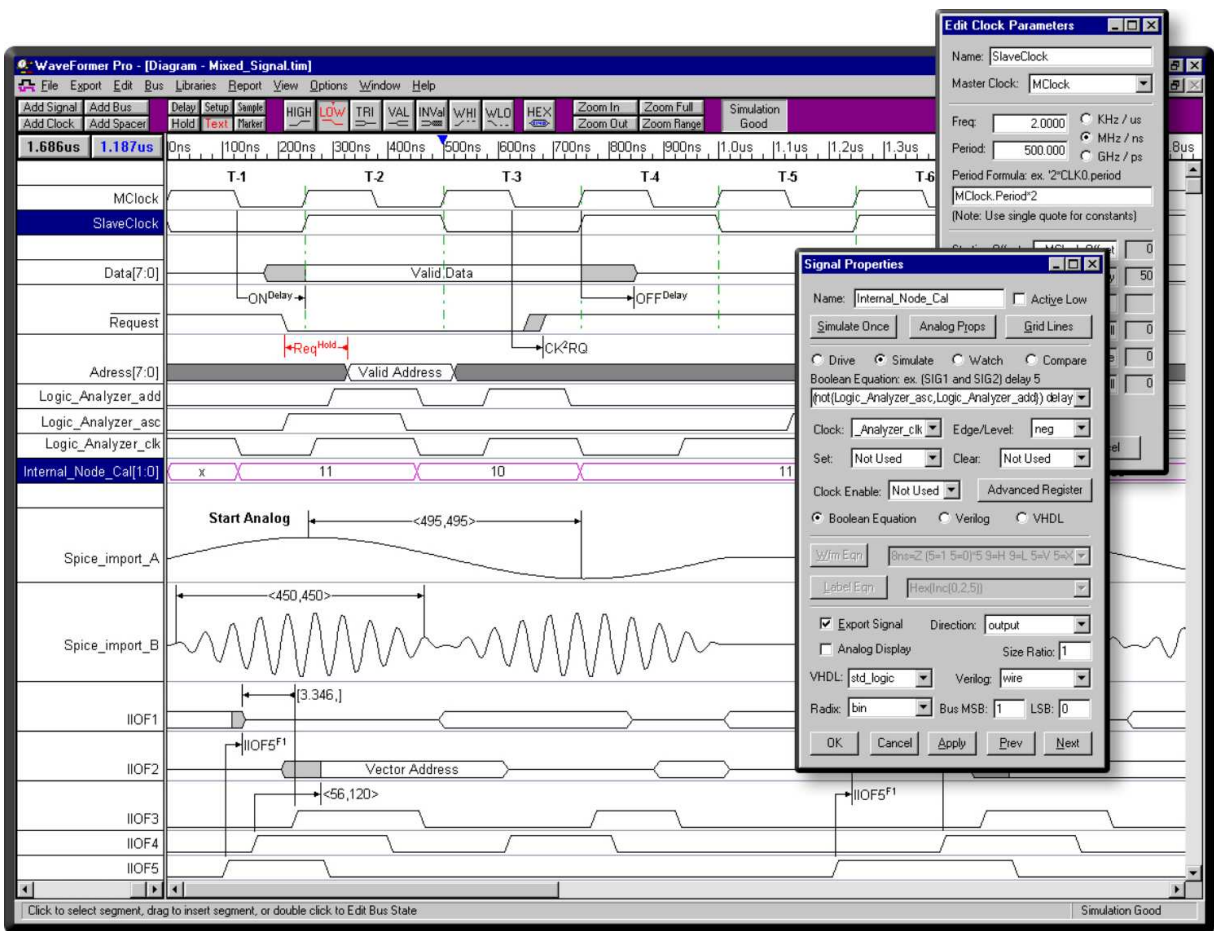


# SynaptiCAD Tutorials



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

## **SynaptiCAD Tutorials (rev 10.0) copyright 1994-2005 SynaptiCAD**

### Trademarks

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

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

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

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

For latest product information and updates contact SynaptiCAD at:

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

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

phone: (540)953-3390

## Table of Contents

<b>Table of Contents .....</b>	<b>3</b>
<b>Introduction .....</b>	<b>6</b>
<b>Basic Drawing And Timing Analysis.....</b>	<b>8</b>
1) Set the Base Time Unit .....	9
2) Set the Display Time Unit .....	10
3) Add the Clock .....	10
4) Add Signals .....	11
5) Drawing Signal Waveforms .....	12
6) Edit Signal Waveforms .....	13
7) Adjust Diagram to Match Figure .....	14
8) Moving and Reordering Signals .....	14
9) The Right Mouse Button .....	15
10) Add the D Flip-Flop Propagation Delay .....	15
11) Add the Inverter Propagation Delay .....	17
12) Add the Setup for the Dinput to Clock .....	18
13) Add a Free Parameter .....	19
14) Using Formulas and Constants .....	20
15) Summary .....	20
<b>Interactive HDL Simulation Tutorial.....</b>	<b>22</b>
1) Interactive HDL Simulation .....	22
2) Generate Waveforms From Boolean Equations .....	23
3) Boolean Equations with Delays .....	24
4) Register and Latch Signals .....	26
5) Set and Clear Lines .....	27
6) Multi-bit Equations .....	28
7) Experiment with Behavioral HDL Code .....	29
8) Summary .....	31
<b>Waveform Generation And Bus Tutorial .....</b>	<b>32</b>
1) Generate Waveforms from Temporal Equations .....	32
2) Bus Overview .....	33
2.1) Creating Virtual Buses .....	34
2.2) Creating Group Buses .....	35
2.3) Creating Simulated Buses .....	37
3) Summary .....	37
<b>Display and Documentation Tutorial .....</b>	<b>38</b>
1) Controlling the Parameter Display String .....	38
2) Repeating Parameters .....	40
3) Editing Waveform Edges From an Equation .....	41
4) Drag and Drop Parameter End Points .....	41
5) Adjusting the Vertical Placement of a Parameter .....	42
6) Clock Jitter and Display .....	42
7) Markers .....	43
8) Edit Text Blocks .....	45
9) Summary .....	46
<b>Advanced Modeling and Simulation .....</b>	<b>47</b>

1) Set up a New Timing Diagram .....	48
2) Generate the Clock, Draw Waveforms, and Use Waveform Equations .....	48
3) Modeling State Machines .....	50
4) Checking for Simulation Errors .....	51
5) Incremental Simulation .....	52
6) Modeling Combinatorial Logic .....	53
7) Entering Direct HDL Code for Simulated Signals .....	53
8) Modeling n-bit Gates .....	54
9) Incorporating Pre-Written HDL Models into Waveformer Simulations .....	54
10) Modeling the Incrementor and Latch Circuit .....	55
11) Modeling Tri-State Gates .....	56
12) Debugging External Verilog Models .....	56
13) Verify the Histogram Circuit .....	56
14) Controlling the Length of the Simulation .....	57
15) Editing Verilog Source Files .....	57
16) Simulating Your Model with Traditional Verilog Simulators .....	58
17) Summary .....	58
<b>Parameter Libraries.....</b>	<b>59</b>
1) Adding Libraries to the Project's "Library Search List" .....	59
2) Setting Library Specifications .....	61
3) Startup Library Configuration .....	61
4) Referencing Parameters in Libraries .....	61
5) Using Macros to Examine Tradeoffs Between Different Libraries .....	63
<b>Advanced HDL Stimulus Generation.....</b>	<b>65</b>
1) Getting Started .....	65
2) Default Mappings: Hex and Binary Translations .....	66
3) Generating Verilog Code .....	67
4) VHDL - Advanced Data Types .....	67
5) Exporting to VHDL .....	67
<b>Basic Verilog Simulation .....</b>	<b>69</b>
Part 1: Project Management and Simulation .....	69
1.1) Add Files to the Project .....	69
1.2) Build the Tree and Use the Editor Window .....	70
1.3) Simulate the Project .....	70
1.4) Watch and View Internal Signals .....	71
1.5) Save the Project, Waveforms and Source Code .....	71
Part 2: Graphical Test Bench Generation .....	72
2.1) Remove TestBench Model and Clean Results Diagram .....	72
2.2) Build the Project and Examine the Black Signals .....	72
2.3) Use the Debug Run and Simulation Mode .....	73
2.4) How to Draw Waveforms .....	73
2.5) How to Edit Waveforms .....	73
2.6) Draw the Stimulus Waveforms .....	74
2.7) Simulate Using the Auto Run Simulation Mode .....	74
2.8) Import and Generate Waveforms .....	75
3) Breakpoints, Stepping and Tracing .....	76
<b>Reactive TestBench Tutorial.....</b>	<b>77</b>
1) Overview .....	77
2) The Model Under Test .....	77

3) Create Signals .....	78
4) Draw Single Write (without waiting on TRDY) .....	79
5) Add Wait for TRD Assertion .....	79
6) Draw Single Read .....	81
7) Add a Sample to Verify Data Read from MUT .....	81
8) Drive Data Using a Test Vector Spreadsheet File .....	82
9) Create For-Loop to Perform Multiple Writes and Reads .....	83
10) TestBencher Pro Transactor - Add Address Argument .....	83
11) Alternatives .....	84
<b>TestBencher Pro: Basic Tutorial .....</b>	<b>85</b>
1) Create a Project .....	85
2) Create the Write Cycle Transaction Diagram .....	87
3) Create the Read Cycle Transaction Diagram .....	88
4) Create the Initialize Transaction Diagram .....	90
5) Modify the Sequencer Process .....	91
6) Generate Test Bench and Simulate .....	94

# Introduction

There are several tutorials shipped with all versions SynaptiCAD's software. These 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.

	Timing Diagrammer	WaveFormer	DataSheet	TestBencher	VeriLogger BugHunter
Basic Drawing and Timing Analysis	****	****	****	***	
Interactive HDL Simulation		****	*	*	
Waveform Generation and Bus	****	****	***	*	*
Display and Documentation	****	**	****	**	
Advanced Modeling and Simulation		***	*	*	
Parameter Libraries	**	**	**		
Advanced HDL Stimulus Generation		***			
TestBencher Pro: Basic Tutorial				****	
Basic Verilog Simulation					****

**Table 1: Determining which tutorials to perform**

After installing one of SynaptiCAD's products, choose the **Help > Tutorials** menu to open the tutorial help page. Each tutorial can be printed by using the print command in the help window.

**Evaluators:** If you are evaluating the product we recommend that you do at least the General Design tutorials. These will give you a good idea of the flexibility of the product. If you design in VHDL or Verilog you should also look at the HDL tutorial and the TestBencher tutorial.

## General Design Tutorials

The **Basic Drawing And Timing Analysis** tutorial explains the basic timing diagram editing environment: how to set the base time unit and the display time unit of a timing diagram; how to draw and edit signals, delays, and setups; and how to perform time measurements. This tutorial is essential to anyone evaluating or learning to use any SynaptiCAD product.

The **Interactive HDL Simulation** tutorial explores the various time saving techniques of generating waveforms using equations. This tutorial explains how the Interactive HDL Simulator can simulate Boolean Equations with delays, register and latched signals, and behavioral Verilog code. It also demonstrates how instant re-simulations can be performed when input waveforms are modified, so that tedious calculations, once done by hand, are now automatically generated.

- The **Waveform Generation and Bus** tutorial demonstrates techniques for working with multiple bit signals. These techniques include generating the waveforms and automatically labeling those waveforms using equations. This tutorial also covers how to create Virtual, Group, and Simulated buses for solving different design problems. These features augment the drawing environment and provide a quick way to generate signals without having to draw each signal transition.
- The **Display and Documentation** tutorial demonstrates different methods for controlling the information that is displayed by delays, setups, holds, and samples. It also describes how to manipulate the vertical placement of a parameter, and how to change the transition attachments. These features allow you to control the information displayed and the appearance of the timing diagram.
- The **Advanced Modeling and Simulation** demonstrates how WaveFormer Pro can quickly model and simulate a digital system of moderate complexity. This tutorial will teach you how to model state machines using Boolean equations, use the Report window to find simulation errors, enter direct HDL code, model tristate gates, model n-bit gates, and call external HDL models. All WaveFormer and TestBencher Pro users should do this tutorial.

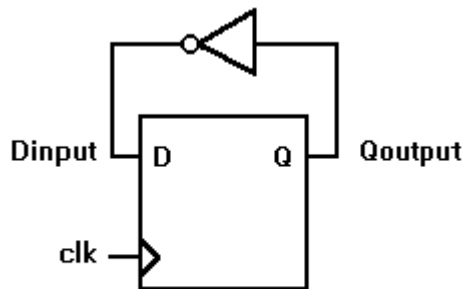
### Specialized Feature Tutorials

- The **Parameter Libraries** tutorial covers the use of libraries and macro lists. This tutorial is important to do before starting a large project. Using Libraries and macro lists can save you a great deal of time if they are configured properly.
- The **Advanced HDL Stimulus Generation** tutorial covers the basic concepts of HDL stimulus generation, such as graphical waveform states, language-independent hexadecimal and binary bus translation, and user-defined types. WaveFormer Pro and TestBencher Pro users should do this tutorial.
- The **Basic Verilog Simulation** tutorial covers the basic simulation features of BugHunter Pro. This tutorial also discusses how to create and manage projects, as well as how to build and simulate your design. BugHunter Pro and TestBencher Pro users should perform this tutorial.
- The **TestBencher Pro: Basic TestBencher Tutorial** covers the basic concepts of using TestBencher Pro to generate bus-functional models for Verilog, VHDL, & OpenVera. It covers signal properties (type, direction, vector size, and bi-directional segments), samples, parameterized state values, end diagram markers, interface diagrams, modifying top-level template files, and generating test benches. TestBencher Pro users should do this tutorial.

# 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 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 WaveFormer Pro, there won't be any surprises during production when you get components at extreme ends of their tolerance range.



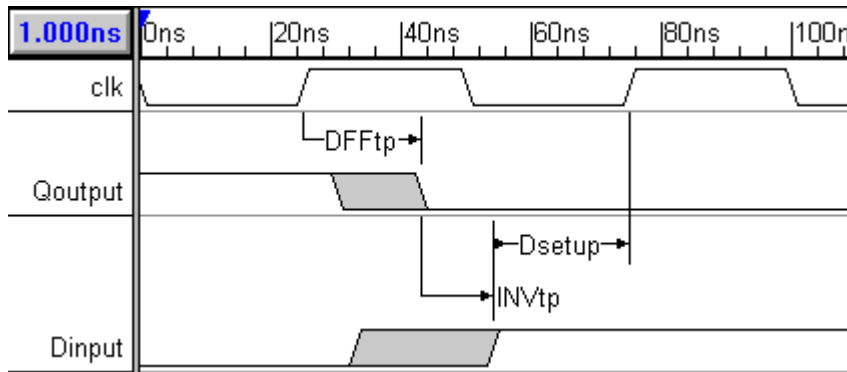


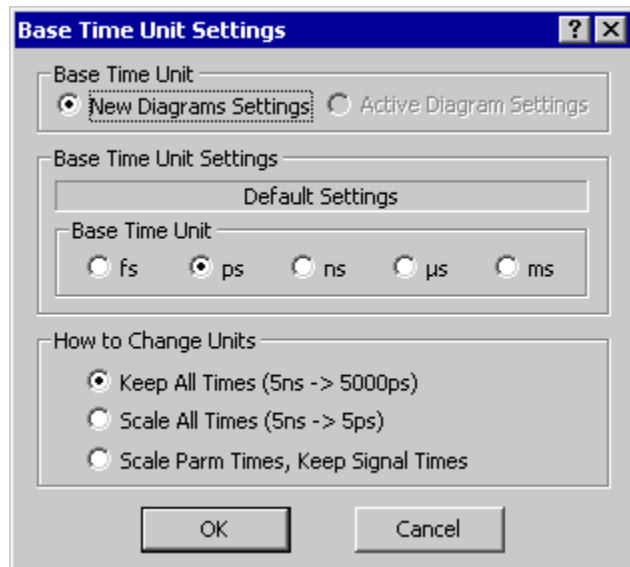
Figure 2: Completed Timing Diagram

## 1) Set the Base Time Unit

At the beginning of each project, you will set the base time unit. The base time unit is the smallest representable amount of time that WaveFormer Pro can display. The base time unit determines the range of times that can be represented in your timing diagram. All time values are internally stored in terms of the base time unit.

