in the full version.

- Choose the **File > Compare Timing Diagram** menu to open a file dialog and select the file **add4test.btim** that you created earlier in the tutorial.
- The differences between the VCD file and the expected results file will show up in red. The differences are also displayed in list form in the **Differences** tab in the *Report window*.

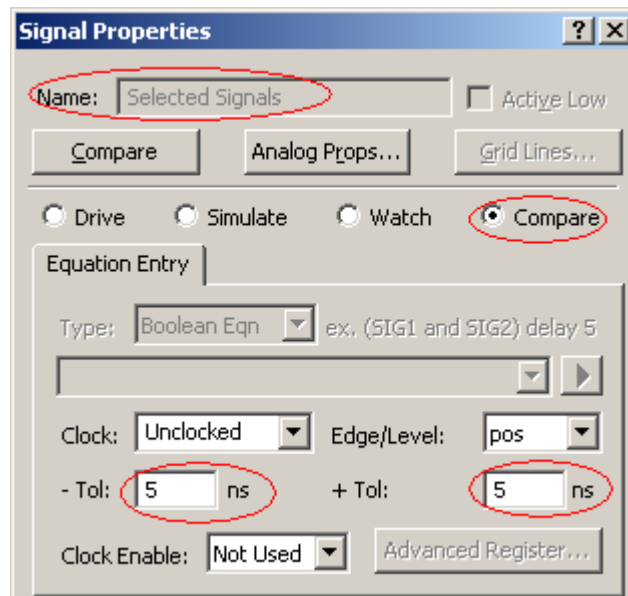


- When drawing the btm file waveforms, you may have some differences with the diagram shown on the tutorial. These differences will show up as highlighted lines in red in the comparison diagram. Minor differences can be removed by using the compare tolerances.

- Click the **SET ALL** button on the compare toolbar to open the *Signal Properties* dialog



- Notice that the **Name** box is greyed out. This means that the dialog is operating on all the compare signals.
- Type in a tolerance of 5 for both the **-Tol** and **+Tol** controls. This will allow the compare feature to ignore small changes in values.
- Click the **OK** button to apply the changes



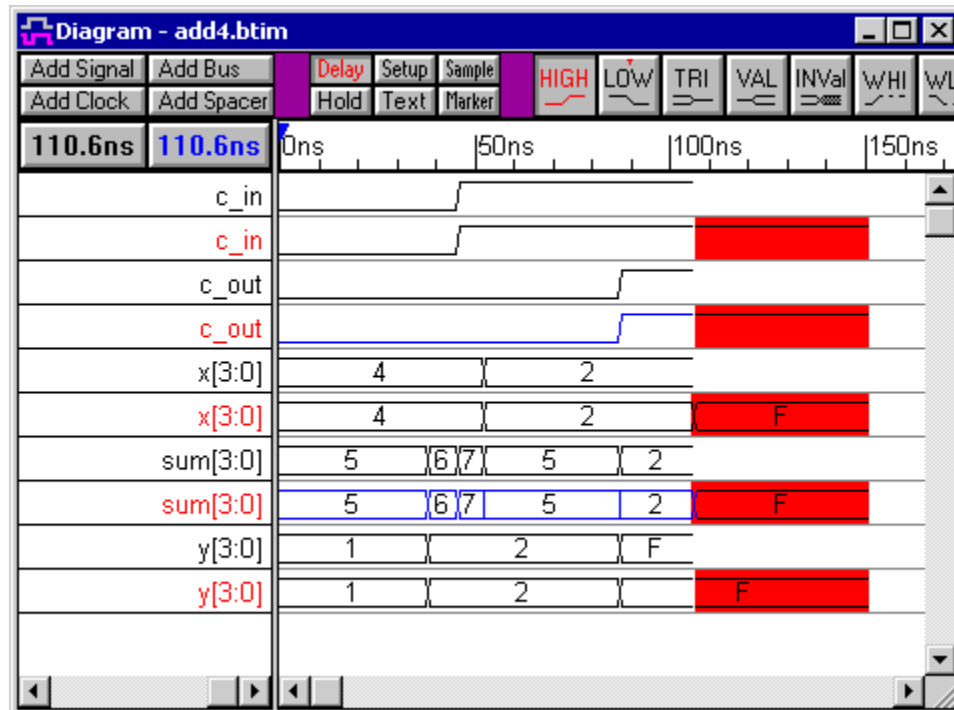
- Press the **Compare All Compare Signals** button to re-compare the waveform signals.



- Below is the resulting diagram. Notice that the thinner lines in red have now disappeared. The



remaining differences occur at the end of the waveforms because the simulator stops the simulation as soon as there are no more changes on the waveforms.



- Also look at the **Differences** tab in *Report* window, which shows a hyperlinked list of the differences. If you cannot see the *Report* window, then choose the **Window > Report** menu option to bring it to the front.

Report - Differences					
	Signal Name	Start Time	End Time	Reference S...	Compare State
0	sum	106	151		F
1	c_out	106	151		1
2	x	106	151		F
3	y	106	151		F
4	c_in	106	151		1

## (Sim) 2.7 Summary of Using WaveFormer with ModelSim VHDL

Congratulations! You have now learned how to use WaveFormer Pro with ModelTech's ModelSim VHDL. In this tutorial we covered how to create a new .hpj project in WaveFormer, adding new source files to the project, and drawing the necessary stimulus on the input and output signals. We also learned how to export waveforms to VHDL and how to generate the .vcd file required for testing through ModelSim. Last, but not least, we learned how to compare the simulation results against the expected results.

## Waveform Comparison Tutorial

Waveform Comparison is an optional module that can be added to all of SynaptiCAD's products except Timing Diagrammer Pro. This feature allows comparison between two timing diagrams or between individual signals in a timing diagram. The results of two simulation runs, or of logic analyzer data and a simulation run, can be compared very easily using this feature. This tutorial will describe both types of waveform comparison.

This tutorial will use the following files in the **Examples\TutorialFiles\WaveFormComparison** subdirectory of the installation directory (**C:\SynaptiCAD** by default on Windows):

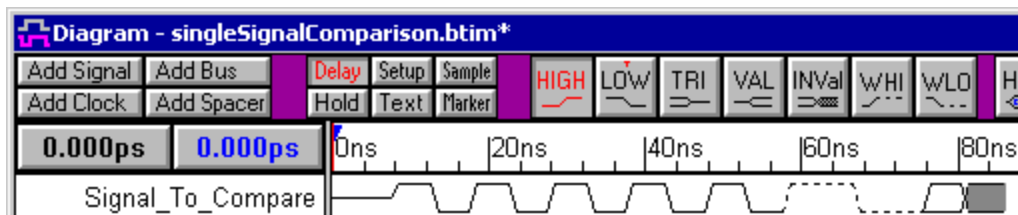
- singleSignalComparison.btim
- simulationResults.vcd
- analyzerData.txt
- simulationResults\_offset.btim

### 1. Creating Individual Compare Signals

One of SynaptiCAD's signal types is a **Compare** signal. This signal type can be used to compare an individual signal's waveform to that of another signal (from a simulation result, for example).

#### 1.1 To Create a Compare Signal

1. Open the **singleSignalComparison.btim** timing diagram. The diagram should look like this:

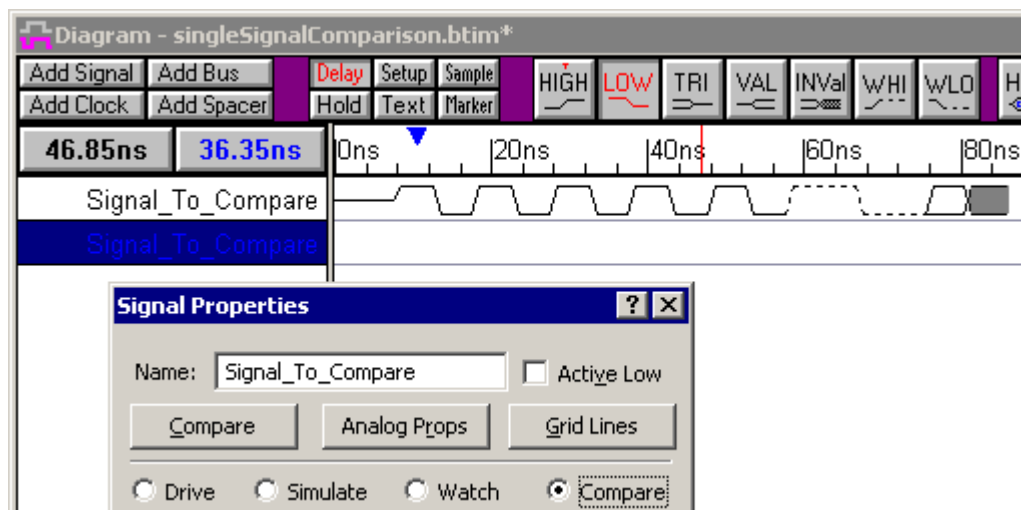


2. Click the **Add Signal** button to add a new signal to the diagram.
3. Double click the name of the new signal (SIG0) to open the *Signal Properties* dialog.
4. Set the *Name* of the new signal to match the original signal: **Signal\_To\_Compare**.

Note: the Compare signal and the original signal must have the same name, otherwise the product does not know which signals to compare.

5. Select the **Compare** radio button near the top of the *Signal Properties* dialog.
6. Notice that the signal name of the compare signal in the label window is blue.

Leave the *Signal Properties* dialog open for the next section.



## 1.2 Draw the Expected Waveform on the Compare Signal

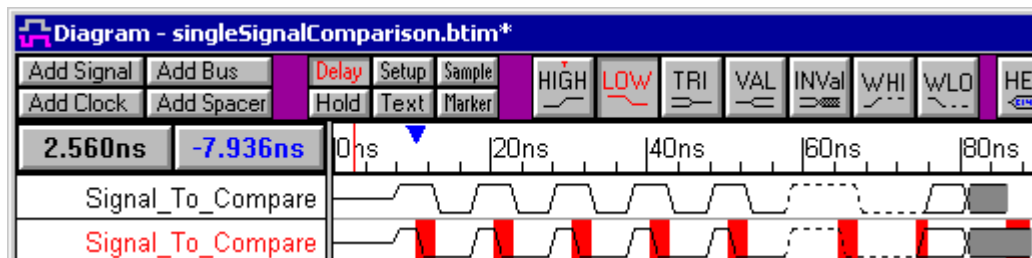
We will use the Waveform Equation generator to create a waveform that is similar to the one on the original signal.