In the circuit in Fig. 1, the propagation times for the gates are in units of nanoseconds and the clock has a period of 20ns. Generally it is a good idea to set the base time unit for your project 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). Therefore we will set the base time units to picoseconds. To set the base time unit:

1. Select the **Options > Base Time Unit** menu option. This displays the *Base Time Unit* dialog box with radio buttons that set the base time unit. The other options con-



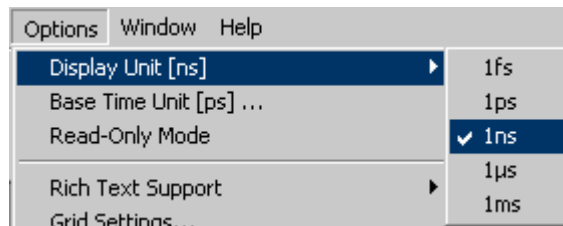
control how any existing parameters or signals are changed when the base time unit is changed and have no effect on an empty timing diagram. See the on-line help if you want to know more about these options.

2. Click on the **ps** radio button to make picoseconds the base time unit (if it is not already selected).
3. Press the **OK** button to close the dialog.

## 2) Set the Display Time Unit

Next you need to set the display time unit. The display time unit sets the units for times which you enter and for times which are displayed. Set the display time unit to the units you most commonly use in the design. To set the display time unit:


1. 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).
2. Click on **ns**, to make nanoseconds the display time unit if it is not already checked.



## 3) Add the Clock

First we will create the clock. The clock is named **clk**, has a period of **50ns** (20MHz), and starts with a low segment.

### To add a clock:

1. Move the mouse cursor over the **Add Clock** button  (DO NOT CLICK), located in the top left hand corner of the Diagram Window.
2. Notice that the status bar at the very bottom of the window reads "Left click to add a clock signal, right click to set clock name prefix" This status bar changes depending on the mouse location and the mode that you are in. Move the mouse around and watch the status bar change. **The status bar is very useful when you want to know what buttons do or when you need to know what to do next in the middle of a program operation.**




3. Click the **Add Clock** button.
4. The *Edit Clock Parameters* dialog box will appear.

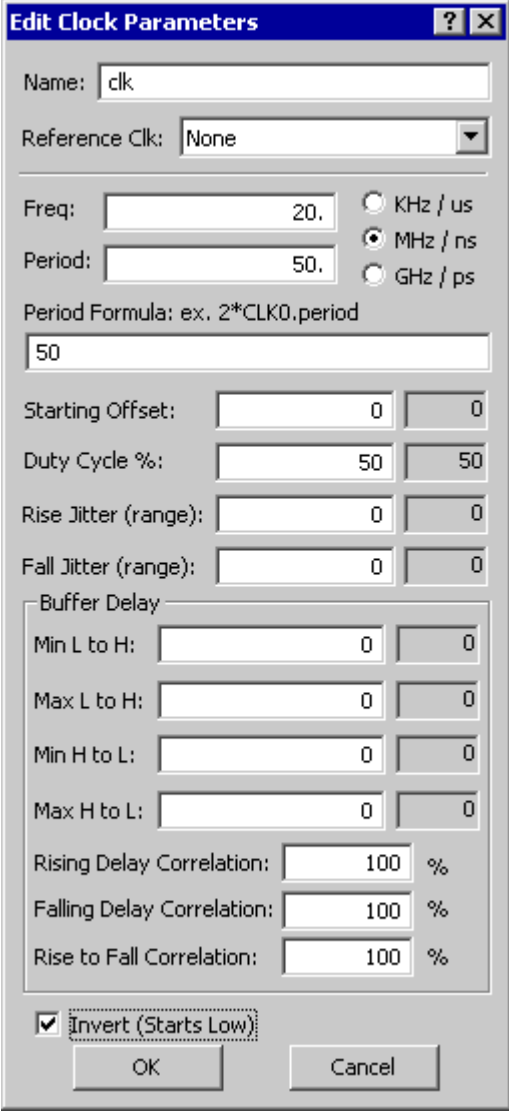
5. Enter the name **clk** in the *Name:* edit box.
6. Enter **50** in the *Period:* box. Make sure the **MHz/ns** radio button is selected. Note that the frequency will change to match the new period value, when you move the selection to another control.
7. Check the **invert** check box. Clocks are normally displayed high at time zero, so "invert" makes the clock start low at time zero.
8. Click the **OK** button to close the dialog box.

**Note:** For more information on clocks, master clocks, clocks with formulas, and clock grids read *Chapter 2: Clocks* in the on-line help. If you made a mistake designing the clock, then double left click on the clock segment to reopen the *Edit Clock Parameters* dialog box. Double left clicking on a clock edge opens up the *Edge Properties* dialog box which displays the edge time. You may also reach the *Edit Clock Parameters* dialog box by double clicking on the clock name and choosing the clock properties button in the *Signal Properties* dialog.

#### 4) Add Signals

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

1. Click twice on the **Add Signal** button  to add two signals. The signals will have default names such as SIG0 and SIG1.
2. **Double left click** on the **SIG0** signal name to open the *Signal Properties* dialog.



**Edit Clock Parameters** [?] [X]

Name:

Reference Clk:

Freq:   KHz / us  
 MHz / ns  
 GHz / ps

Period:

Period Formula: ex. 2\*CLK0.period

Starting Offset:

Duty Cycle %:

Rise Jitter (range):

Fall Jitter (range):

Buffer Delay

Min L to H:

Max L to H:

Min H to L:

Max H to L:

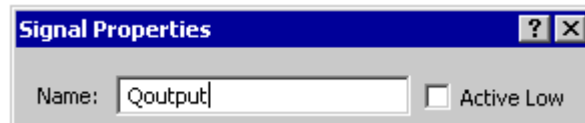
Rising Delay Correlation:  %

Falling Delay Correlation:  %

Rise to Fall Correlation:  %

Invert (Starts Low)

- Enter **Qoutput** into the **Name:** edit box. (DO NOT CLOSE THE DIALOG)



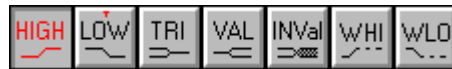
- Click the **Next** button or **ALT-N** to move to the next signal on the list. SIG1 is now displayed in the **Name:** edit box.
- Enter **Dinput** into the **Name:** edit box and press the **OK** button to close the dialog.

If you accidentally close the *Signal Properties* dialog, double click on the signal name to open the dialog again. 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.

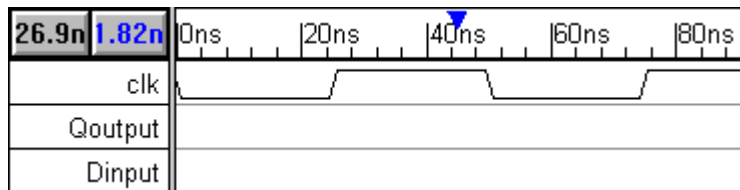
## 5) Drawing Signal Waveforms

Next, we will draw some random waveforms to become familiar with the drawing environment.

- Notice the buttons with the waveforms drawn on them. These are the State Buttons. The active button is colored **red** and indicates the type of signal state that will be drawn next. In this case, the HIGH signal state is active.



- Move** the mouse cursor to inside the *Diagram* window at the same level as the signal name **Qoutput**, and at about **40ns**.



- Click** to draw a waveform segment from 0ns to the cursor. Notice that a HIGH signal was created.

- A different state button is now activated. The State Buttons automatically toggle between the two most recently activated states. The small red T above the signal name denotes the toggle state, for instance. (If you have a 3 button mouse, click the middle mouse button to toggle between the two most recently activated state buttons.)



- Move the cursor to about **80ns** on the same signal and **left click**. Now a LOW segment is drawn from the end of the HIGH signal to the location of the cursor.


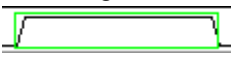

- Left click on the TRI button to activate the tristate State Button and draw another waveform segment.

- 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.

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

## 6) Edit Signal Waveforms

There are four main editing techniques used to modify existing signals (Note: these techniques will not work on clocks). The most commonly used technique is the dragging of signal transitions to adjust their location. The other three techniques all act on signal segments, the waveforms between 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:

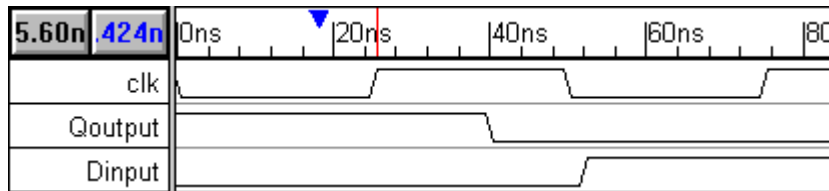
- Move a signal transition:** Left click and hold down the mouse button on a signal transition and drag it to the desired location. A green bar  will appear that follows the mouse cursor. Release the mouse button when the green bar is at the location where you wish to place the transition.
  - Change the state of a segment:** A segment is the waveform between two consecutive signal transitions. Click on the segment to select it (a selected segment has a highlighted box drawn around it ). Then click on the State Button of the new state desired.
- If you try to select a narrow segment and one of the transitions gets selected, widen the segment by clicking the Zoom In  button. This button is located on the right hand corner of the button bar.
- Delete a segment:** Select a segment (see above) and then press the **delete** key on the keyboard.
  - Insert a segment:** Inside a large segment, click and drag to the right or left then release. A new segment will be added in the middle of the original segment. For this operation to work the original segment must be wide enough to be selected.

These techniques will not work on clocks, because clocks have fixed edges and segments. To edit a clock, double-click on a segment of the clock waveform in the *Diagram* window. This causes the *Edit Clock Parameters* dialog box to appear. All clock parameters can be changed in this dialog box. If you cannot double-click on a segment without selecting a transition, zoom in until the segment is large enough.

For more information, read *Chapter 1: Signals and Waveforms* in the Diagram Editor & Universal Features on-line help.

## 7) Adjust Diagram to Match Figure

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



Tile the Parameter and *Diagram* windows so that you will be able to see the interaction between the two windows. The *Report* window is not used in this tutorial, so you can minimize it if your screen is small.

- Select one of the **Window > Tile** menu options in the timing window.

Adjust the zoom level of the drawing so that only 3 whole clock periods are shown on the screen.

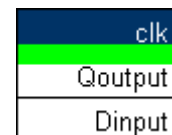
- Click the **Zoom In** or **Zoom Out** buttons, which are located on the right hand corner of the button bar, to show less or more of the waveforms. Zooming can also be performed with the 'Click-and-Drag' method. Simply click in the time bar over the waveforms and drag the cursor to the left or right to zoom out or in, respectively.

## 8) Moving and Reordering Signals

All signals are moved by dragging and dropping the signal's name. 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.

### Moving a Single Signal:

1. Select the signal **clk** by clicking on the name. (A selected signal will be highlighted.)
2. 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.
3. **Drag** the green bar until it is in between **Qoutput** and **Dinput**.
4. **Drop** the green bar by releasing the mouse button. Notice that the timing diagram has redrawn itself.
5. 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:**

1. Select **Dinput**, then select **Qoutput** by left clicking on the signal names in that order.
2. **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.
3. Select **Qoutput** and then select **Dinput**.
4. **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.
5. Return the signals to their original order, (**clk**, **Qoutput**, **Dinput**).

**9) The Right Mouse Button**

In the next sections we will add delays, setups, and comments to the timing diagram. These objects are added using the right mouse button. The function of the right mouse button is determined by the second group of buttons on the button bar marked DELAY, HOLD, SETUP, TEXT, SAMPLE and MARKER.

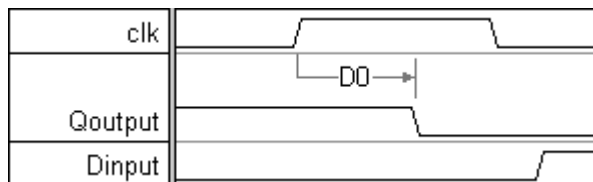


The red or active mode button indicates the current functionality of the right mouse button. To activate a different mode button, click on it.

**10) 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. To add the delay do the following:

1. Activate the **Delay** mode by clicking the **Delay** button.
2. Click on the first rising edge of the clock.
3. Right-click on the first falling edge of the **Qoutput** signal.



This will draw the D flip-flop delay, and creates a blank delay in the *Parameter* window.

name	min	max	margin	comment
D0			na (delay)	

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 edges of the **Qoutput** signal. Now edit the delay's parameters.

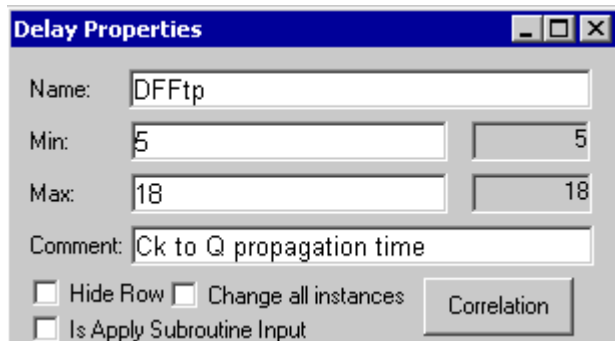
1. **Double-click** on the parameter name **D0** in the Parameter window to open the Parameter Properties dialog. Adjust the position of the *Parameter Properties* dialog so that you can see the parameter in the *Diagram* window and at least part of the parameter in the *Parameter* window. For simplicity, we will refer to the dialog as *Parameter Properties*, even though the name at the top may say *Delay Properties* or *Setup Properties*.
2. 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, or 5ns for this project.

Two things happened when you pressed the TAB key:

1. First, the falling edge of **Qoutput** adjusted itself so that it was 5ns from the clock edge. Measure this for yourself using the time readouts above the signal name window. Left click on the first transition and then move the cursor to the second transition. Notice that the blue readout shows approximately 5ns, depending on the zoom level and the base time unit. The delay can also be made to display the exact distance by choosing the **distance** choice of the **Display label** section of the *Parameter Properties* dialog (make sure to return D0 to "Global Default" when you are done experimenting).
2. Second, 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.

Next, finish editing the rest of the parameter. The parameter is named "DFFtp", has a max time of 18ns, and a comment of "Ck to Q propagation time". Use the *Parameter Properties* dialog that is still open to add the following data:

1. Click in the **Name** edit box and type **DFFtp** into it.
2. Press **TAB** twice so that the **Max** edit box is selected.
3. Type **18**. This means 18 display time units, or in this project 18ns.
4. **TAB** once so that the comment cell is selected.
5. Enter **Ck to Q propagation time** and press the **Enter** key.





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. Use the time measure readouts to verify that the edges of the region are 5ns and 18ns from the clock edge (13ns of uncertainty). Double-click on the edge to see the exact edge values.

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). This tutorial has you open and close it several times so that you learn all the different ways to open the dialog. Also, the tutorial attempts to conserve screen area for laptop users. However, in a normal design you will probably want to keep this dialog open much of the time.

**Tip:** 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).

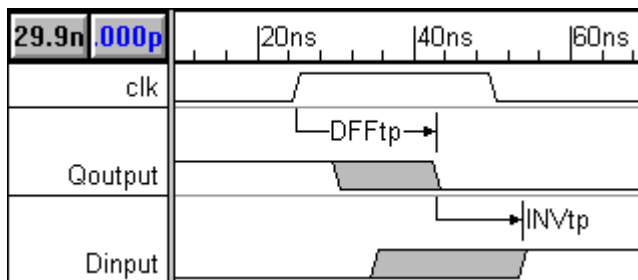
## 11) Add the Inverter Propagation Delay

Add the delay that represents the propagation time of the inverter from its input Q to its output D. To add the delay do the following:

1. Activate the DELAY mode by clicking on the delay button.
2. **Click** on the first falling edge of the **Qoutput** signal (the same edge that ends the "DFFtp" delay).
3. **Right click** on the first rising edge of the **Dinput** signal.