1. In the *Signal Properties* dialog, modify the expression in the **Wfm Eqn** edit drop down so that it reads:

```
8ns=Z (3=1 7=0)*5 7=H 10=L 6=V 8=X
```

2. Click the **Wfm Eqn** button to draw the specified waveform.
3. Click the **Compare** button to compare the two waveforms.
4. Click **OK** to apply the changes and close the dialog.

Notice that there are now portions of the Compare signal waveform that are highlighted in red. These are the areas where differences were detected while doing a comparison with the tolerance set to 0.



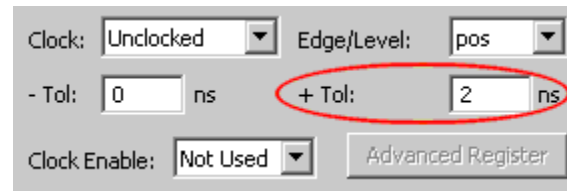
## 1.3 Comparison Using Tolerance

Each Compare signal can have a **+Tolerance** and a **-Tolerance** specified. This is one method for ignoring small differences during a comparison. Next, we will create a second compare signal that has a tolerance specified.

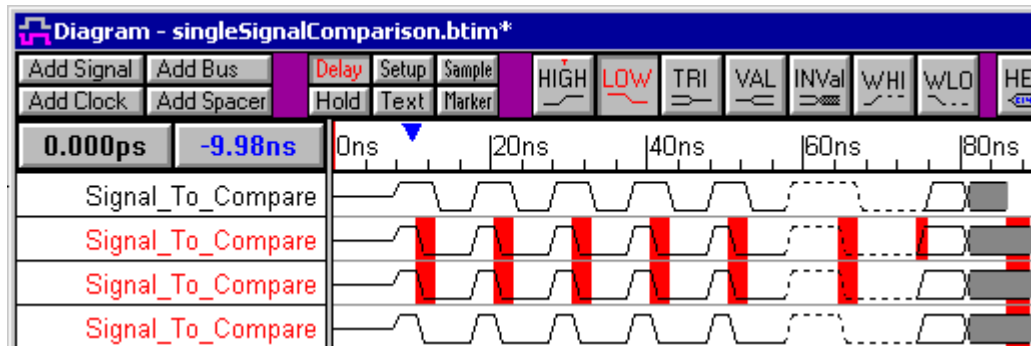
Add a copy of the original Compare signal and set the tolerance:

1. Select the Compare signal by clicking the red **Signal\_To\_Compare** signal name.
2. Press **<Ctrl>C** on the keyboard to copy the signal.
3. Select the **Edit > Paste Signal(s)** menu option or press **<Ctrl>P** to paste a copy of the original compare signal.

4. Double click the name of the new Compare signal to open the *Signal Properties* dialog.
5. Set the *+Tolerance* to 2.
6. Click the **Compare** button to view the changes.



Note that there is one less difference in the diagram now. If you repeat the steps above, but specify a Tolerance of 3, then all of the differences disappear except the one at the end of the Compare signal (where the state is driven longer than the original signal).



The diagram above shows an original Signal (with the black name label), and then compare signals in the following sequence: Tolerance = 0, Tolerance = 2 and Tolerance = 3.

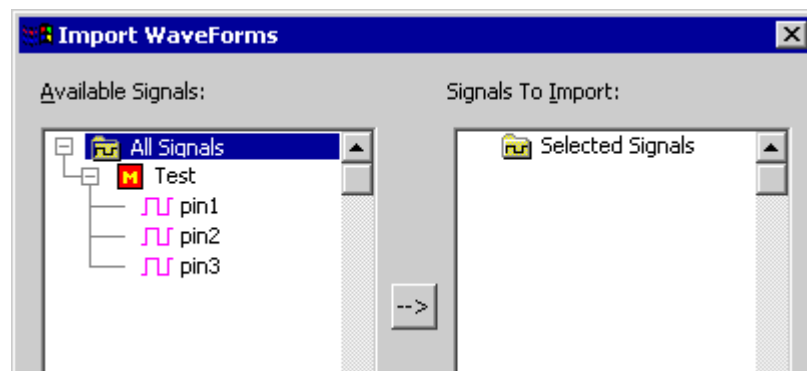
## 2. Comparing Timing Diagrams



The timing diagrams that are being compared can be hand drawn or could contain data from a logic analyzer or another supported timing data format. For this tutorial, a VCD file will be imported and saved in the **btim** format. It will be compared to data that is in a Spreadsheet format similar to that generated by a Tektronix logic analyzer.

### 2.1 Import VCD Data and Save as a .btim

When comparing two files, the file that is being compared/merged into the reference diagram must be a SynaptiCAD/btim file. Since the timing diagram editor reads many different file formats, it is a simple operation to translate a waveform file to .btim. In this step we will convert a VCD (Verilog simulation file) into a .btim file. Then, in the following steps, we will use it to compare to a native logic analyzer file.


1. Select the **Import/Export > Import Timing Diagram From...** menu option.
2. Select the **Verilog Value Change Dump (\*.dump, \*.vcd)** option from the *File of Type:* drop list.
3. Select the file **simulationResults.vcd** using the Browse functions of the dialog.
4. Click **Open** to launch the *Import WaveForms* dialog.
5. Click the **All Signals** folder in the *Available Signals* tree control on the left side of the dialog.
6. Click the right arrow button to place these signals in the *Signals to Import* tree.



7. Click **OK** to import the data to a timing diagram.
8. Use the **Zoom In** button  to get a better view of the changing signal edges.
9. Select the **File > Save As...** menu option to option the *Save File* dialog.
10. Select **Timing Diagram - Binary (\*.btim)** from the *Save as Type* drop list.
11. Click the **Save** button  to save the timing diagram in **btim** format (you may be prompted to overwrite a file if a file by the same name already exists).

## 2.2 Open the Logic Analyzer Data File

Next open the logic analyzer file.

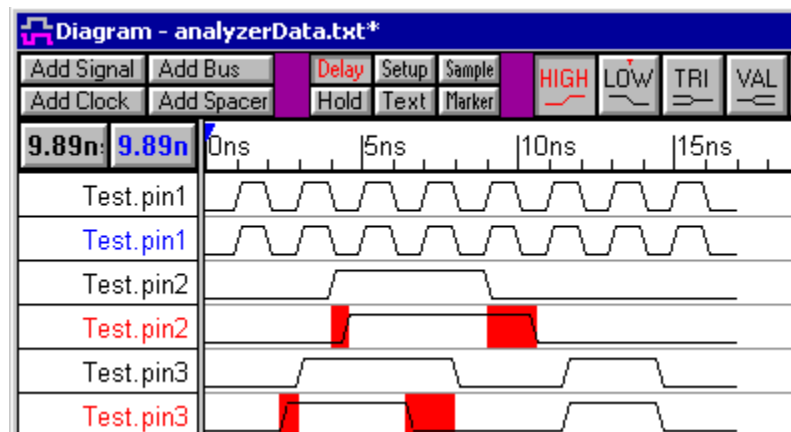
1. Use the **File Open**  button to open the *Open File* dialog.
2. Select **Test Vector Spreadsheet (\*.txt)** from the *File of Type* drop list.
3. Select the **analyzerData.txt** test vector spreadsheet file.
4. Click the **Open** button to open the file in a timing diagram.

## 2.3 Compare the Two Timing Diagrams

When two diagrams are compared, the signals from one diagram are brought into the other diagram as compare signals. These signals can either be grouped at the bottom of the open diagram or the Compare signals can be 'interleaved' with the original signals. That is, the Compare signal for a given original signal will be inserted directly under the original signal. This behavior is controlled using the **View > Compare and Merge > Interleave Compare and Merge Signals** menu option. This tutorial will have this feature turned on.

To Compare the two timing diagrams:

1. Select the **File > Compare Timing Diagram...** menu option to launch the *Compare* dialog.  
Notice that the default file type is **Timing Diagram (\*.tim, \*.btim)**.
2. Select the **simulationResults.btim** file.
3. Click **Open** to compare the two diagrams.



Note that the Compare signals are placed immediately following the signals that they are being compared to since the **Interleave Compare and Merge Signals** option is on. Two of the signal labels, **Test.pin2** and **Test.pin3**, have turned red because there are differences on these signals.

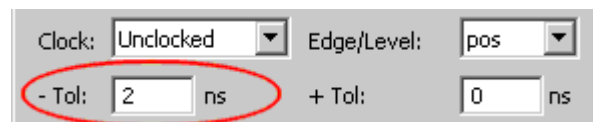
**Tip:** If the data sets being compared have slightly different naming schemes then the signals will not properly interleave because the program will not be able to properly match the signals for comparison. The **Edit > Search and Rename Signals** feature is handy for modifying one of the sets of signal names in the case. This feature performs pattern matching on the signal names and allows you to replace characters in the name, or append/remove a prefix or a suffix to the signal name.

## 2.4 Editing all Compare Signal Properties

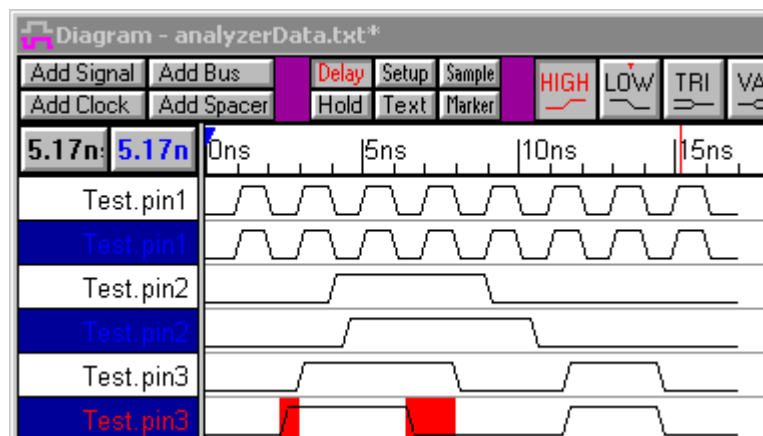
Typically it is desirable to have the Tolerance range in a comparison of two timing diagrams be uniform - the same for all of the Compare signals. All of the Compare signals in a diagram can be edited simultaneously without the need to manually select all of the signals.

Set the Tolerance for all of the Compare signals in the diagram:

1. Click the **Set All** button <sup>SET</sup>ALL or select the **View > Compare and Merge > Edit Compare Signals** menu option. This will automatically select all of the Compare signals in the diagram and launch the *Signal Properties* dialog. You can choose to deselect individual signals prior to modifying the signal properties - this may be useful if you want to specify a Tolerance range for all but one signal, for example.



2. Set the **-Tolerance** to 2ns.
3. Click the **Compare** button.



Notice that with the *-Tolerance* set at two, the **Test.pin2** signal no longer has any differences to report.

4. Set the *-Tolerance* back to 0ns for the next section and click the **Compare** button.

## 2.5 Methods of Examining the Differences Reported During a Comparison

There are three different ways to examine the differences that are reported:

- (1) a tab will be opened in the *Report* window that displays these differences
- (2) a tab delimited text file is saved in the same directory as the diagram used for comparison
- (3) the differences can be navigated graphically in the waveform window

### (1) The Differences Report Window Tab

If your *Report* window is not visible, select the **Windows > Report Window** menu option to bring the window to the foreground. Click the **Differences** tab to bring this report to the front of the *Report* window.

The differences found during the comparison are reported one per line, starting at time 0. In this example, the first difference found is on **Test.pin3**. The difference started at time 2.5ns and ended at time 3ns. The Reference Signal had state value of 0 while the Compare Signal had a State of 1. If you double click this line in the *Report* window, it will highlight the difference being reported.

Report - Differences					
	Signal Name	Start Time	End Time	Reference S...	Compare State
0	Test.pin3	2.5	3	0	1
1	Test.pin2	4	4.5	1	0
2	Test.pin3	6.5	8	1	0
3	Test.pin2	9	10.5	0	1

simulation.log waveperl.log Errors Differences Grep

### (2) The Differences File

The data displayed in the *Differences* tab of the *Report* window is loaded from a tab-delimited text file generated during the comparison. This data can be backed up and stored for later reference. The file in this example is named **analyzerData\_diff.txt** (formed from <referenceFileName>\_diff.txt). You can use the *Report* window to view this file by selecting the **Report > Open Report Tab** menu item to open the file.

### (3) Using the Graphical Interface

The Compare toolbar has five buttons. In order, from left to right, they are:



**Compare:** this button performs the same function as the **Compare** button in the *Signal Properties* dialog.



**Move to First Difference:** This button will move to the first difference found after time 0 in the diagram.



**Move Previous:** This button will move to the previous difference found in the diagram or on the signal.



**Move Next:** This button will move to the next difference found in the diagram or on the signal.




**Set All:** This button will select all of the Compare signals in the open diagram and open the *Signal Properties* to edit the Compare signal properties. This is the easiest way to control what type of comparison is performed and to set the tolerances for all signals.

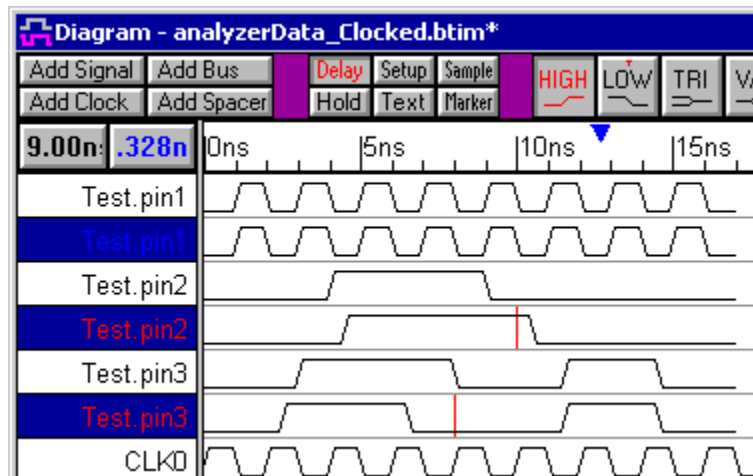
## 3. Performing a Clocked Comparison

A clocked comparison compares the signal states at clock edges instead of continuously. Add a clock to the current diagram:

1. Click the **Add Clock** button to add a clock named *CLK0* to the timing diagram.
2. Set the *Period* of the clock to **2ns** in the *Clock Properties* dialog and click **OK**.

Make the Compare Signals clocked:

3. Click the **Set All** button  on the Compare toolbar to select all of the Compare signals in the diagram and open the *Signal Properties* dialog.
4. Set the *Clock* to **CLK0** and set the *Edge/Level* to **pos**.
5. Click the **Compare** button to apply the changes and perform the new comparison.

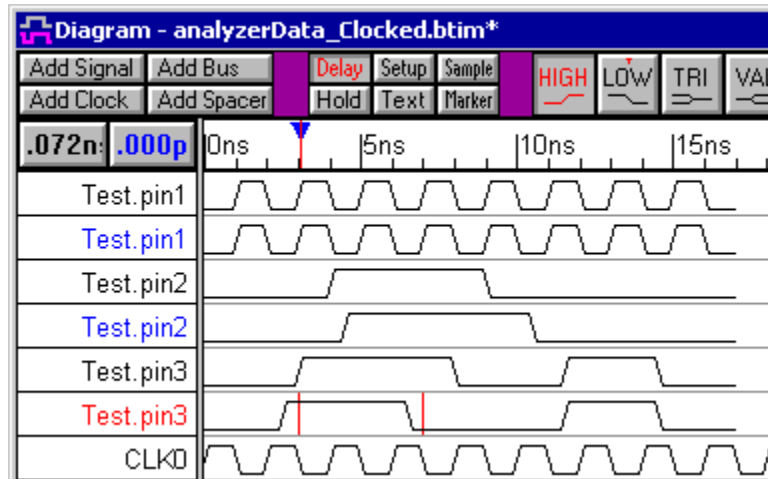


Notice that there are now only two differences in the diagram. These two differences occur on the



**8ns** (on **Test.pin3**) and **10ns** (on **Test.pin2**) rising edges of the clock. Change the clocking edge to the negative edges and re-compare:

- Set the *Edge/Level* to **neg** and click the **Compare** button to perform the new comparison.



There are still two differences in the diagram, but they have moved to the falling clock edges at **3ns** and **7ns** - both on **Test.pin3**.

## 4. Comparing During Clock Cycle Windows

The clocked comparison described in the previous step provides for comparison on clock edges. You can use a second clock, offset from the first, to create windows during which to compare during the high or low segments of the original clock instead.

### 4.1 Add Offset Clock to Diagram


To create windows comparisons consistent over the clock segments of the original clock we will use the same frequency and period that the first clock uses and offset the start time for the second clock.

- Add a second clock to the diagram and set the following in the *Edit Clock Properties* dialog:
  - Set the *Name* to **Offset\_Clock**.
  - Set the *Period* to **2ns**.
  - Set the *Starting Offset* to **.5**.
- Click **OK** to close the *Edit Clock Properties* dialog and apply the changes.

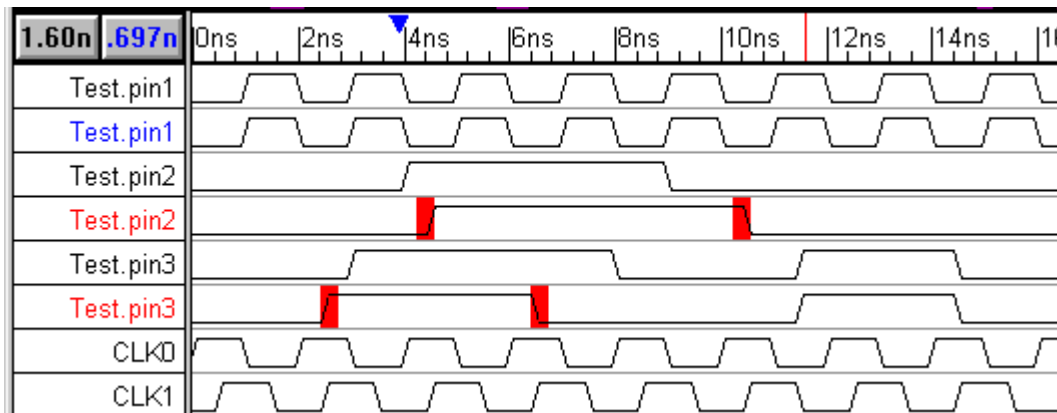
The **Offset\_Clock** now starts halfway through the first segment of **CLK0**.

### 4.2 Change the Compare Signals to Use the Offset Clock

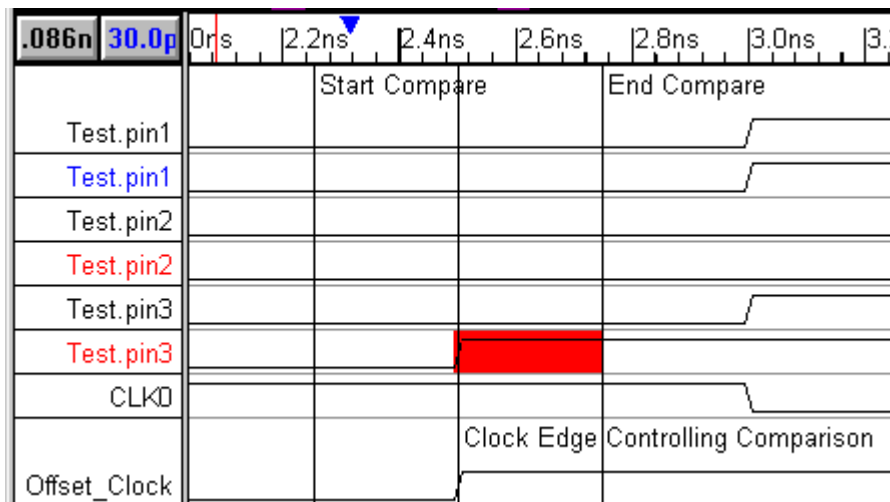
Next, change the three Compare signals to use the **Offset\_Clock** as the clocking signal and use the Tolerance settings to create the window for comparison:

- Click the **Set All** button  to edit the signal properties of all three Compare signals.
- Set the *Clock* to **Offset\_Clock**.
- Set the *Edge/Level* to **pos**.
- Set both *Tolerance* values to **0.25**.
- Click the **Compare** button to apply the changes and perform the comparison.