This will draw the inverter delay, and create a blank delay in the Parameter window. Now let's edit the parameters from the inside of the Diagram window instead of going to the Parameter window.

1. Double-click on the new delay in the Diagram Window and enter the following values in the dialog box that appears:
  - Name is **INVtp**.
  - Min time is **3 ns**.
  - Max time is **11 ns**.
  - Comment is **Inverter (Q to D) delay**.
  - Click on the **OK** button to close the dialog.

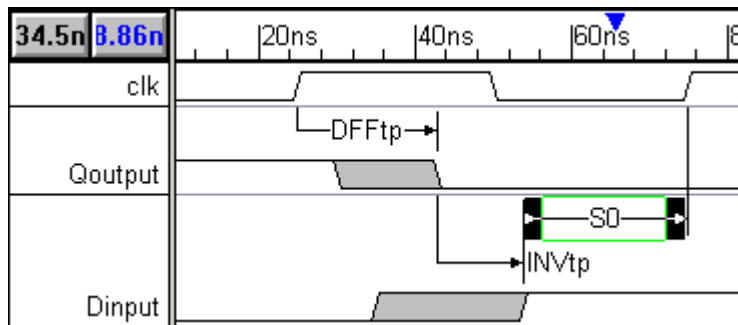


Notice the large uncertainty region for the *Dinput* transition. Click on the first rising edge of **Dinput**, then use the blue delta readout to verify that the uncertainty region lasts for 21ns (13ns from DFFtp + 8ns from INVtp = 21ns). Next, click on the first edge of *clk* and measure to the end 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.

## 12) Add the Setup for the Dinput to Clock

Next add the setup for *Dinput* to clock transition.

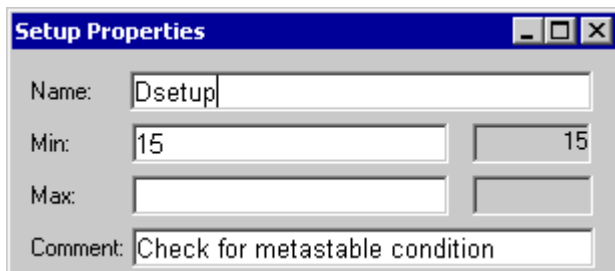
1. Activate the **Setup** mode by clicking the **Setup** button.
2. **Left click** on the first rising edge of the **Dinput** signal (the same edge that ends the "INVtp" delay).
3. **Right click** on the second rising edge of the clock.



This will draw the setup parameter. Notice that the arrows of the setup are pointing to the control signal. This means that you added the setup correctly.

Like delays, setups are also created with empty min/max values. They must have a min value before they start to monitor the data signal's position. Now we will edit the setup.

1. **Double left click** on the setup name in the Diagram window. This will open the *Parameters Properties* dialog box.
2. Enter **Dsetup** into the Name edit box.
3. Enter **15** into the min edit box.
4. Enter **Check for metastable condition** into the comment edit box.
5. Click the **OK** button to close the dialog.

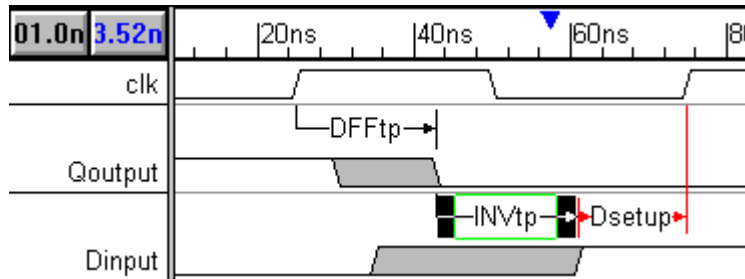


Notice that the margin column in the *Parameter* window says that there is a **6ns** safety region before the setup is violated. Verify this by clicking on the second rising edge of the clock and placing the cursor on top of the maximum edge of the **Dinput** signal. The blue time readout should say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

Next, we will demonstrate what happens when a setup is violated. Increase the inverter's delay so that the maximum delay is 18ns instead of 11ns:

1. Double-click on **INVtp** in the *Diagram* window.
2. Type **18** into the **max** edit box and **TAB** to another control.

Notice that the setup has turned red in both the *Diagram* and *Parameter* windows. Change the inverter delay back to **11ns** and click **OK** to close the dialog.




### 13) Add a Free Parameter

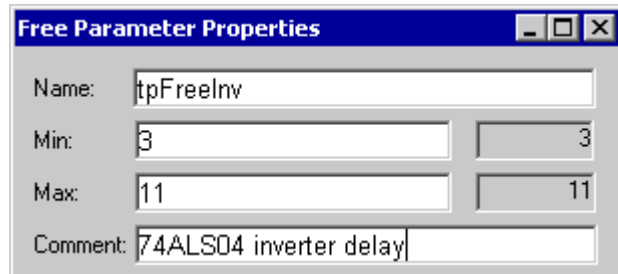
So far we have always directly edited a parameter's values. This is inefficient and error prone if the circuit is large. It would be better to define one variable that held the value and make everything that needed that value reference this variable. Then if the value needs to be changed, you only have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

#### To add a free parameter:

1. Click the **Add Free Parameter** button  in the *Parameter* window. A blank free parameter is added to the *Parameter* window.
2. Double click on the free parameter to open the *Parameter Properties* dialog box and enter (**tpFreeInv**, **3ns**, **11ns**, and **74ALS04 inverter delay**) for the (name, min, max, and comment cells) of the free parameter.

- Use the **Previous** and **Next** buttons in the *Parameter Properties* dialog to locate the **INVtp** parameter.
- Type **tpFreeInv** into the min and max cells of **INVtp**. Changes to the timing values of the free parameter will now affect **INVtp**.



Note: 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*.

## 14) Using Formulas and Constants

Parameters can contain mathematical formulas as well as numeric time values. Legal operations are: 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 the following equation into **INVtp**'s **min** edit box:

**3 \* tpFreeInv**

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.

## 15) Summary

Congratulations! You have completed the **Basic Drawing and Timing Analysis** tutorial. In this tutorial we have covered three main topics. The first is how to start a project. Next we covered signals, which includes clocks, signals, and drawing the waveforms of the signals. And finally we covered parameters.

- Starting a project

- **Always set the Base Time Unit one unit below the Display Time Unit** to avoid rounding errors. Default values are: Base Units= ps and Display Units = ns.

2) Drawing a timing diagram with Signals and Clocks

- Use the Add Clock and Add Signal buttons to add clocks and signals to the diagram. Double left click on the signal name to edit the signal name.
- Left click to draw a waveform with the state of the selected State Button.

3) Editing waveforms

- Drag and drop signal transitions.
- To change the graphical state of a segment, select it then press a State Button to indicate the new graphical state.
- To Delete a segment, select it then press the delete key on the keyboard.
- To Insert a segment, left click and drag to the right.

3) Timing Analysis with Parameters

- Add delay, setup, and hold parameters by (1) activating the mode button of that name, (2) left clicking on the first signal transition, and (3) right clicking on the second signal transition.
- Edit parameters by double left clicking on the parameter in either the Diagram or the Parameter window.
- Free parameters are variables that other parameters reference. Use the **Add Free Parameter** button in Parameter Window to add a free parameter.
- To use a free parameter, type the name of the free parameter in the min or max column. Free parameters can also use the dot min/max property to specify a specific value. For example, Inverter.min retrieves just the minimum value of the parameter called Inverter.

# Interactive HDL Simulation Tutorial

This tutorial introduces the Interactive HDL Simulation. WaveFormer, VeriLogger and TestBencher Pro have a built-in Interactive HDL Simulator that greatly reduces the amount of time needed to draw and update a timing diagram. Using Boolean and Registered logic equations written in VHDL, Verilog, or SynaptiCAD's syntax you can describe signals in terms of other signals in the diagram. You will no longer have to figure the output of a combinational circuit or calculate the critical path of a synchronous circuit by hand. SynaptiCAD's interactive simulator will generate the HDL code using information entered into the Logic Wizard dialog and then simulate the result. Since the simulator is interactive, changes to input waveforms will automatically re-simulate so that your timing diagrams always reflect accurate design data.

This feature is included in the VeriLogger and TestBencher products even though they have a built in Verilog simulator because it makes generating test benches and timing diagrams so fast that we couldn't hold it back. In WaveFormer, it is the backbone of the timing analysis and design features. The Interactive simulator supports multi-bit equations and true min-max timing. This tutorial contains some examples of equations that are supported.

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.

If you are evaluating Timing Diagrammer Pro and you would like to learn about the simulation features, close the program and restart the evaluation version in WaveFormer Pro mode.

## 1) Interactive HDL Simulation

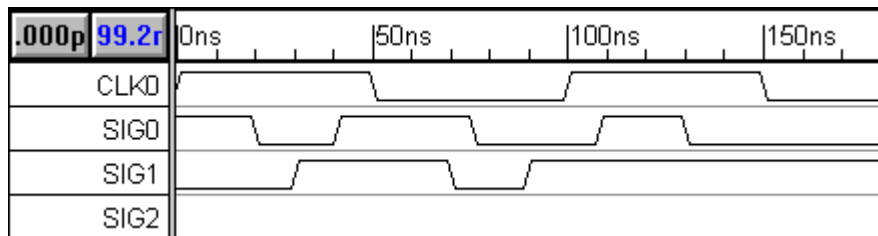



Figure 1.1: Timing diagram used for the Interactive HDL Simulation Tutorial

Create a timing diagram for experimenting with the Interactive HDL Simulator:

1. Add a clock named **CLK0** and accept the default properties of **100ns** period.
2. Add two signals, **SIG0** and **SIG1**, to the timing diagram.
3. Draw the waveforms for signals *SIG0* and *SIG1* so that they resemble the signals in Figure 1.1. These will be the input signals for our simulation.

- In the *Parameter* window, click the **Add Free Parameter** button



to add a free parameter **F0** to the *Parameter* window.

- Double-click on the free parameter **F0** (in the *Parameter* window) to open the *Parameter Properties* dialog.
- Type **10** in the *Min* edit box and **15** in the *Max* edit box.
- Click the **OK** button to close the dialog.
- Add signal **SIG2** to the timing diagram. You do not have draw the waveforms now, we will be simulating this signal.


## 2) Generate Waveforms From Boolean Equations

We will begin by simulating a Boolean Equation. WaveFormer Pro 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**.

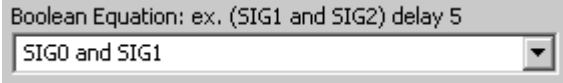
The **delay** operator takes a signal on the left, and a time or parameter name on the right, and returns a signal. If a parameter name is used on the right hand side of the delay operator, then the equation will simulate true min/max timing. This true min/max timing is the main advantage that SynaptiCAD's format has over the VHDL or Verilog format.

### Simulate a Boolean equation:

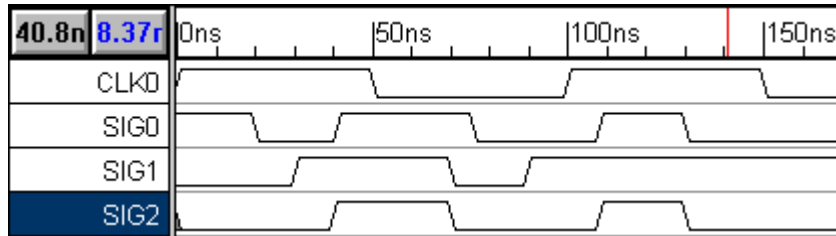
- Double click the **SIG2** signal name to open the *Signal Properties* dialog. Arrange the *Signal Properties* dialog so that you can see the dialog and the 3 signals at the same time. This dialog is modeless, so leave it open for this entire section. All controls and buttons used in this section are contained in the *Signal Properties* dialog.

- Make sure that the **Boolean Equation** radio button  **Boolean Equation** is selected.

- Type the following equation into the Boolean equation edit box (signal names are case sensitive):  
**SIG0 and SIG1**

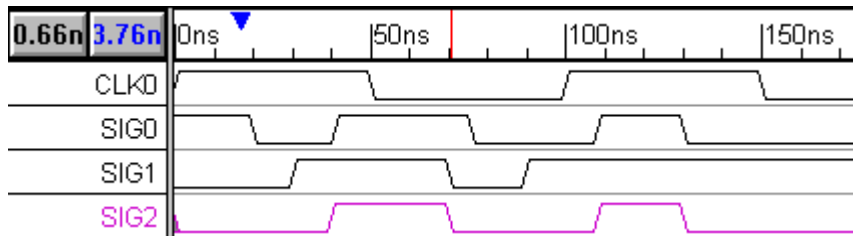


- Click the **Simulate Once** button and watch the signal draw itself. Notice that SIG2 is the result of the Boolean Equation "SIG0 and SIG1". By default, the **Simulate** radio button is not checked, so if you moved an edge on SIG0, SIG2 is not automatically re-simulated.



### Continuously Simulate the Boolean Equation:

- Enable the **Simulate** radio 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 resimulations. 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.



- 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).

## 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.

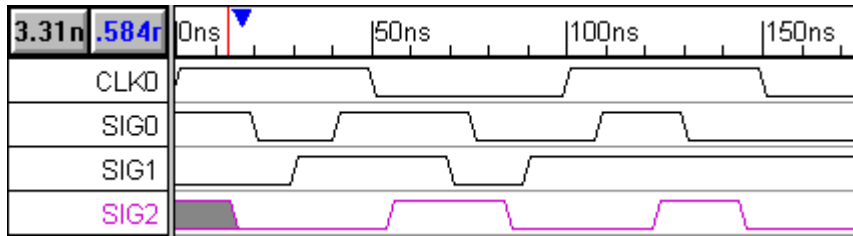
### Simulate a simple delay:

- Enter one of the following Verilog, VHDL, or SynaptiCAD equations into the **Boolean Equation** edit box of SIG2:

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

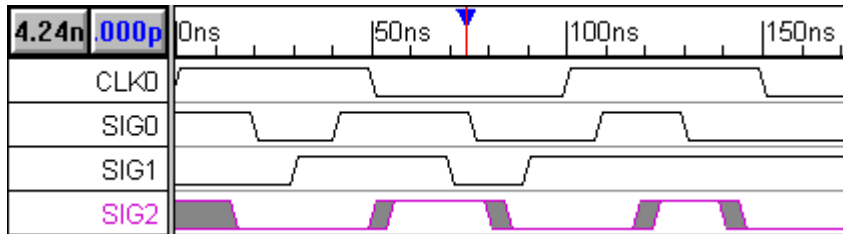


- Click the **Apply** button and verify that SIG2 is correctly drawn.



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

- Modify the **Boolean Equation** of *SIG2* to take into account the min and max propagation delay of the AND gate. Enter **(SIG0 and SIG1) delay F0** into the *Boolean Equation* edit box. This Boolean Equation references the Free Parameter *F0* that you added at the beginning of the tutorial.



- Click the **Apply** button to cause a simulation. Notice the gray uncertainty regions on SIG2. This true min/max timing is the main advantage that SynaptiCAD's format has over the VHDL or Verilog format.

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

- Click the **Verilog** or **VHDL** radio button  Verilog  VHDL to view the HDL code that simulates the Boolean Equation. Native HDL code can be added here to perform a special function (ability to use Native HDL code is only supported for Verilog at this time, not for VHDL). Do not modify the code now. The code should resemble the following example:

```

wire # F0_min SIG2_wf0 = (SIG0 & SIG1 );
wire # F0_max SIG2_wf1 = (SIG0 & SIG1 );
assign SIG2 = (SIG2_wf0 === SIG2_wf1) ? SIG2_wf0 : 'bx;

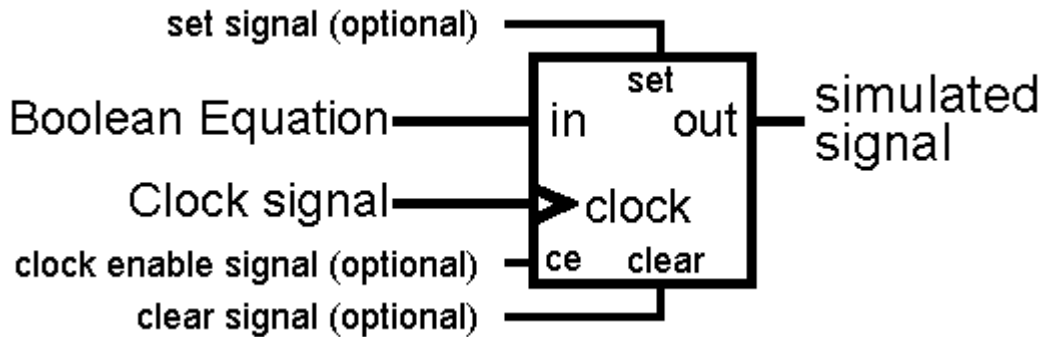
```

- Click the **Boolean Equation** radio button to display the Logic Wizard section (or Boolean Equation Section) of the *Signal Properties* dialog.
- Leave the *Signal Properties* dialog open. We will be using it in the next section.

**Note:** 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.

#### 4) Register and Latch Signals

The Interactive Simulator can register or latch the result of a Boolean equation. Figure 1.2 represents the circuit that is modeled.



The *Signal Properties* dialog should still be open and displaying the **SIG2** information from the last section. Let's experiment with the register and latch functions:

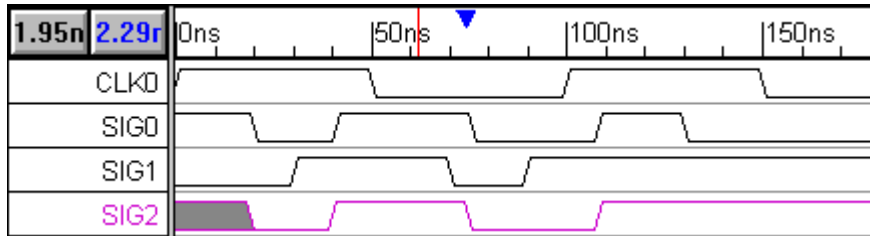
1. Enter the equation **SIG1** into the **Boolean Equation** edit box of SIG2.
 

Boolean Equation: ex. (SIG1 and SIG2) delay 5
2. Click the **Simulate Once** button to simulate the equation. SIG2 should look like an exact copy of SIG1. When we register SIG2 you can visually compare it to SIG1 to see the effects of the register.
3. Next use the **Clock** drop down list box and choose **SIG0** as the clocking signal. The clocking signal can be any clock or signal in the timing diagram (the default value "Unclocked" means no flip-flop is present).
 

Clock:
4. Next use the **Edge/Level** drop down list box (on the right side of the dialog) and choose **both** as the triggering edge.
 

Edge/Level:

- Click the **Simulate Once** button to simulate the circuit. Notice that SIG2 only transitions when SIG0 has a positive or negative edge transition (move some edges on SIG0 and SIG1 to verify this).



Whether a Register or a Latch is simulated depends on the type of triggering in the **Edge/Level** list box. For a Register circuit choose **neg** for negative edge triggering, **pos** for positive edge triggering, and **both** for edge triggering. For a Latch circuit, choose either **low** or **high** level latching.

- Choose different **Edge/Level** values and press the **Simulate Once** button to verify the operation of the register and latch functions.

## 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 will demonstrate how to design a **divide by 2 circuit** using a negative edge triggered register with an asynchronous active-low set line.

### To specify the initial value:

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

7. Click the **Advanced Register** button in the *Signal Properties* to open the *Advanced Register and Latch Controls* dialog. All the register and latch individual propagation times, setup/hold constraints, clock enable, and set/clear options are set here. Note that Global defaults are set using the **Options > Simulation Preferences** menu.



8. Make sure the **Active Low** and the **Asynchronous** check boxes in the *Set and Clear* section are checked. Click **OK** to close the dialog.
9. Choose **SIG0** in the **Set** drop down list box of the *Signal Properties* dialog.
10. Click the **Simulate Once** button. This button is located at the top left corner of the *Signal Properties* dialog, under the signal name. Notice that **SIG3** now has a simulated waveform. Experiment with **SIG0** to see how the active low set line affects the operation of the flip-flop. You may want to redraw **SIG0** so that it goes **low** early in the timing diagram, and then stays **high** for four or five clock cycles.

The **Clock to Out**, **Setup**, and **Hold** edit boxes in the *Advanced Register* dialog accept time values for various timing constraints on the register and latch circuit. For more information on Register and Latch timing, see the on-line help *Chapter 12: Interactive HDL Simulation*.

## 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.

First setup the diagram to experiment with multi-bit equations:

1. **Delete** the **SIG3** signal by selecting it and pressing the **Delete** key.
2. Create a copy of **SIG2**. Click on the **SIG2** name in the Label window to select it. Select the **Edit > Copy Signals** menu option to copy the signal, then the **Edit > Paste Signals** option to paste the signal. There are now two signals named **SIG2** in your diagram. Rename the bottom **SIG2** to **SIGX**. **SIGX** should have the exact same waveform as **SIG2**.

Next, change the output of **SIGX** to a multi-bit signal:

1. Double-click on the **SIGX** signal name to edit it in the *Signal Properties* dialog.
2. Make sure the **Simulate** radio button is selected.
3. Type **3** in the **Bus MSB** edit box. This will make **SIGX** a 4-bit signal.
4. Click the **Apply** button. **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 **SIG1** to a multi-bit signal:

1. Double-click on the **SIG1** signal name to edit it the *Signal Properties* dialog.
2. Change the name of **SIG1** to **SIG1[3:0]**. Changing the name using the bracket notation has the same effect as changing the values in the **MSB** and **LSB** edit boxes.
3. Click the **Apply** button to accept the change. Now all four bits of **SIGX** should be toggling 1111 and 0000. If the radix is in **Hex**, the signal will toggle between 0 and F. 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 4-bit values, like 1010 or 0011, into the **Virtual** edit box and watch how it affects the output of **SIGX**.

Next, set up the diagram for the next section:

1. Delete signals **SIG1**, **SIG2**, and **SIGX** by selecting the names and pressing the **Delete** key.
2. Add a signal called **SIG1**. Do not draw the waveform, we will simulate it in the next section.
3. The timing diagram should consist of one clock (**CLK0**), and two signals (**SIG1** and **SIG0**).

## 7) Experiment with Behavioral HDL Code

In addition to the simulation of Boolean and registered logic circuits, SynaptiCAD products can simulate behavioral HDL code. To enter behavioral code for a signal, click on either the **Verilog** or the **VHDL** button in the *Signal Properties* dialog and type code directly into the edit box.

WaveFormer Pro, VeriLogger Pro and TestBencher Pro also provide a template feature that allows you auto generate the register and latch models used by the Logic Wizard. 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 negative edge of the clock. To model this circuit:

1. Double-click on the **SIG1** signal name to edit it in the *Signal Properties* dialog.
2. Select the **Simulate** radio button.

3. Choose **CLK0** from the **Clock** drop-down list box.
4. Choose **neg** from the **Edge/Level** drop-down list box.
5. Type **3** into the **Bus MSB** edit box.
6. Click the **Verilog** code button to view the resulting template code:

```
wire [3:0] SIG1_wf1 = PLACEHOLDER ;
wire [3:0] SIG1_wf0;

registerN_Asyn #(4,1,1) registerN_Asyn_SIG1(SIG1_wf0,CLK0,
    SIG1_wf1,1'b0,1'b1,1'b1, $realtobits(0.0),$realtobits(0.0),$re-
    altobits(0.0),$realtobits(0.0) );

assign SIG1 = SIG1_wf0;
```

**Note:** the internal wire name SIG1\_wf\*\*\* will vary depending on how many signals you have simulated.

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. The **registerN\_Asyn** line instantiates (defines an instance of) a 4 bit negative-edge-triggered register of the type used by the logic wizard. This register takes PLACEHOLDER as an input and outputs a synchronized version on **SIG1**.

7. We will use the PLACEHOLDER variable to store the edge count. Edit the behavioral code so that it looks like this (add the bold lines):

```
reg [3:0] PLACEHOLDER;
initial PLACEHOLDER = 0;
always @(SIG0)
    PLACEHOLDER = PLACEHOLDER + 1;
wire [3:0] SIG1_wf1 = PLACEHOLDER;
wire [3:0] SIG1_wf0;

registerN_Asyn #(4,1,1) registerN_Asyn_SIG1(SIG1_wf0,CLK0,
    SIG1_wf1,1'b0,1'b1,1'b1, $realtobits(0.0),$realtobits(0.0),$re-
    altobits(0.0),$realtobits(0.0) );

assign SIG1 = SIG1_wf0;
```

8. Click the **Simulate** radio button. Verify that SIG1 is counting the edges of SIG0. The new edge count is presented on each negative edge of CLK0.

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 **SIG0** changes state, the always block will execute, incrementing PLACEHOLDER. The last two lines consist of the automatically generated template code that instantiates the synchronizing register.

**Tip:** More information on the HDL simulator can be found in *Chapter 12: Interactive HDL Simulation* in the manual and the on-line help. Also the *Advanced Modeling and Interactive Simulation* tutorial demonstrates how to model a complex circuit using external models, behavioral HDL code, and incremental simulation techniques. The HDL Simulation features are different from the VHDL and Verilog testbench generation features which are covered in the Advanced HDL Simulation, VeriLogger Pro, and TestBencher Pro tutorials.

## 8) Summary

Congratulations! You have completed the Interactive HDL Simulation tutorial. In this tutorial we have introduced the use of Boolean Equations and the beneficial features of the Boolean Equation edit box. We examined the generation of waveforms using equations, simulation of delays, and viewing HDL code generated from an equation. We also covered Register and Latched signals, Multi-Bit signals, and editing behavioral code. For more information please refer to the manual or the on-line help.

# Waveform Generation And Bus Tutorial

In this tutorial you will learn techniques to quickly generate signals from temporal equations, add equations to existing signals, and create Virtual, Group and Simulated buses. This tutorial assumes that you are able to draw signals and are familiar with the features explained in the previous tutorials. 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.

## 1) Generate Waveforms from Temporal Equations

Temporal equations provides a quick way to generate signals that have a known pattern that is more complicated than a periodic clock. Temporal equations are entered in the *Signal Properties* dialog using the edit box to the right of the **Wfm Eqn** button. The edit box contains the default equation: **8ns=Z (5=1 5=0)\*5 9=H 9=L 5=V 5=X**. The default equation draws a waveform that uses all of the available waveform states. If you start by editing the default equation you do not have to memorize the syntax of these equations.

The syntax consists of time-value pairs separated by spaces. The values are 0, 1, Z, V, H, L, and X which represent the graphical states of the waveforms. For example, the **8ns=Z** part of the default equation draws an **8 ns tristate** segment.

To repeat a sequence of states, enclose a list of time-value pairs in parentheses and use the multiply symbol \* followed by the number of times the list is to be repeated. For example **(5=1 5=0)\*5**, draws five copies of a 5ns strong high segment followed by a 5ns strong low segment.

To experiment with temporal equations:

1. Click the **Add Signal** button to add a signal, and change its name to **TIMEeqn** using the Signal Properties dialog. (If the dialog is closed, double click on the signal name to open it.)
2. In the waveform equation edit box, (to the right of the **Wfm Eqn** button) enter the equation:

**10=Z (6=V 6=X) \*3 10=0**

3. Click on **the Wfm Eqn** button to apply the equation.

The **10=Z** in the equation means that the signal will be initially tri-stated for 10ns. Next the **(6=V 6=X)** will cause the signal to be valid for 6ns then invalid for another 6ns. The **\*3** will cause the sequence inside the parentheses to be repeated three times. Finally, the **10=0** will cause the signal to be a strong low for 10ns.

The text in the diagram above is made with a combination of text objects and setup parameters with custom labels. It is used to illustrate the different components of a temporal equation. This is just a quick demonstration of the documentation abilities of the program. For more information on documentation read *Chapter 8: Formatting Timing Diagrams*. You do not have to add the text for this tutorial.



Waveform equations are stored in the waveform equation drop down box, located next to the **Wfm Eqn** button. The equations can be used to create new signals or concatenated to the end of an existing signal.

#### Adding equations to existing signals:

1. In the *Signal Properties* dialog, click on the down arrow of the equation drop-down box to display the previous equations.
2. Select the default equation  $8ns=Z(5=1\ 5=0)*5\ 9=H\ 9=L\ 5=V\ 5=X$ . You may have to scroll down to find it.
3. Click the **Wfm Eqn** button. Notice that the waveform was added to the end of the TIMEeqn signal.



Temporal Equations and a related feature called State Label Equations provide a quick method of generating and then labeling signals that represent Counter and Shifter circuits. The on-line *help Chapter 11: Waveform Equation Generation* has more information on these features.

## 2) Bus Overview

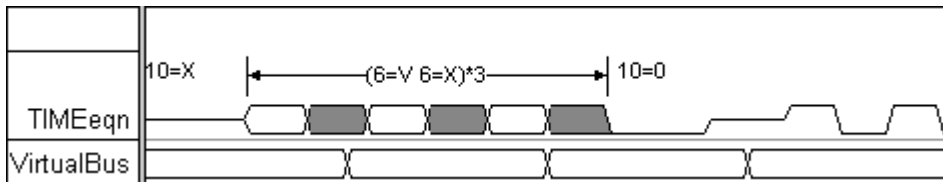
There are three kinds of buses supported by the timing diagram editor: **group buses**, **virtual buses**, and **simulated buses**.



- 2.1 Virtual Buses** are normal signals that use extended state information to represent bus values. Virtual buses are added using the **Add Signal** button. The state information is added using the HEX state button and the Virtual edit box. A virtual bus does not have member signals.
- 2.2 Group Buses** are composite signals whose transitions and state values are determined by their member signals. Instead of individually editing related signals (like the address lines of a part), a group bus can compress all the signals' data into one compact signal. The individual member signals can be uncoupled, or displayed along with the bus. Buses are added using the **Add Bus** button.
- 2.3 Simulated Buses** are similar to group buses. The primary difference is that a simulated bus is purely simulated - the member signals cannot be edited manually. When any kind of simulation is performed, the simulated bus will be re-simulated and any changes will take place at that time.

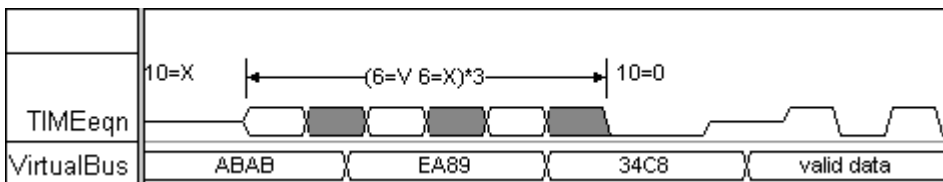
## 2.1) Creating Virtual Buses

Virtual Buses are the recommended way to display and work with bus information. Virtual Buses are also supported by the VHDL and Verilog stimulus and test bench generation features. If timing parameters are attached to a bus then virtual buses will increase computational performance for timing diagrams that use large buses (32 bits or more). To create a virtual bus:

1. Click the **Add Signal** button to add a new signal and name it **VirtualBus**.
2. Click the **VAL** state button twice, so that it stays active (state buttons will not toggle). The Valid button should be red and have a red **T** at the top of the button .