The diagram should now look like this:



You can zoom into the diagram to see a closer view of the window of comparison. In the image below, the starting and ending points of one of the comparison windows are marked, as well as the triggering clock edge:



## 5. Masking Clock Segments During Comparison

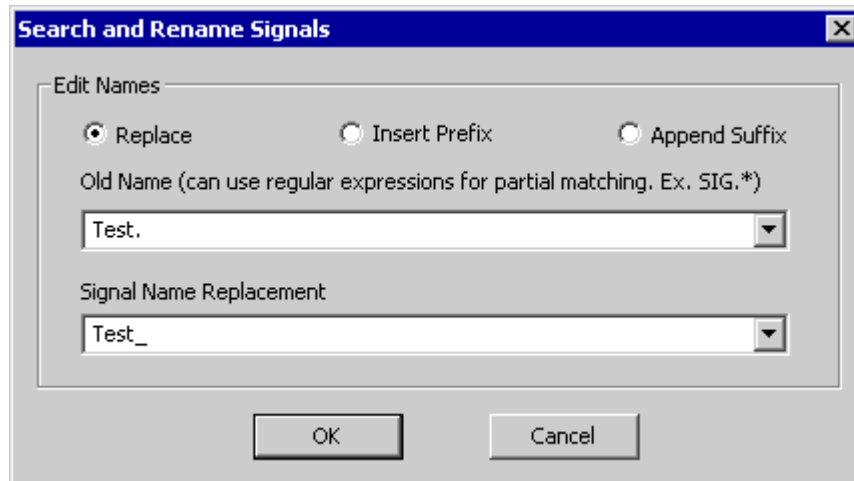
It is sometimes useful to mask some segments during comparison. In this example, we will create a Compare\_Enable signal that will specify the specific clock segments over which we want the comparison to be performed.

### 5.1 Search and Replace Signal Names

First, we will change the names of the signals in the diagram so that when we add the simulated signal, we do not get simulation errors (with the signal names Test.\* the simulator will look for a module that we will not have defined).

1. Press **<CTRL>A** to select all of the signals in the diagram. The Search and Rename signals feature will search the selected signals. If no signals are selected, then a dialog will appear asking if you would like to select all signals in the diagram.
2. Select the **Edit > Search and Rename Signals** menu option to open the *Rename Signals* dialog.

3. Enter **Test.** in the *Old Name* edit box.
4. Enter **Test\_** in the *Signal Name Replacement* edit box.
5. Click **OK** to close the dialog and rename the signals.



## 5.2 Add the Compare Enable

Next, add a signal to use for the compare enable:

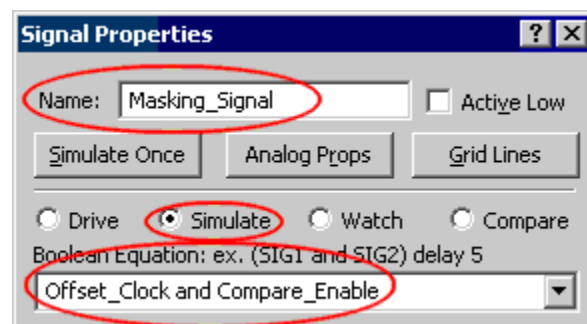
1. Add a new signal named **Compare\_Enable** to the diagram.
2. Draw a **Low** segment that ends around **4ns**.
3. Draw **High** segment that ends around **8ns**.
4. Draw another **Low** segment that extends beyond **16ns**.

In this example the High segment of the Compare\_Enable signal represents the time that we will allow the comparison to be performed. The segments will be masked where this signal is low.

## 5.3 Add the Simulated Masking Signal

Next, place a simulated signal in the diagram to use as the clocking signal for the comparison:


1. Add a signal to the diagram and set the following properties in the *Signal Properties* dialog:
  - Set the *Name* to **Masking\_Signal**.
  - Select the **Simulate** radio button (instead of *Drive*).
  - Enter **Offset\_Clock and Compare\_Enable** in the *Boolean Equation* edit box.
2. Click the **Simulate Once** button to generate the simulated signal.



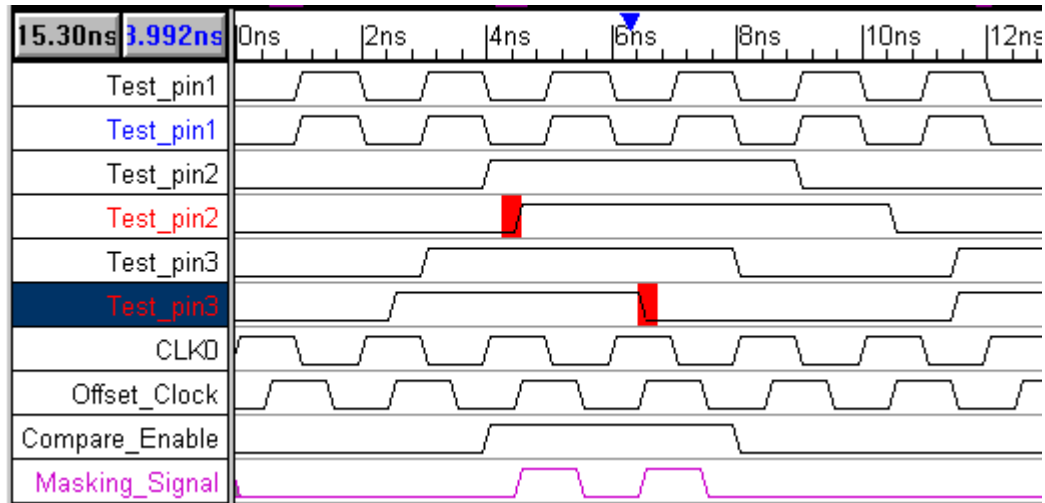
Notice that the beginning and ending of the simulated signal's waveform are low and the middle segment (when the Compare\_Enable is high) has the same value as the clock. These are the clock segments that are not masked. The comparison will be performed during these segments.

## 5.4 Use Masking Signal to Mask Clock Segments

Next we will edit the compare signals so that they use the new **Masking\_Signal** as a clock.

1. Click the **Set All** button  to edit all of the Compare signals in the diagram.
2. Set the *Clock:* to **Masking\_Signal**.
3. Click **Compare** to apply the changes and perform the comparison.

Note that the first and last differences that were previously in the diagram have now disappeared - they are masked from the comparison. The resulting diagram should look something like this:



## 5.5 Setting up Don't Care Regions

A related function to the masking signals is don't care regions. Sometimes you may not want to compare certain regions in time on a particular signal. One case would be on a bus signal where the actual data did not matter. If you want the compare function to skip a particular section, then just turn the waveform into a valid region on the reference signal.

**To create a don't care region:**

- On the reference signal (not the compare signal), highlight the section of waveform that you wish to exempt from comparison by left clicking on it.
- Click on the **VAL** button in the button bar at the top of the waveform window. This will change the waveform to a valid waveform.
- Rerun the compare function.

## 6. Adjusting the Time Difference Between Two Diagrams

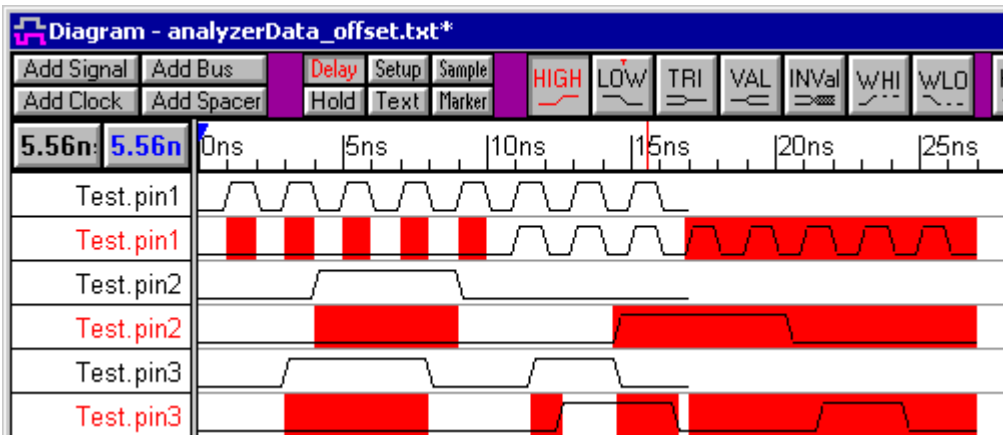
The time difference between two timing diagrams can be easily adjusted using the *Edit Waveform Edges* dialog. Next we will compare two data sets that have a time offset.

- Close the file you have been working with without saving and re-open the original **analyzerData.txt** file as a timing diagram (see Step 2.2: Open the Logic Analyzer Data File).

Note: If you previously saved your diagram during a previous step, simply delete the three compare waveforms from your diagram.

- Compare to the **simulationResults\_offset.btim** provided (refer to Step 2.3: Compare the Two Timing Diagrams as needed).

The resulting diagram should look something like this:



Notice that all three of the compare signals have a longer starting segment than their counterparts. This causes much of the diagram to show as differences, when in reality the two data sets have a time offset. This offset can either be removed from the compare signals or it can be added to the other signals. In this tutorial the offset will be removed.

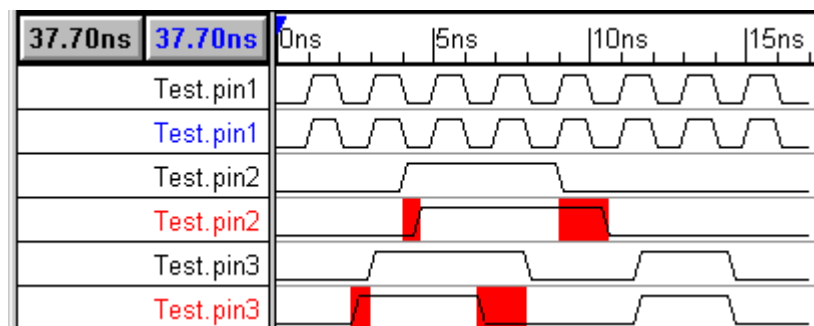
**To modify the time difference between two diagrams:**

- Select all three of the compare signals by clicking the names in the *Label* window.
- Select the **Edit > Edit Waveform Edges** menu option to open the *Edit Waveform Edge* dialog.
- Select the **Shift Edges in Range** radio button and enter the *time to shift* in the edit box below the radio button (both positive and negative numbers can be entered).
- Click **OK** to apply the changes and close the dialog.

Note: If no signals are selected when this operation is performed then all of the signals will be modified.

- Click the **Compare** button  on the *Compare* toolbar to recompare the signals.

**Now the diagram should look like the first diagram comparison:**



**Tip:** This dialog can also be used to perform frequency multiplication. Section 1.7: Editing Waveform Edges from an Equation the WaveFormer Pro manual describes this feature more in depth.

## 7. Summary

The Waveform Comparison option provides the ability to perform comparisons on individual signals or between two timing diagrams. The Tolerance settings allows you to watch for signal states to

match during a specified time range, while Clocked Comparisons allow you to compare states at specific times during the simulation. Without setting either a clocking signal or a tolerance specification, the comparison will be performed continuously on the signals. Finally, edge adjustment can be easily performed to manage time offsets between two data sets.

# Gigawave and WaveViewer Viewer Tutorial

This tutorial covers the following topics: opening a waveform file, the differences between opening and importing a file, saving a .btim file, creating a filter file to selectively load sets of signals from a waveform file, and available licensing options for enhancing WaveViewer.

## 1. Converting a vcd file into a btim file

When viewing non-native waveform formats, such as VCD files, we recommend first converting the file to SynaptiCAD's native BTIM format. The resulting compressed BTIM file will generally be around 200x smaller than the original file and will load much faster (for example, a BTIM file will typically load around 500x faster than an equivalent VCD file).

To convert a VCD to a BTIM file, follow the steps below:

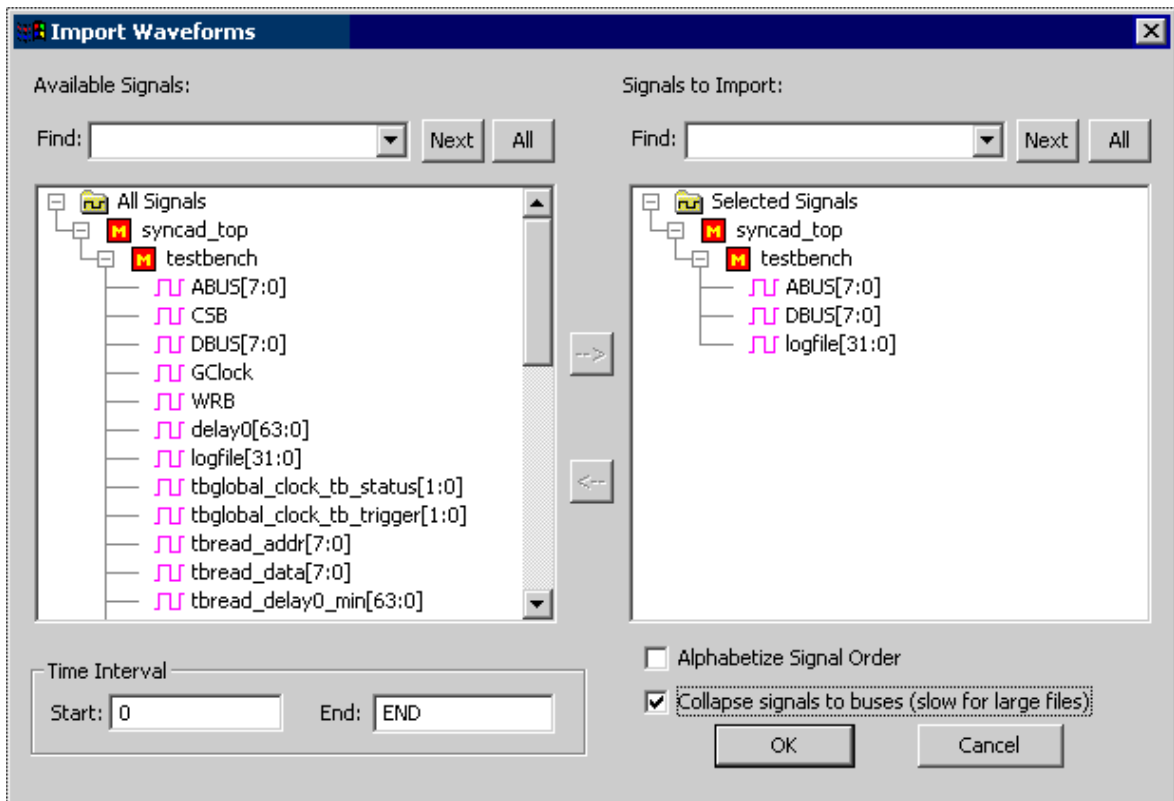
- Load the **exampleTim.vcd** VCD file into the viewer (this is a large file so it may take a few seconds):
  - Choose **File > Open Timing Diagram** menu option to launch the *Open File* dialog.
  - In the **File of type** dropdown, choose **Verilog Value Change Dump**.
  - Select the file **C:\SynaptiCAD\Examples\exampleTim.vcd**.
  - Click the open button to load the diagram.
- Save the diagram as a BTIM file
  - Choose **File > Save As** menu option to open the *Export Timing Diagram As* dialog.
  - In the **Save as type** dropdown, choose **Timing diagram – Binary**. This will change the filename to **exampleTim.btim**.
  - Click **Save** to close the dialog box.
- Close the diagram so we can load a subset of the signals in the next step.

## 2. Importing a subset of the Waveforms

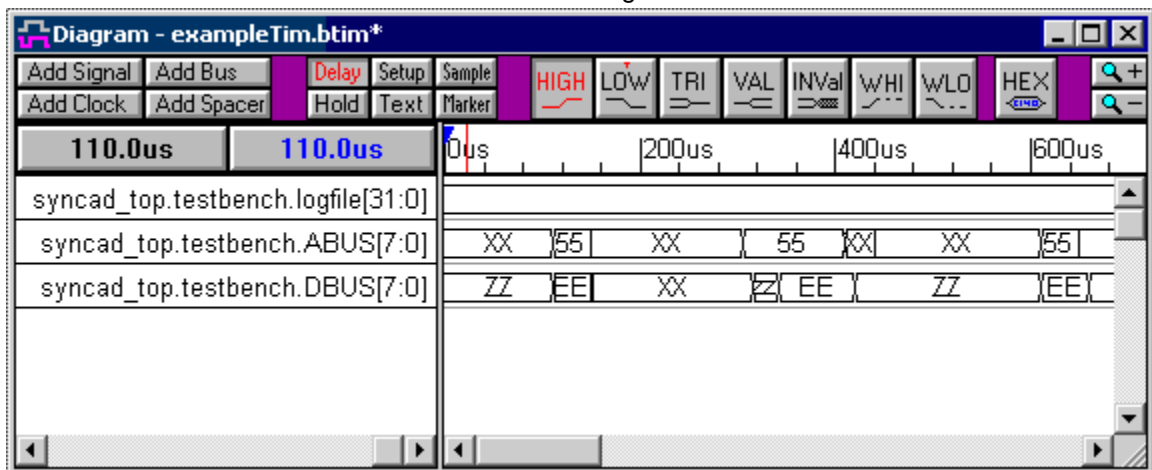
In the previous section, we loaded an entire waveform file using the **File > Open** menu. It is also possible to load a subset of the waveforms in a file by using the **Import/Export > Import Timing Diagram** menu.

To import a subset of waveforms from a file:

- Load the **exampleTim.btim** we created in the previous step using the **Import/Export** menu:
  - Choose the **Import/Export > Import Timing Diagram From** menu to open the *OpenFile* dialog.
  - In the **File of type** dropdown, choose **Timing diagram – Binary**.
  - Select the file **c:\SynaptiCAD\Examples\exampleTim.btim**.
  - Press the **Open** button to close this dialog and open the *Import Waveforms* dialog.
- In the *Import Waveforms* dialog, move the **ABUS[7:0]**, **DBUS[7:0]**, and **logfile[31:0]** signals from the **Available Signals** list to the **Signals to Import** list by selecting the signal names and clicking the -> button.
  - Note: Signals can be moved in groups by pressing the control button and selecting several signals at once.