4. Double click on the first segment in the signal to open the *Edit Bus State* dialog box.
5. Enter data into the *Virtual* field and use either the **Next** and **Previous** buttons or the key combinations **Alt-N** and **Alt-P** to move between the different segments. Any string of characters and numbers can be displayed in the bus. We used the following data: **ABAB**, **E389**, **34C8**, **valid data**.
6. Click **OK** to close the *Edit Bus State* dialog when all the segments have been edited.
7. Click the **ZOOM OUT** button  a couple of times to demonstrate how the extended state data automatically hides itself when its segment becomes too small to display the text.
8. Click the **ZOOM IN** button  the same number of times to return the diagram to its original zoom level.



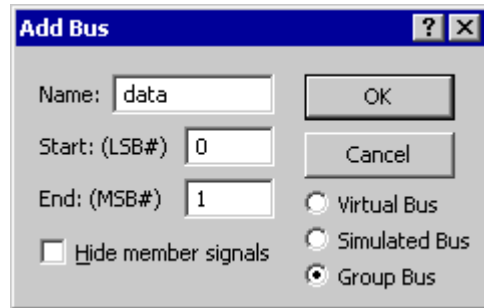
When exporting to VHDL or Verilog, the Virtual State information contained in a valid supersedes the graphical state of a segment. This allows you to export the state values of signals with types that have no graphical representation (integers for example).

## 2.2) Creating Group Buses

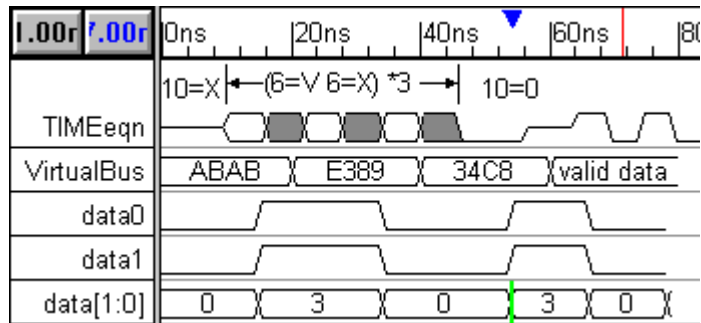
Use group buses only when you need to get access to an individual bus signal at some point in your design or if you need to compress several signals that already exist. Group buses are useful for analysis of data that is imported from simulators or test equipment. Before a group bus can be created, its 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:

1. Make sure that no signal names are selected (clear selected signals by clicking in the *Diagram* window).
2. Click on the **Add Bus** button. This will open the Add Bus dialog box.
3. Type **data** into the **Name** box. The member signals will be named the same name as the bus, plus their signal number.
4. Enter **0** into the **Start(LSB#)**: edit box. This is the least significant bit of the bus.
5. Enter **1** into the **End(MSB#)**: edit box. This is the most significant bit of the bus.
6. Make sure the **Group Bus** radio button is selected. The radio buttons provide an easy method for creating group or virtual buses.



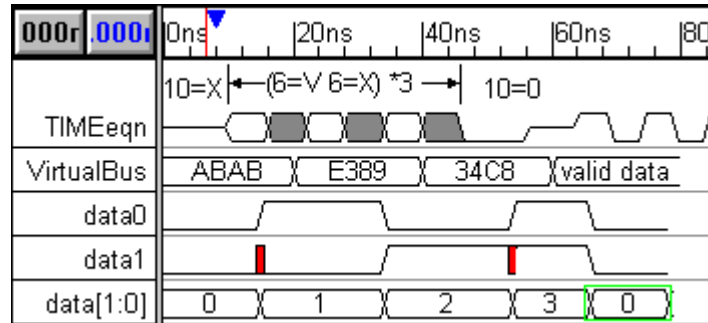
7. Verify that the **Hide member signals** check box is **NOT** checked. We want to be able to see the member signals in this demonstration.



8. Click the **OK** button to 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.
9. Next **draw** 5 high and low segments on *data* (the bus signal) and notice that the member signals are automatically drawn.
10. **Double click** on the first segment of *data* to open the *Edit Bus State* dialog.

11. Type the value **0** into the **Hex** edit box.

12. Use the **Alt-N** key combination to move to the next segment. Enter the values: **1,2,3,0** into the remaining four segments. Notice that the member signals have redrawn properly (except the red transition markers which we will fix later). The red transition markers preserve all the edge information of the member signals during a bus editing session. Click **OK** to close the dialog.

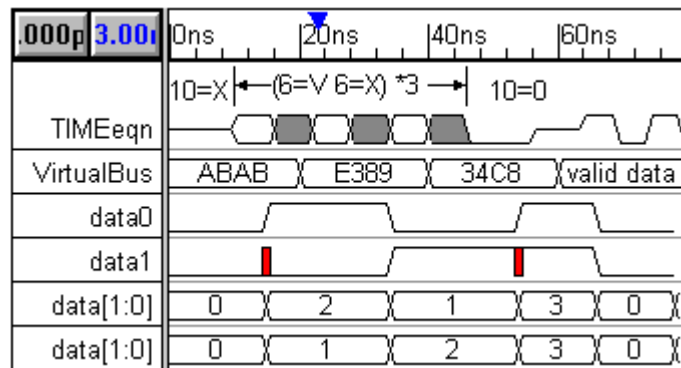


13. Select the **Edit > Clear Red Events** menu option to remove the edge place holders on the member signals.

### Creating a group bus from existing signals

Select the signal names to be grouped, in order from LSB to MSB, then click the **Add Bus** button. In the next example we will create a second group bus whose member signals are reversed from the **data** bus.

1. Select **data1** by clicking on the name. This will be the LSB of the new bus.
2. Select **data0** by clicking on the name. This will be the MSB of the new bus.
3. Click on 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.



4. 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**.

Group buses have many features that are covered in *Chapter 3: Group, Simulated, and Virtual Buses* of the manual and the on-line help. Before you use group buses extensively, you should read this chapter and experiment with the **align**, **bind**, and **expand** features.

## 2.3) Creating Simulated Buses

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, TestBench will generate a Boolean equation for the timing transaction so that the simulated bus can include input signals as member signals.

### To add a Simulated Bus:

1. Make sure that no signals are selected.
2. Click the **Add Bus** button to open the *Add Bus* dialog.
3. Select the **Simulated Bus** radio button and name the bus **SimBus** with an **LSB** of **0** and an **MSB** of **2**.
4. Click **OK** to close the dialog and add SimBus and 3 member signals to the diagram.
5. Double click on the **SimBus** name to open the *Signal Properties* dialog. Notice that the Boolean equation is a concatenation of the member signal signals. Draw the member signal waveforms and watch the Simulated Bus change. If the diagram did not simulate, choose "Options > Diagram Simulation Preferences" menu and check the "Continuously Simulate" check box.

## 3) Summary

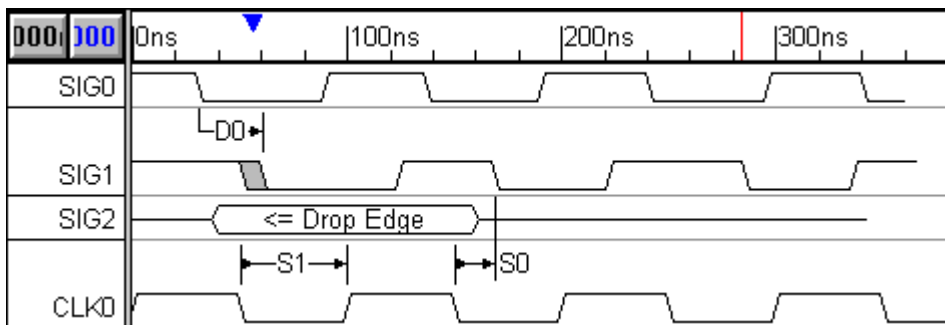
Congratulations! You have completed the Waveform Generation and Bus tutorial. In this tutorial we covered the generation of Waveforms from Temporal equations and adding to existing signals. We also covered virtual buses, group buses, and creating a group bus from existing signals. We examined the differences between Virtual, Group, and Simulated buses, and the recommended use for each. For more information, please refer to the manual or the on-line help.

# Display and Documentation Tutorial

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**.



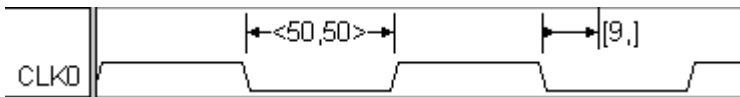
## 1) Controlling the Parameter Display String

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.

- 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.
- 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.
- Select the **Distance** from the **Display Label** drop-down list box. The label now shows the minimum and maximum distances between the transitions.
- 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

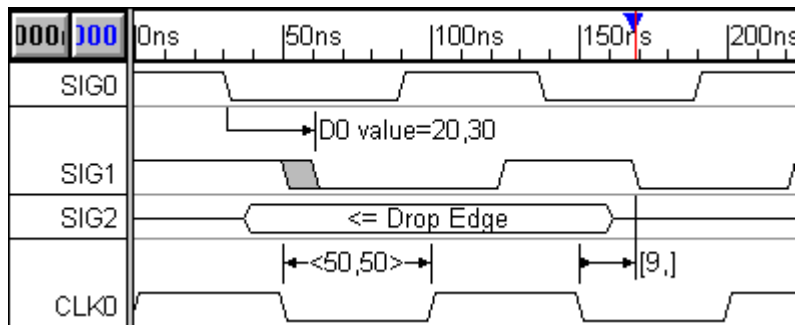
A parameter label can be made to show more than one attribute or to show a custom string of characters and attributes using the **Custom** string in the *Parameter Properties* dialog. In a custom string, certain character sequences are interpreted as attribute control codes, and when such a sequence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