- Note: The *Import Waveforms* dialog has several useful options that are not covered in the tutorial:
  - Checking **Alphabetize Signal Order** will cause the signals to be alphabetized when they are displayed in the viewer. Otherwise the signals will appear in the timing diagram in the same order that they are found in the file they are imported from.
  - Checking **Collapse signals to buses** causes all numbered signals (e.g. bus0, bus1, bus2) to be collapsed into virtual buses (bus[2:0]).
  - The **Time Interval** section allows waveform data to be limited to only a specific section of time.
- Click the **OK** button to close the dialog and load waveforms into WaveViewer



### 3. Creating a Filter File to selectively load signals

[WaveViewer/Gigawave Viewer TutorialWaveViewer\\_Gigawave\\_Viewer\\_Tutorial](#)



### 3. Creating a Filter File to selectively load signals

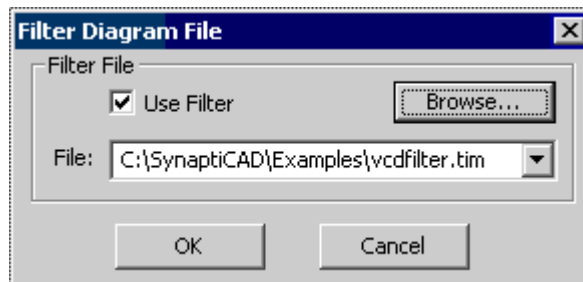
The **Set Filter File** feature provides the user with the ability to specify an optional "filter file" that controls what signals get imported from a waveform file (e.g. a VCD file) and the order in which they get imported. The signal properties, such as MSB, LSB, and direction, are also controlled by the signal information that is stored in the filter file. Filter files do not contain any waveform data. Below we will change the ordering of the signals we imported in step 2 and save them into a filter file.

To create a filter file using WaveViewer:

- Select the **File > Save As** menu option to open the *Save As* dialog.
- In the **Save As type** dropdown, select **Waveform filter (\*.tim)** if it is not already selected.
- Type **vcdfilter** as the name of the filter file in the **File name** edit box.
- Click the **Save** button to create the filter file.

Once a filter file is created, it needs to be set as the current filter file for the Filter file feature to be enabled:

- Select the **Options > Set Filter File** menu option to open the *Filter Diagram File* dialog.
- Check the **Use Filter** checkbox.
- Select the file by clicking the **Browse** button and finding where you saved it in the previous section.
- Click **OK** to close the dialog.



When a filter file is set, only the filtered signals will be loaded. The filter file will operate on all files that are loaded, regardless of the file format.

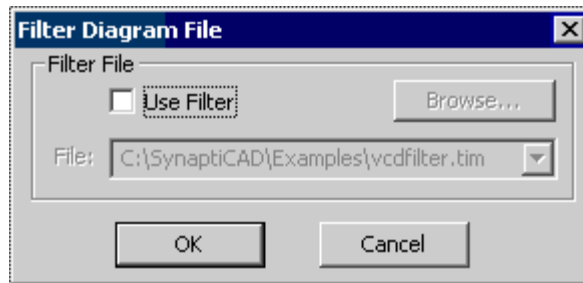
To try this out, let's load the original BTIM file with the filter enabled:

- Choose the **File > Open timing diagram** menu option to launch the *Open File* dialog.
- Select the file **C:\SynaptiCAD\Examples\exampleTim.btim**.
- Click the **Open** button to load the diagram. Note that only the signals that match the signal names in the filter diagram have been loaded.

To disable the Filter file feature:

- Select the **Options > Set Filter Diagram File** menu option to open the *Filter Diagram File* dialog.
- Uncheck the **Use Filter** checkbox.
- Click **OK** to close the dialog and disable the Filter File feature. Any diagrams loaded after this point will load normally, without filtering.

- Close the currently open timing diagram, then choose the **File > Open Timing Diagram** menu and reload the **exampleTim.btim** file again. Note that all signals are loaded from the diagram because we've disabled the filter file.



## 4. Show and Hide Signals in the display

Instead of selectively loading signals, it's often easier to load all the signals initially and then hide all but the signals of interest.

Follow the steps below to hide a set of signals:

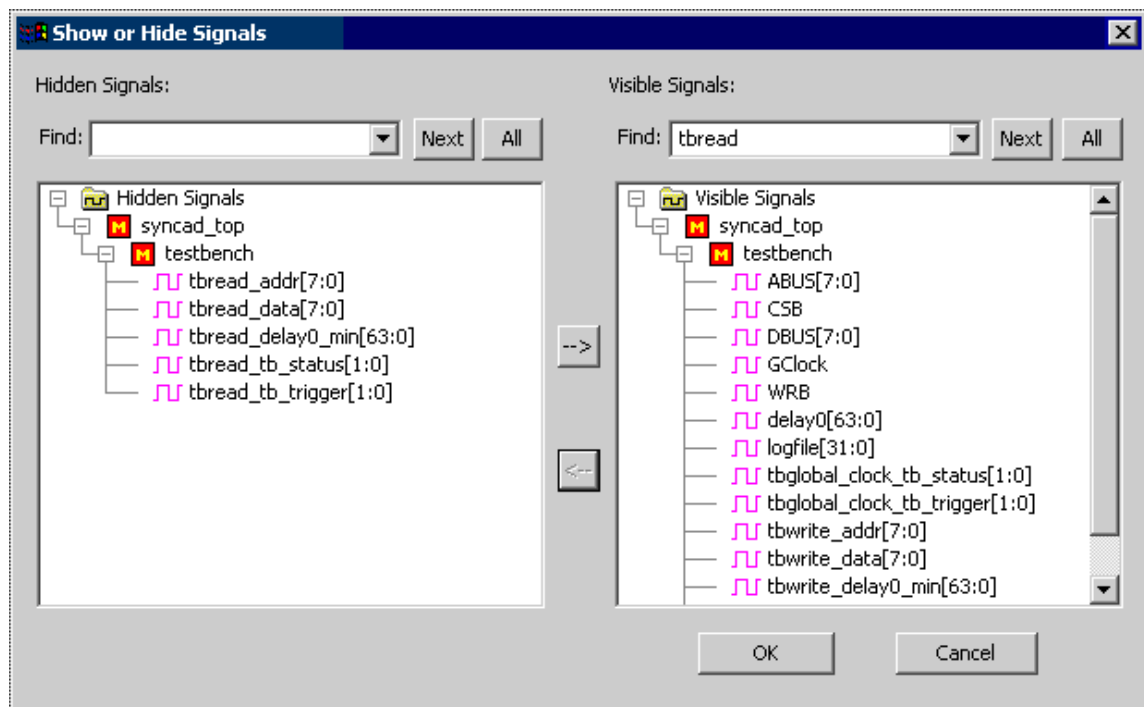
- Click on the **View>Show and Hide Signals** menu option to open the *Show or Hide Signals* dialog. The hidden signals are shown on the left side and visible signals (in this case, all signals) on the right.
- Select signals on the right side (**Visible Signals**) and press the ←← button, followed by the **OK** button to hide these signals.

To make the signals visible again:

- Reopen the dialog and select **All Signals** from the list on the left side (**Available Signals**) and press the →→ button, followed by the **OK** button, and the signals will reappear in the diagram window.

You can also use the **Find** edit boxes to select a set of signal names that match a particular regular expression. Follow the steps below to hide all the signals that have "tbread" in their signal name:

- Type "tbread" in the **Find** edit box in the **Visible Signals** section of the dialog and press the **All** button to select the matching signals.
- Press the ←← button to move these signals to the list of signals to be hidden.
- Press **Ok** to hide these signals.



## 5. Options: Gigawave, Waveform Comparison, Transaction Tracking

Several optional features can be purchased to extend the power and functionality of the free WaveViewer:

The **GigaWave feature** converts WaveViewer into GigaWave Viewer, a high capacity waveform viewer capable of handling multi-gigabyte VCD files. Without GigaWave, WaveViewer is limited to diagrams of less than 10,000 signals and less than 1 million waveform state changes. The GigaWave feature also comes with a PLI-based library that can be integrated with your favorite simulator to directly generate highly compressed BTIM files (no intermediate dump to VCD is required). Using direct BTIM waveform dumping can speed up simulation by up to 3x compared to the same simulation using an ordinary VCD dump because of the reduction in file I/O, and the resulting files are generally 200x smaller.

The **Waveform Comparison feature** lets users compare waveforms for two timing diagrams or individual signals. This feature is exceptionally useful comparing two different simulation runs, as well as for comparing logic analyzer data to a simulation run. The specific regions where waveforms differ will turn red when the two waveforms are compared. In addition to standard waveform comparison, where all differences are detected, the comparison feature also supports "Clocked comparison" where waveforms are only checked at clock edges. Tolerances can also be specified to determine what constitutes a significant difference between two waveforms. For more information on using the Waveform comparison feature, please refer SynaptiCAD On-Line Help > Bug Hunter Pro and VeriLogger Pro Table of Contents > Chapter 5: Waveforms and Test Bench Generation > 5.4 Waveform Comparisons (Optional Features)

The new **Transaction Tracking feature** lets users view waveform data as "transactions" instead of as just signals. The user specifies PSL Sugar expressions that describe what transactions to search for, and the transaction tracker search engine finds matching transaction records and displays them graphically in the timing diagram window.