- Double click on **D0** delay parameter to open the *Parameter Properties* dialog.
- 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 parameter's label.
- 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
```
- 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**

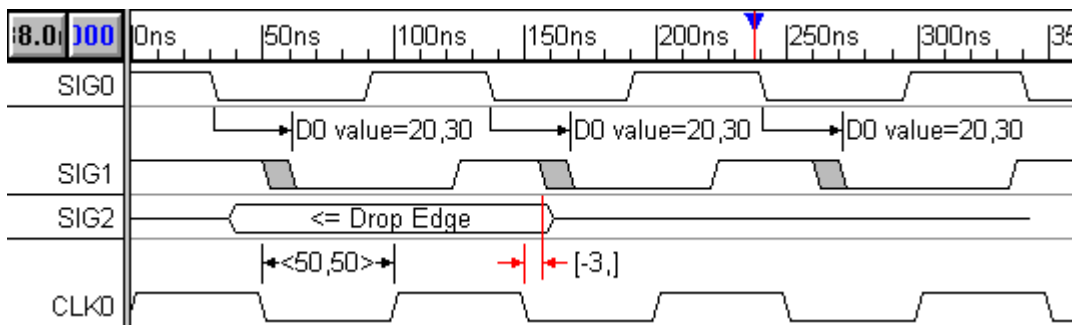
5. Click the **Apply** button. The and D0's label will show:



## 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.

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.
3. 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.
4. Close the Parameter Properties dialog.

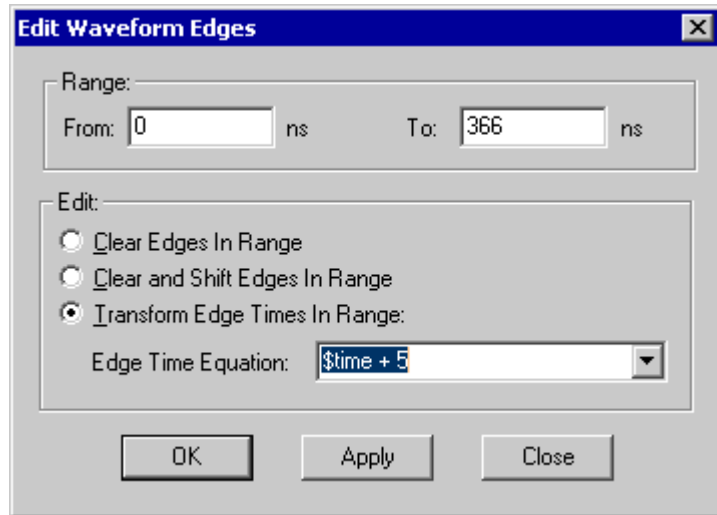




### 3) Editing Waveform Edges From an Equation

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.

1. Select the **SIG0** and **SIG1** names by clicking on the signal names.
2. 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.

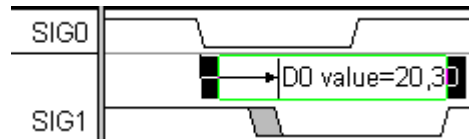


3. Type **\$time + 5** into the **Edge Time Equation** edit box. The \$time variable represents the time of each edge.
4. Press the **OK** button to apply the equation and close the dialog. Notice the edges have shifted over and the **S1** setup is satisfied.

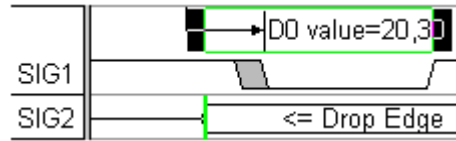
### 4) Drag and Drop Parameter End Points

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:

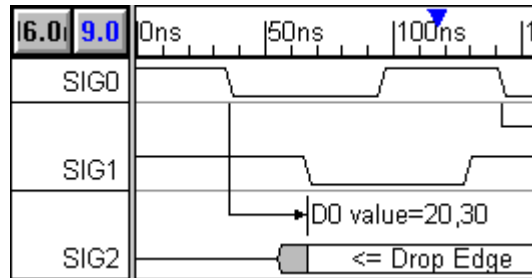
1. In the *Diagram* window, select the first delay parameter *D0* to select it by clicking on it. A selected parameter is surrounded by a rectangle with a solid handle box on either end.
2. Place the mouse over the solid handle box on the right side of the selection rectangle.



- 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.



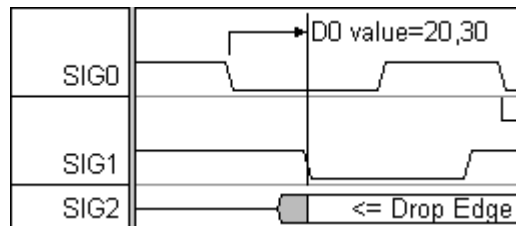
- Release the mouse button. Now *D0* ends on this transition.



## 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.

- 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.
- 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:

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

## 6) Clock Jitter and Display

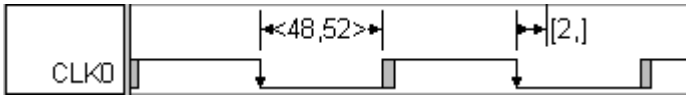
Clocks have many display and timing analysis settings that are covered in *Chapter 2: Clocks*. In this section we will add edge jitter and see the effect on the distance measurement. We will also add ar-

rows 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.


## 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. TestBencher 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.

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

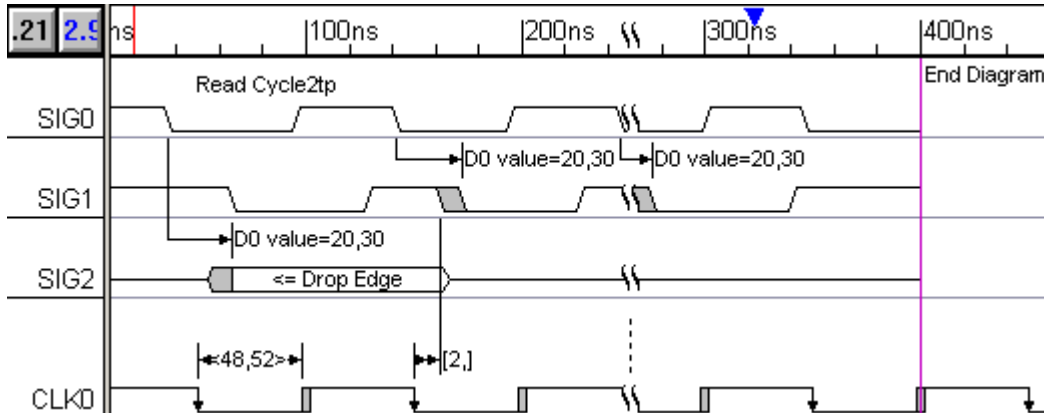
Next add a documentation marker the diagram and experiment with the display and time compression.

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 where 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.

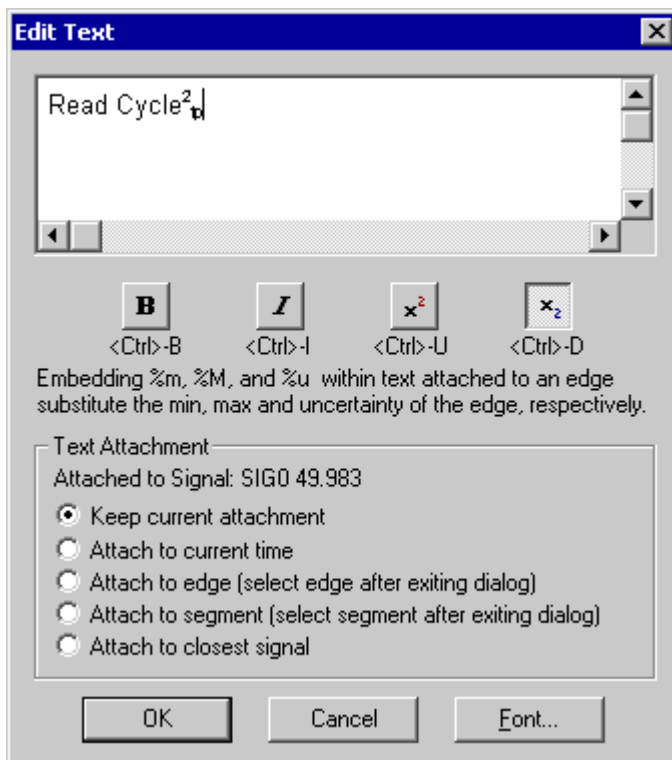
6. Drag and drop the end diagram marker and notice how the waveforms draw themselves.



## 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 Cycle<sup>2tp</sup>**, 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.

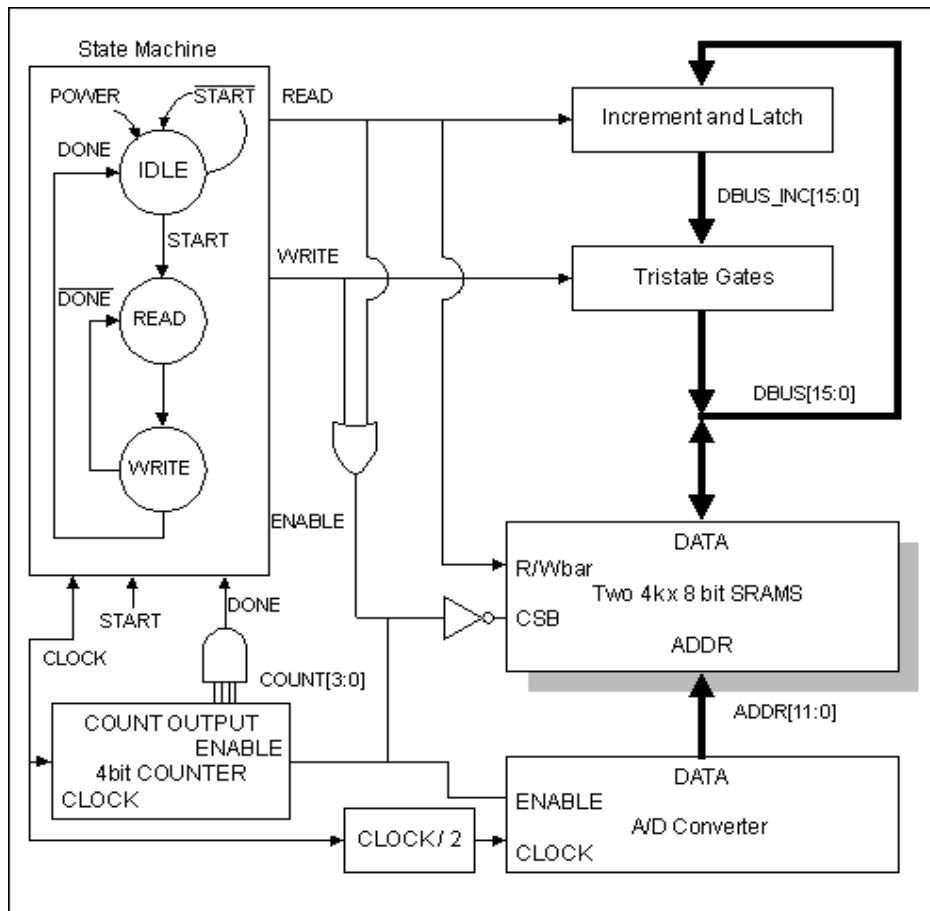
3. Double click on the text block to open the *Edit Text* dialog. From this dialog you can edit the text, add multi-line text blocks, and set the bold, italics, superscript, and subscript settings.
4. Select the **2** in the text box and press the super script button. Select **tp** and press the subscript 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 multilane text block.

## 9) Summary

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.

# 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 ( $2^{12}$  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 ( $2^{16} = 64K > 4K$ ).

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

## 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.

## 2) Generate the Clock, Draw Waveforms, and 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 200ns.
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:



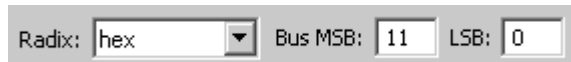
Waveform drawing and editing techniques can be found in *Chapter 1: Signals and Waveforms* in the online help.

## 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.

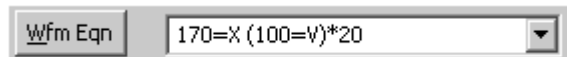
Add a virtual bus signal called **ADDR**:

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.



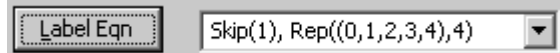
The A/D converter is driven by a clock that is 1/2 the frequency of the state machine clock **CLK0**, so the **ADDR** value should change every other clock cycle (this maintains the same address for the read out of each RAM cell's count data and its write back after it is incremented). The **ADDR** signal 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**
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.



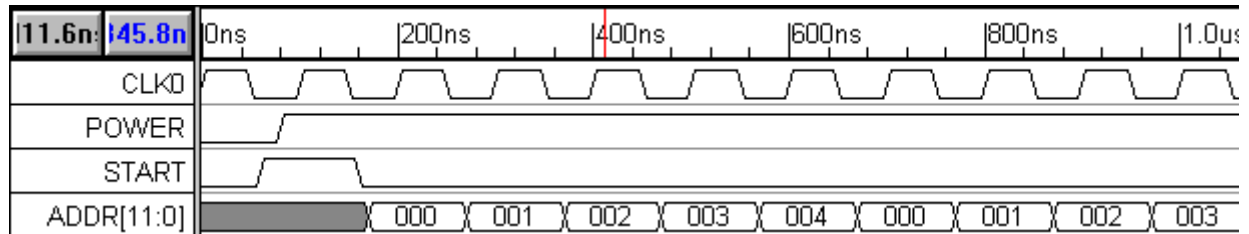
Next, we will label the states of the **ADDR** bus using a **Label Equation**. Each state could be labeled individually using the extended state field of the **HEX** dialog box, but labeling twenty states would take a long time. Instead, we will write an equation to label all the states at once. *Chapter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



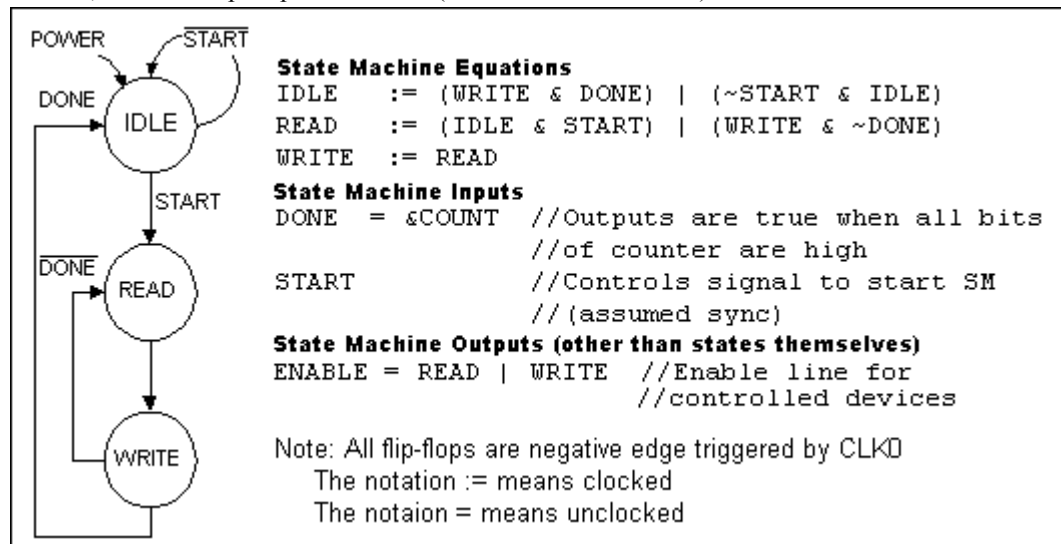
2. Press the **Label Eqn** button to apply the equation.

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.



### 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).



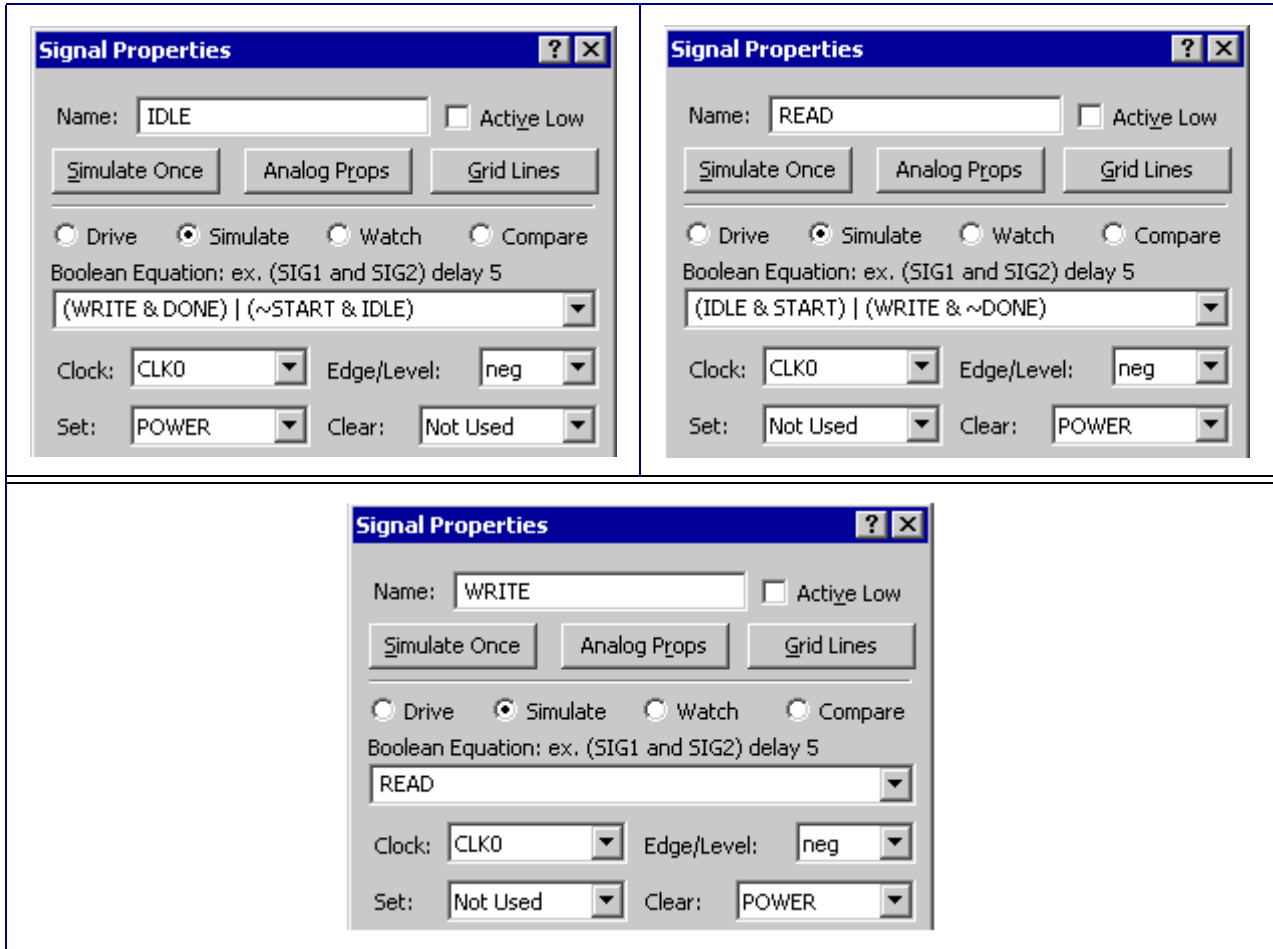
**Figure 2: State diagram and design equations for the histogram controller state machine**

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:



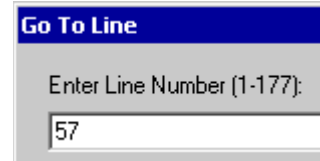
Notice the display **Compile Error** in the bottom right hand corner and notice that the state machine signal names turned gray. This is because the IDLE and READ equations reference a signal called DONE. This signal has not been defined so if you try to simulate you get errors. In the next section we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

#### 4) Checking for Simulation Errors

If you check the simulator log file, **simulation.log** `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:\SynptiCAD\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.

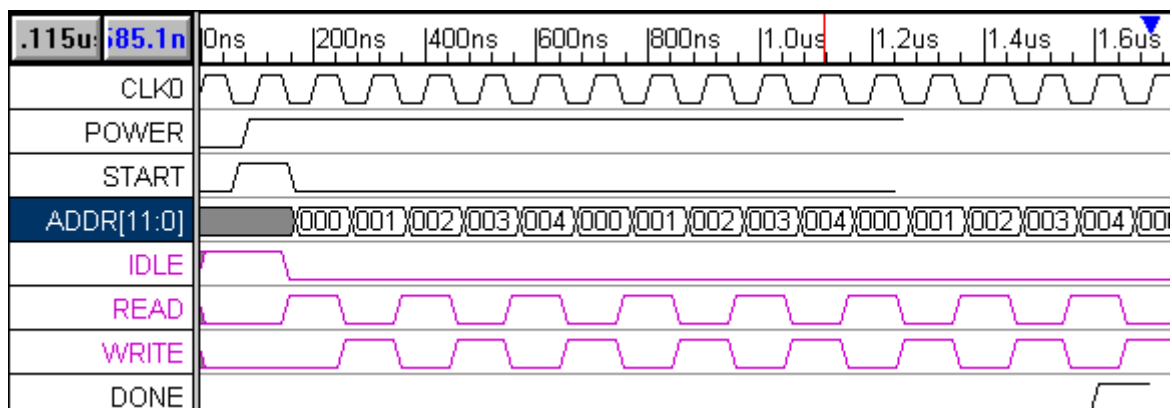
## 5) Incremental Simulation

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.

SynptiCAD 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. If the indicators still show an error, then the **simulation.log** file will help you to pinpoint the error in your diagram.

Simulation Good

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:

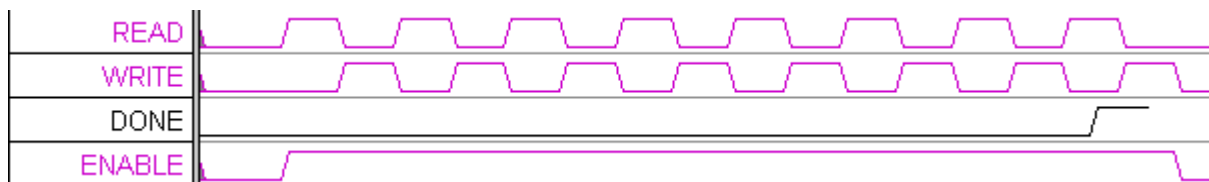
1. **Drag** the falling edge of the *START* pulse back to approximate **140 ns** (before the falling clock edge at 150 ns). This step causes the state machine to stay in the *IDLE* state (the *IDLE* signal stays high).
2. **Double click** on the falling *START* edge and enter a time of **160** into the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

## 6) Modeling Combinatorial 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 *unlocked*. If a signal other than *unlocked* 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:

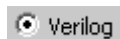
1. Create a new signal called **ENABLE**.
2. Enter the equation: **READ | WRITE** into the **Boolean Equation** edit box in the *Signal Properties* dialog.
3. Check the **Simulate** radio button.
4. Verify that *ENABLE* is the **OR** of *READ* and *WRITE*. If *ENABLE* did not simulate, use the techniques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



## 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
```

## 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)!

## 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]	7777	0000	7777	0000	7777	0000	

## 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		

## 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		0002		0001

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).

## 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
```

## 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!



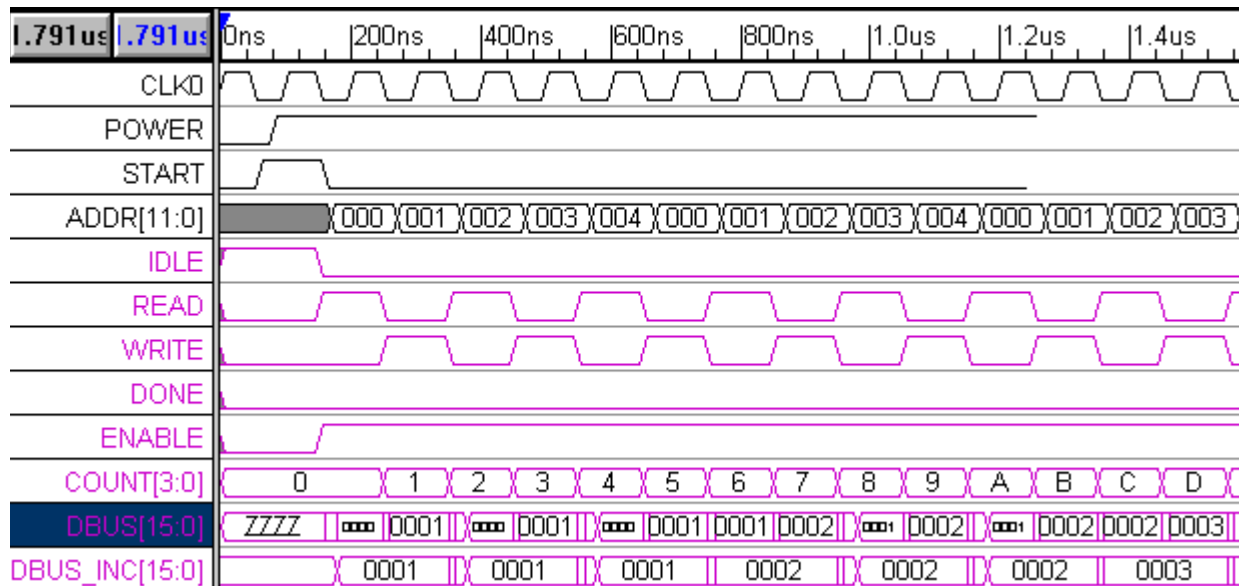




Figure 3: Completed Timing Diagram

## 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.

## 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;`

## 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. **srām.v** for this tutorial). Follow the instructions of your Verilog simulator to simulate these files together.

## 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**.

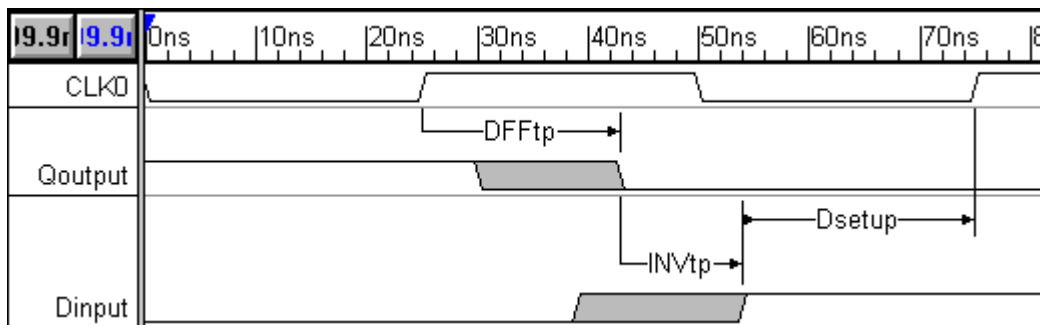
# Parameter Libraries

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.

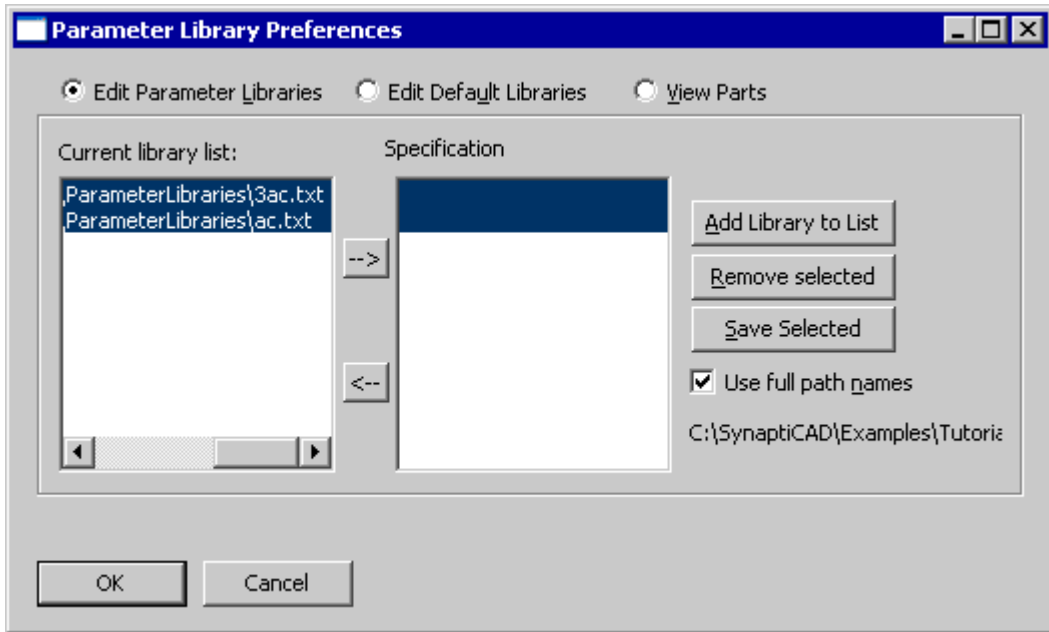


## 1) Adding Libraries to the Project's "Library Search List"

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.

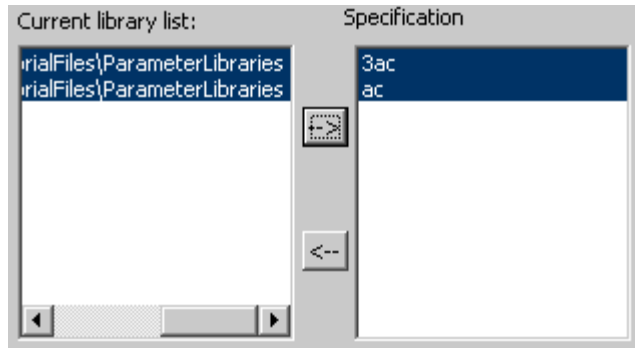
## 2) Setting Library Specifications

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 **3act.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.



## 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.

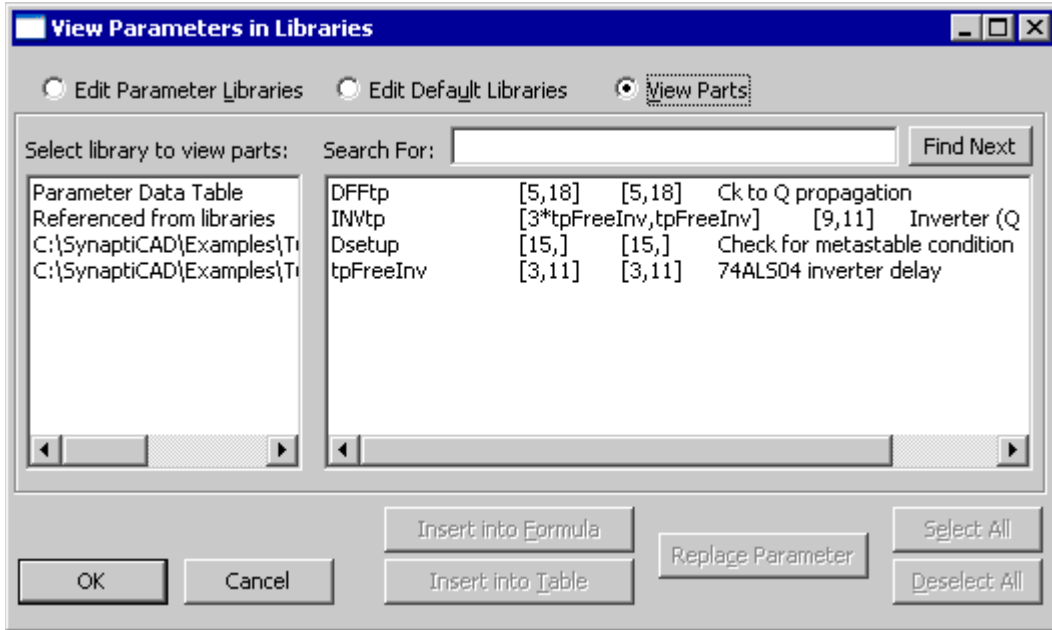
## 4) Referencing Parameters in Libraries

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.

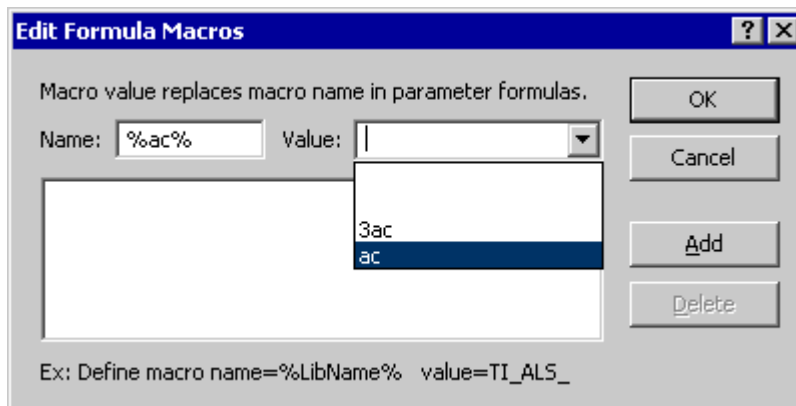
Repeat the above process for the min and max values of DFFtp, inserting **074;CK2Q\_tp** from the ac.txt library. Try using the **Search For** edit box in the *View Parameters in Libraries* dialog, instead of scrolling, to find a parameter name.

## 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.



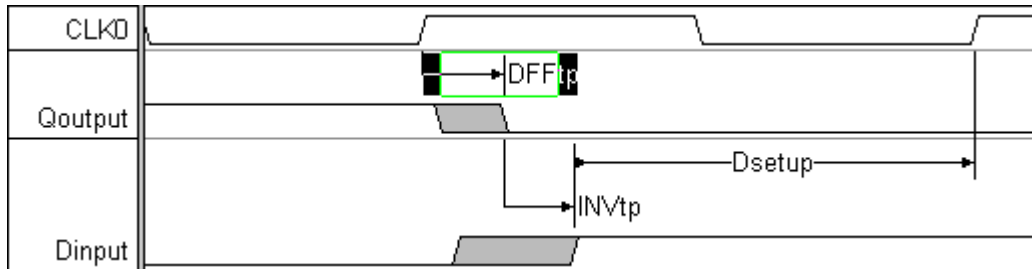
4. Click **OK** to add the macro to your macro list.

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.



# Advanced HDL Stimulus Generation

This tutorial describes how to generate Verilog and VHDL stimulus files using WaveFormer Pro, VeriLogger Pro and TestBencher 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.

TestBencher Pro customers should also work through the on-line TestBencher 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:

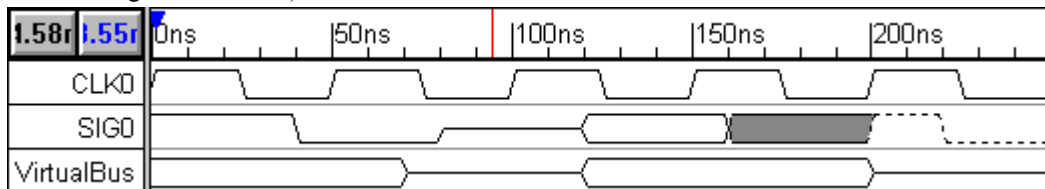
- clocks & signals
- graphical waveform states (high, low, tristate, valid, invalid, weak high, weak low)
- virtual buses with hex, binary, and other data values
- VHDL user-defined types and integer types

## 1) Getting Started

Get a Full Version License

If you are evaluating WaveFormer Pro, VeriLogger Pro or TestBencher 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.

## 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.

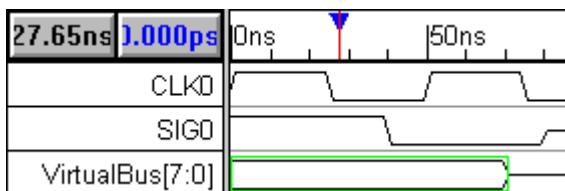
To demonstrate the hex and binary translations, we will edit the signal `VirtualBus` so that it will correctly export as an 8-bit bus. We will also use the **'b** and **'h** values to set the segment values and compare how they export in VHDL and Verilog. Later in the tutorial we will generate the Verilog 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.

### 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.

### 4) VHDL - Advanced Data Types

#### 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.



# Basic Verilog Simulation

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

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

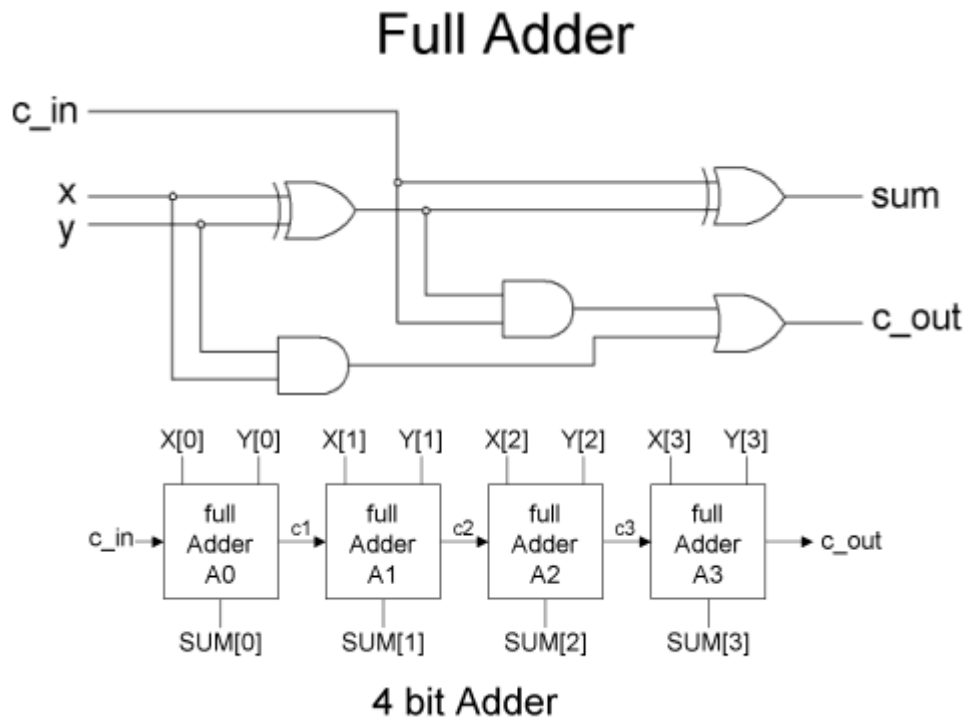


Figure 1: Schematic of the 4-bit adder simulated in this tutorial.

## Part 1: Project Management and Simulation

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

### 1.1) Add Files to the Project

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

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


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

In the *Project* window:

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

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

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


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

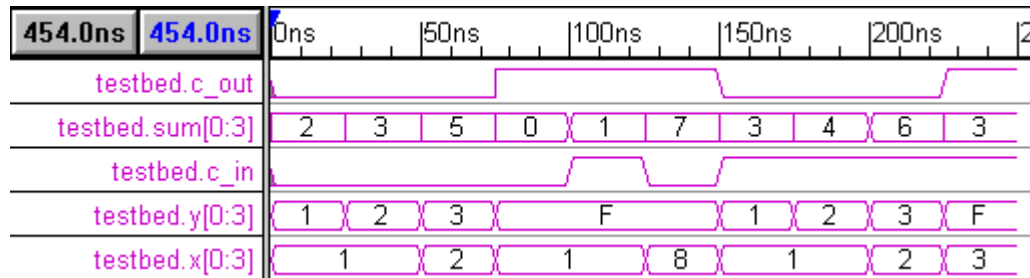
One module, **testbed**, is placed in the *Simulated Model* folder and surrounded by brackets to indicate that it is the top-level module (the highest-level instantiated component). All sub-modules can be viewed by descending the top-level module's tree. When the tree is expanded it can display the signals,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

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

## 1.3) Simulate the Project

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

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



## 1.4) Watch and View Internal Signals

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

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

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

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

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

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

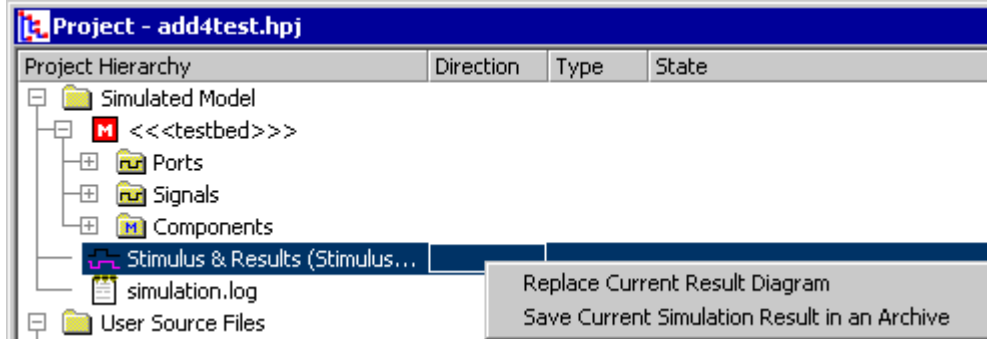
Next we will learn to save the project, waveforms, and source code. The project saves the simulation options and the names of the files contained on the project tree. It does not save the source code or the watched signals. To save the project:

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

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

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

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



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

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

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

## Part 2: Graphical Test Bench Generation

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


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

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

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

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

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

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



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

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

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

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



### 2.4) How to Draw Waveforms

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

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

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



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

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

### 2.5) How to Edit Waveforms

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

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

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

## 2.6) Draw the Stimulus Waveforms

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



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


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

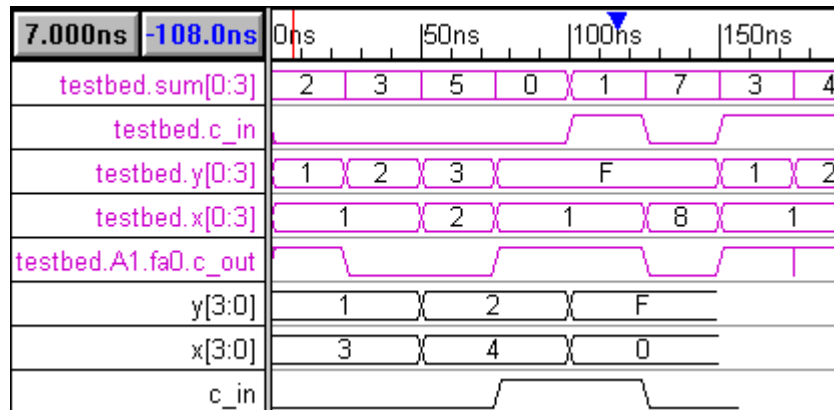
1. Double click on the first segment on the x signal to open the *Edit Bus State* dialog.
2. Verify that the default radix is **hex**.
3. Enter **1** into the *Virtual* edit box.
4. Press the **ALT-N** keys or **Next** button to move to the next segment on the signal.
5. Continue to enter values into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

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

## 2.7) Simulate Using the Auto Run Simulation Mode

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

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



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

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

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

## 2.8) Import and Generate Waveforms

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

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

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