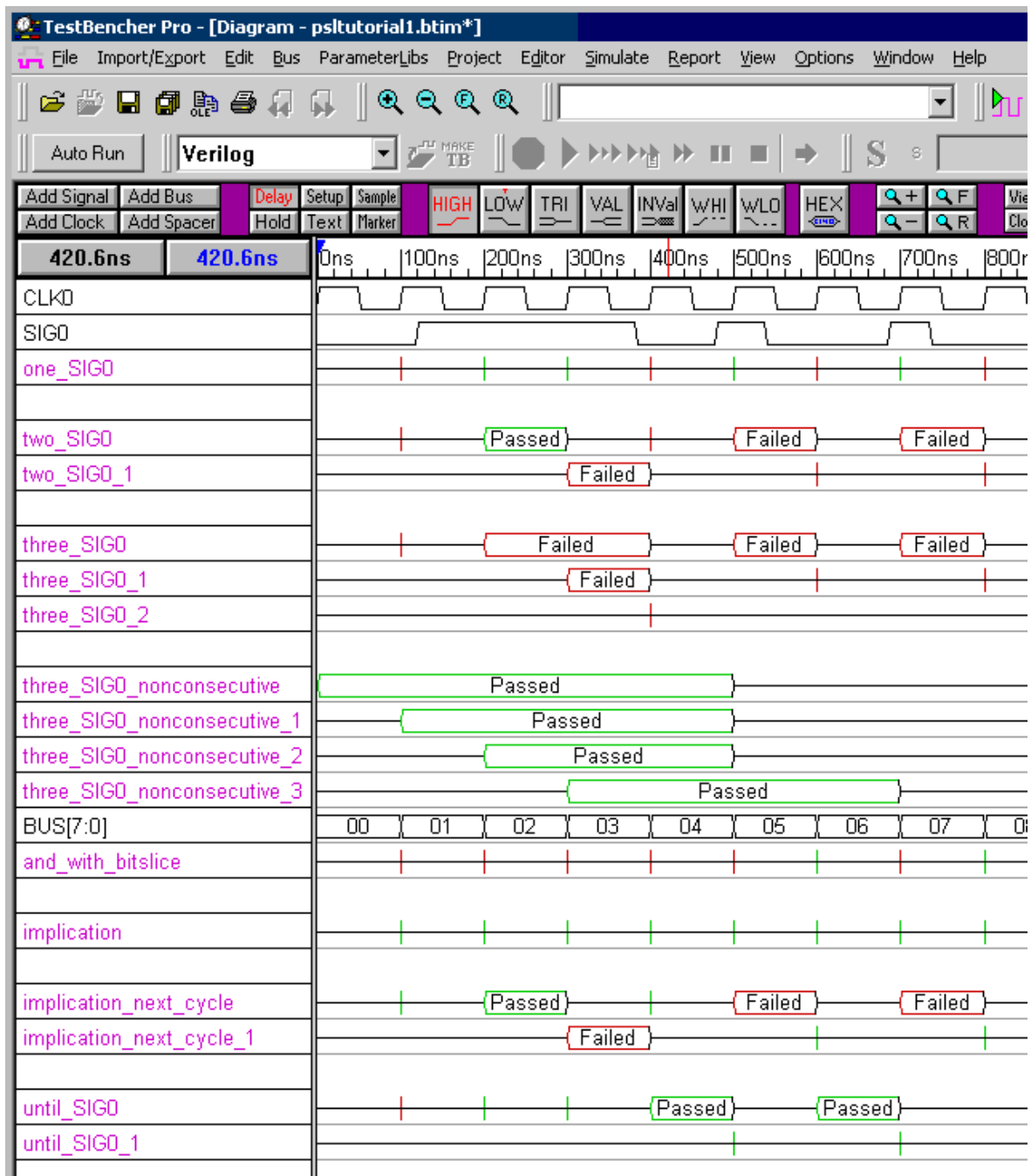
## 6. Waveviewer/GigaWave Viewer Tutorial Summary

Congratulations! You have completed the WaveViewer and GigaWave Viewer tutorial. In this tutorial we introduced various techniques for opening, importing, saving and converting .vcd files into btim files for faster file loading. We created a filter file and used it to selectively load signals (filter files can also be used to reorder signals and override the default properties on the loaded signals). We also covered the differences between GigaWave Viewer and WaveViewer and briefly discussed the optional Waveform Comparison and Transaction Tracking features. More information on these topics can be found in the online help.

## Transaction Tracker Tutorial

This tutorial explores semantic differences between some of the most commonly used PSL operators. The assertions in this tutorial have been kept very simple, so that it is easy to see the differences between the operators. It is important to understand these distinctions before attempting to write practical, real world assertions. You will need a Transaction Tracker License to complete this tutorial

For all the examples in this tutorial, we use two input signals (SIG0 and BUS[7:0]) and all the assertions are clocked off the positive edge of CLK0. The assertions will make a new match attempt on each positive edge of CLK0, even though previous match attempts by the assertion are still in progress. When such overlapping match attempts occur, additional "overflow" signals will be created to display the resulting transaction records without overlapping them on a single result signal. These overlap signals are created dynamically as needed during the simulation, and get cleaned up automatically at the beginning of any new simulation run.



- 1) Open the Example File
- 2) Matching all occurrences of simple pattern `one_SIG0 = {SIG0}`
- 3) Match consecutive occurrences with Concatenation Operator `two_SIG0 = {SIG0;SIG0}`
- 4) Match with consecutive repetition Operator** `three_SIG0 = {SIG0[*3]}`
- 5) Match with non-consecutive Repetition Operator `three_SIG0_nonconsecutive = {SIG0[=3]}`

- 6) Implication operator `implication = {{SIG0} |-> {SIG0}}`
- 7) Implication Next-Cycle operator `implication_next_cycle = {{SIG0} |=> {SIG0}}`
- 8) PSL Property `until_SIG0 = ((BUS > 2) until SIG0)`

[Return to Tutorial Table of Contents](#)

## 1) Open the Example File

First, open the timing diagram file `c:\SynaptiCAD\Examples\sugar\psltutorial.btim` using the menu option **File > Open Timing Diagram**. This file contains all the assertions discussed in this tutorial. As we go over each assertion, you can double click on the associated signal in the timing diagram to see how the assertion was specified.

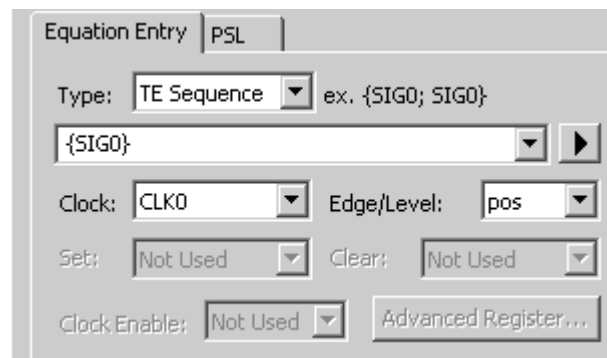
## 2) Match all occurrences of a simple pattern

**one\_SIG0 = {SIG0}**

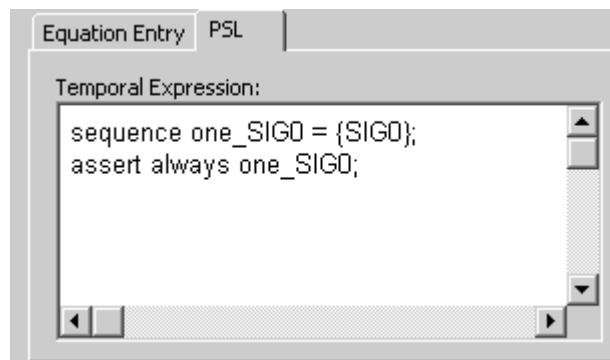
This sequence matches a single occurrence of SIG0 on the positive edge of CLK0. Since it is a simple Boolean check, its transaction records have no time width and they display as spikes. It fails to match at time 100, so the first spike is red, then passes for two cycles because SIG0 is high, then fails again at time 400.



Double click on **one\_SIG0** to open the *Signal Properties* dialog. Note, that under the **Equation** tab, the **TE Sequence** is chosen. Also note that the equation is just **{SIG0}**.



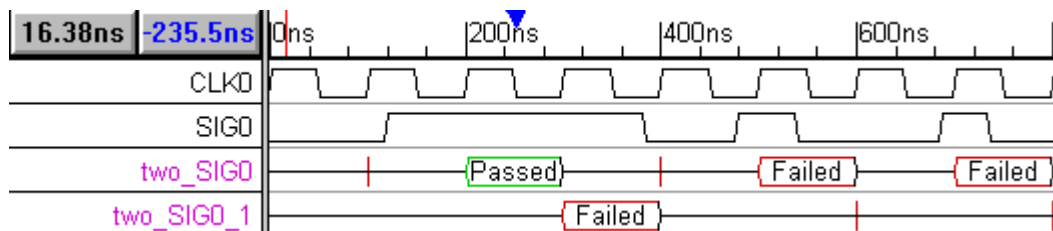
If you click on the **PSL** tab, you can see the actual PSL code that is generated from the information in the TE Sequence equation and the other controls in the **Equation** tab.



### 3) Match Consecutive Occurrences with Concatenation Operator

**two\_SIG0 = {SIG0;SIG0}**

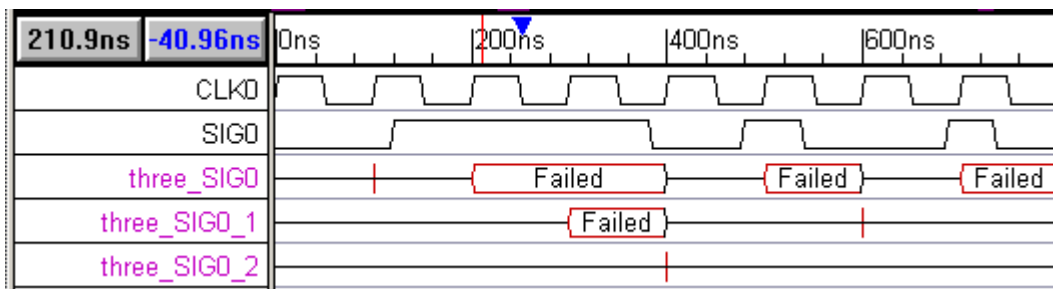
This sequence matches two consecutive occurrences of SIG0. The first attempt at time 100 fails immediately, because SIG0 is low, resulting in a red spike. The second attempt begins to match at time 200, and then succeeds at time 300, since SIG0 is still high at time 300. A new attempt is also begun at time 300, creating a transaction record that overlaps with the transaction record that started at time 200, so the time 300 transaction is placed on an overflow signal (two\_SIG1\_1). SIG0 goes low before time 400, so the time 300 transaction fails at time 400. A new transaction is also begun at time 400, which fails immediately, resulting in a red spike at time 400 on two\_SIG0.



### 4) Match with consecutive repetition Operator

**three\_SIG0 = {SIG0[\*3]}**

This sequence matches against 3 consecutive matches of SIG0. Note that this sequence could also have been specified as {SIG0;SIG0;SIG0}, but this gets awkward for specifying large numbers of repetitions of a pattern. There are no cases where SIG0 is high for 3 consecutive cycles, so all the match attempts eventually fail. The transaction record at time 200 holds for the first 2 cycles, then fails in the third cycle. Because it takes 3 clocks to fail, and two new matches are attempted during this time, two overflow signals are required to display all the transaction records without overlap.

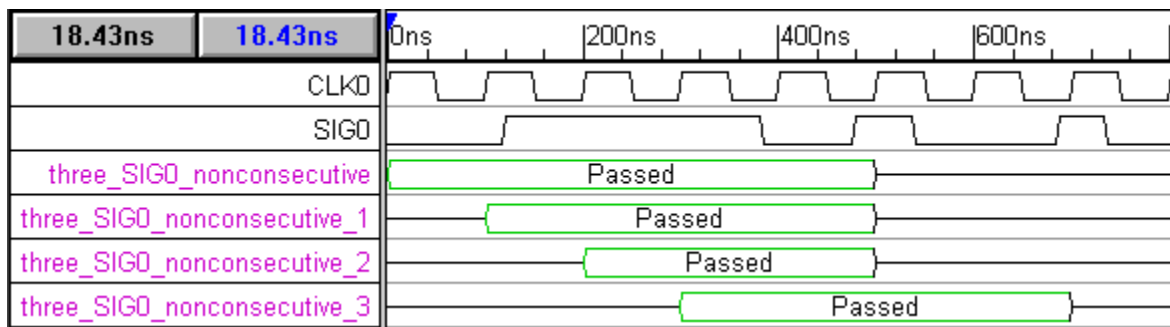




## 5) Match with non-consecutive Repetition Operator

**three\_SIG0\_nonconsecutive = {SIG0[=3]}**

This sequence looks for 3 **nonconsecutive** occurrences of SIG0 since we used [=n] instead of the consecutive operator [\*n]. Note that SIG0 does not have to be true during the first cycle of the match attempt, this operator only that we eventually see 3 clocks during which SIG0 is true, so we end up with a large number of successful matches.

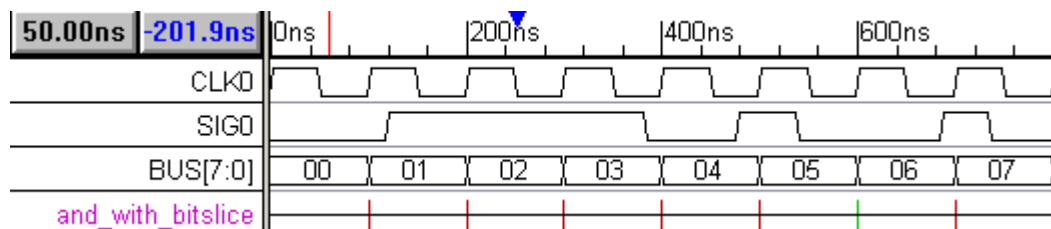


## 6) Bit-slices and the Boolean operators

**and\_with\_bitslice = {BUS[1:0] & BUS[3:2]}**

This assertion uses the **bit slice** operator to AND together the two least significant bits of BUS with the next two bits of BUS. The result of this operation is considered true if the value is greater than 0, so for Boolean truth, it is equivalent to:  $\{(BUS[1] \& BUS[3]) \mid (BUS[0] \& BUS[2])\}$

The assertion matches at time 600 (when BUS has a value of 5, so  $BUS[3:2] = 'b01$  and  $BUS[1:0] = 'b01$ ) and at time 800 (when BUS has a value of 7, so  $BUS[3:2] = 'b01$  and  $BUS[1:0] = 'b11$ ).

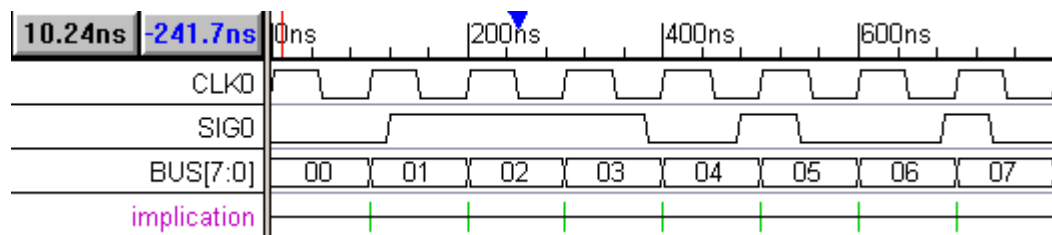


## 7) Implication operator

**implication = {{SIG0} |-> {SIG0}}**

The implication operator guarantees that if the left hand operand holds (the sequence specified by the left hand operand succeeds), then the right hand operand will hold. If the left hand operand does not hold, the implication operator will succeed regardless of whether the right hand operand holds.

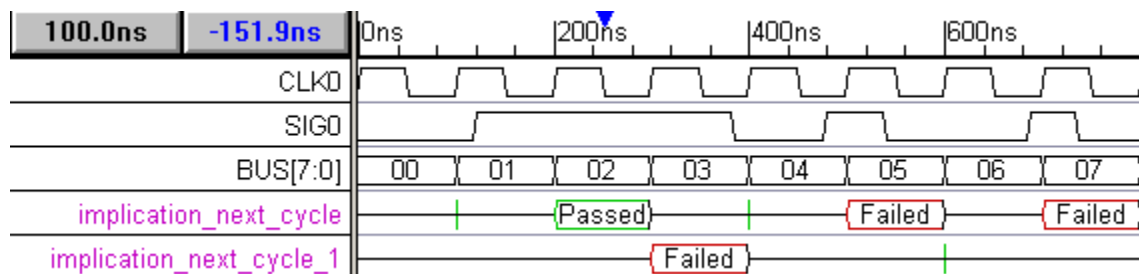
In this assertion we use the implication operator to ensure that: if SIG0 is true during a clock cycle, then SIG0 is true during this clock cycle. Therefore, this is not very a useful sequence as the sequence will always hold and matches immediately, regardless of whether SIG0 is true or false. Please note how this differs from the earlier {SIG0} assertion, which passes or fail each clock cycle based on whether SIG0 is true or not.



## 8) Implication Next-Cycle operator

**implication\_next\_cycle = {{SIG0} | => {SIG0}}**

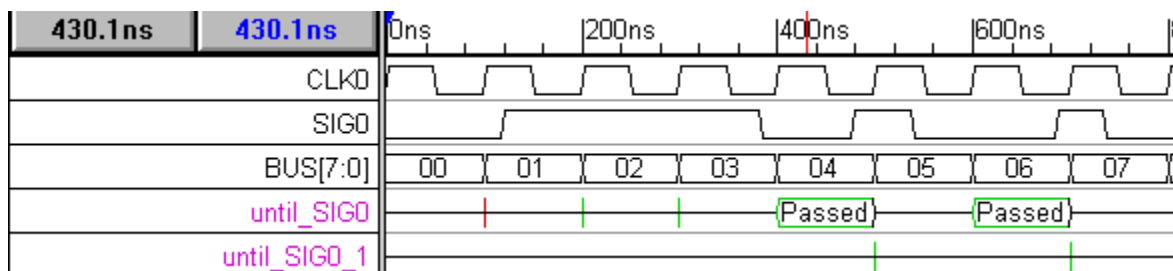
This assertion looks very similar to the previous one, but the implication operator  $\rightarrow$  has been replaced by the implication\_next\_cycle operator  $\Rightarrow$ . So this assertion is equivalent to:  $\{\{SIG0\} \rightarrow \{true; SIG0\}\}$ . It checks that if SIG0 is true in the current cycle, it should be true in the next clock cycle. So if SIG0 is false at the start of a match attempt, that match attempt succeeds immediately. If SIG0 is true during the start of a match attempt, the match will succeed if SIG0 is true during the next cycle, or fail if SIG0 is false during the next cycle. Compare this to the operation of the earlier assertion  $\{SIG0; SIG0\}$  which fails when SIG0 is not true at the beginning of a match attempt.



## 9) PSL Property

**until\_SIG0 = ((BUS > 2) until SIG0)**

For this signal we used a PSL property instead of an assertion, so the assertion body is surrounded by parenthesis instead of curly brackets. In the *Signal Properties* dialog, the equation edit box is **TE Property**. The property checks each cycle to see if the value of BUS is greater than 2 until SIG0 becomes true. Note that for the matches attempted at time 500 and 700, SIG0 is true on the initial clock cycle of the match, so the transaction record succeeds immediately.



# Index

## - A -

- Adding 12, 14, 23, 24
  - clock 12
  - parameters 24
  - setups 23
  - signals 14
- Analog 60, 61, 62, 63, 64
  - CapCharge function 64
  - CapDischarge function 64
  - Ramps by a function 62
  - Ramps with mouse 60
  - SinEnd function 63
  - SinStart function 63
  - Spice output 59
  - Step signals 61
  - step voltage export 59
- Analog Display checkbox
  - piecewise linear 59
- Analog Signals 59

## - B -

- Base Time Unit 11

## - C -

- CapCharge function 64
- CapDischarge function 64
- Clocks 12, 23
  - adding 12
  - adding setup 23
- Comparison 154
  - Masking Segments 154
- Compile SynaptiCAD Library Models 134

## - D -

- Drawing Waveforms 15

## - E -

- Export Diagram as an HDL Test Bench 90

## - L -

- Label Equation 62, 63, 64
  - CapCharge function 64
  - CapDischarge function 64
  - Ramp 62
  - SinEnd function 63
  - SinStart 63
  - SinStart function 63

## - M -

- Masking Segments During Comparison 154
- Match all occurrences of a simple pattern 167
- Match Consecutive Occurrences with Concatenation Operator 168
- Moving Signals 33

## - O -

- Open the Example File 167

## - P -

- Parameters 24
  - adding 24

## - R -

- Ramp function 62
- Ramp Signals 59
- Reordering Signals 33

## - S -

- Setups 23
  - adding 23
- signals 14, 15, 16, 33
  - adding 14
  - drawing waveforms 15
  - editing waveforms 16

signals 14, 15, 16, 33  
  moving 33  
  ramp 59  
  reordering 33  
  step 59  
SinEnd function 63  
SinStart 63  
SinStart function 63  
Step Signals 59

## **- W -**

Waveform Comparison 154  
  masking segments 154  
Waveforms 15  
  drawing 15  
  editing 16