```
8ns=Z (5=1 5=0)*5 9=H 9=L 5=V 5=X
```
3. Press the **Wfm Eqn** button to apply the above equation to the signal.

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

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

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

### 3) Breakpoints, Stepping and Tracing

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

# Reactive TestBench Tutorial

## 1) Overview

This tutorial introduces some of the optional reactive test bench feature set. This feature set is included with TestBench Pro and can be optionally be added to WaveFormer Lite, WaveFormer Pro, Datasheet Pro, and BugHunter Pro. When running any of these products, documentation on these features can be found in the *Reactive TestBench Generation* manual.

The following features will be covered in this tutorial:

- Cycle-based test bench generation
- **For Loop** Markers (see *Section 4.3: Loop Markers*)
- **Point Samples** (see *Section 3.3: Interpreting Sample Conditions and Blocking Points*) for checking model output
- **Sensitive Edges** (see *Section 1.9: Sensitive Edges*)
- **Bi-Directional Signals** (see *Section 1.1: Drawing Waveforms and Bi-Directional Signals*)

All of the relevant files for this tutorial can be found in the <SYNCAD\_INSTALL>\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.

## 2) 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 it's 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**.

### 3) Create Signals

#### 3.1 Extract Ports from MUT

If you're running TestBenchPro or BugHunter Pro you can create a new project that contains the **mymut.v** source file and use the **Extract MUT ports into Diagram** button to create all of the signals. If you're using Libero, the ports will automatically be extracted into a new diagram when WaveFormer is launched.

#### 3.2 Create Clock Waveform

Once the ports are extracted, convert the signal named **CLK** to a Clock by right-clicking on the name of the signal and selecting **Signal <-> Clock**. This will draw a clock waveform with a default frequency of **10MHz**.

#### 3.3 Set Default Clocking Signal and Edge

Next we 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 **Diagram Properties**. Select **CLK** as the default *Clock* to use and **pos** as the *Edge*. Then click **Update Existing** to set the clock for existing signals. Press **OK** to close the dialog.

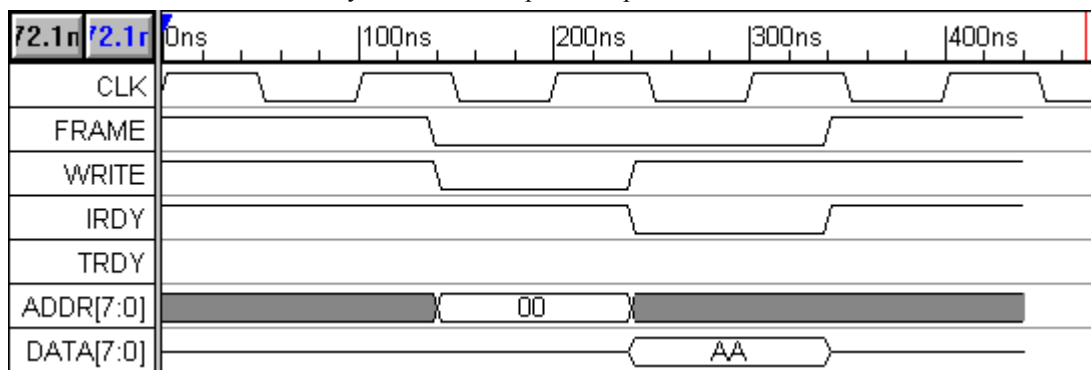
Following is an example of the difference between a cycle-based and time-based test bench. Both of these code segments were exported from the diagram you will be drawing in the next step. The example on the left is time-based and the example on the right is cycle-based.

Time-based	Cycle-based
<pre> #137; FRAME_driver &lt;= 1'b0 #3; WRITE_driver &lt;= 1'b0 ADDR_driver &lt;= 8'h00 #100; WRITE_driver &lt;= 1'b1 IRDY_driver &lt;= 1'b0; ADDR_driver &lt;= 8'hxx DATA_driver &lt;= 8'hAA #100; FRAME_driver &lt;= 1'b1 IRDY_driver &lt;= 1'b1; DATA_driver &lt;= 8'hzz #101; </pre>	<pre> repeat (2) begin   @(posedge CLK); end FRAME_driver &lt;= 1'b0;  WRITE_driver &lt;= 1'b0; ADDR_driver &lt;= 8'h00; @(posedge CLK); WRITE_driver &lt;= 1'b1; IRDY_driver &lt;= 1'b0; ADDR_driver &lt;= 8'hxx; DATA_driver &lt;= 8'hAA; @(posedge CLK); FRAME_driver &lt;= 1'b1; IRDY_driver &lt;= 1'b1; DATA_driver &lt;= 8'hzz; @(posedge CLK); </pre>

**Table 2: Comparing Source for Time-Based to Cycle-Based**

#### 4) 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.



#### 5) Add Wait for TRD Assertion

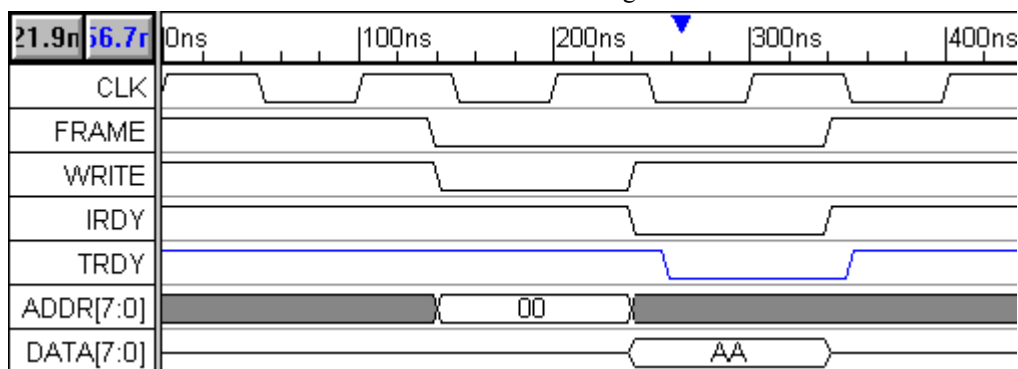
There are two ways to perform this step. One method uses the **Sensitive Edge** feature and will wait indefinitely for **TRDY** to assert. The other method uses a **Sample** instead, where a timeout can be specified. Both methods are explained below. Before doing either method though, you need to draw the expected **TRDY** waveform shown below.

**Note:** **TRDY**'s waveform is blue because it is an input to the diagram, so the data shown is predicted data, not data to be driven. The tool automatically determined the direction of **TRDY** when the **Extract Ports from MUT** step was performed.

##### 5.1 Draw Expected TRDY Waveform

**Note:** If you want to specify a timeout for this wait, skip this step and go to 5.3.

Double-click on **TRDY** to open the *Signal Properties* dialog, enable the **Falling Edge Sensitive** check box, and click **OK**. When this is enabled, the test bench will wait on every drawn falling edge on **TRDY**. This is indicated graphically by an arrow on the falling edge. 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**.



#### 5.2 Wait Methods

##### 5.2.1 Wait Indefinitely Using Sensitive Edges

**Note:** If you want to specify a timeout for this wait, skip this step and go to 5.2.2.

Double-click on **TRDY** to open the *Signal Properties* dialog (see *Section 1.4: Add Signals*), enable the **Falling Edge Sensitive** check box, and click **OK**. When this is enabled, the test bench will wait on every drawn falling edge on **TRDY**. This is indicated graphically by an arrow on the falling edge. 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**.

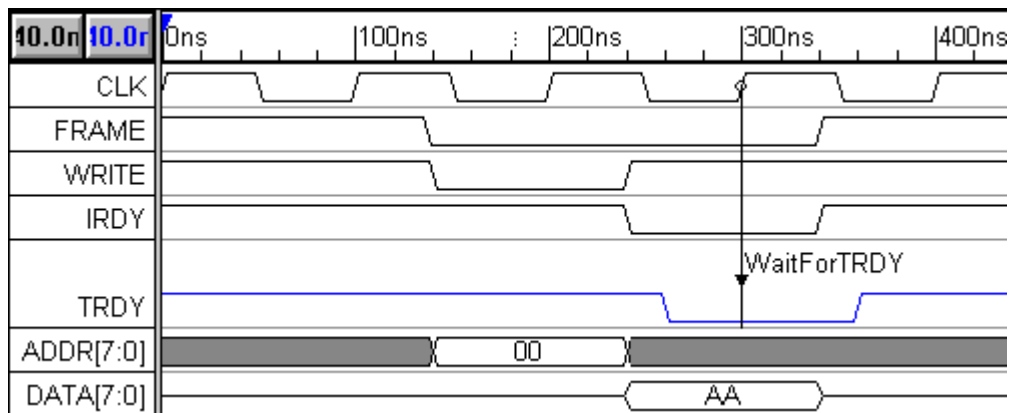
### 5.2.2 Wait with a Timeout Using a Sample (see *Chapter 3: Samples*)

**Note:** Skip this step if you performed step 5.2.1.

Depress the **Sample** button. To create a sample, left-click on the rising edge of **CLK** at **300ns**, then right-click on **TRDY** at **300ns**. 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 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:

- Disable the **Full Expect** check box.
- Specify **100** for the *Multiplier*.
- Enable the **Blocking** check box.

These three options work together to achieve the “wait with timeout” behavior we want. With **Full Expect** off and the **Multiplier** set to 100, this Sample will wait for up to 100 clock cycles for **TRDY** to assert. The **Blocking** check box causes the rest of the transaction to wait on the Sample to finish. Otherwise, the Sample would be run in parallel with the stimulus. More details on these options can be found in the *Reactive TestBench Generation* manual. Here's what the diagram should look like at this point:

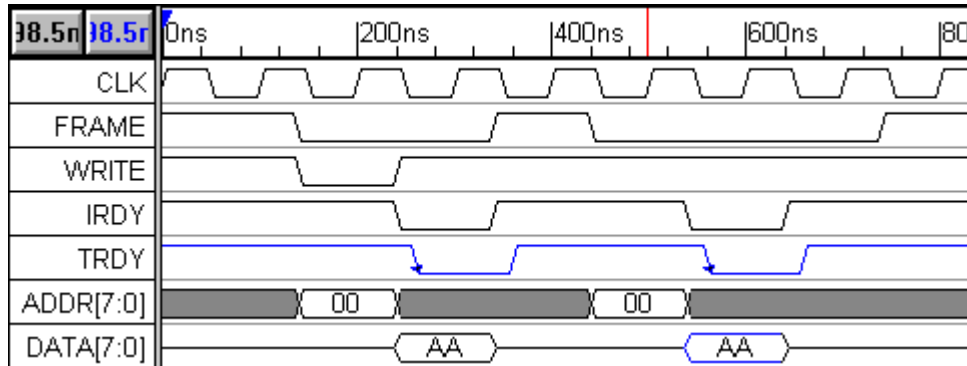




## 6) Draw Single Read

### 6.1 Draw the Waveforms

Draw a complete read transaction following the write transaction. Here is what the waveforms should look like (assuming you used the edge sensitive wait; the sample version will look slightly different, of course):



### 6.2 Disable Drive for the DATA Segment

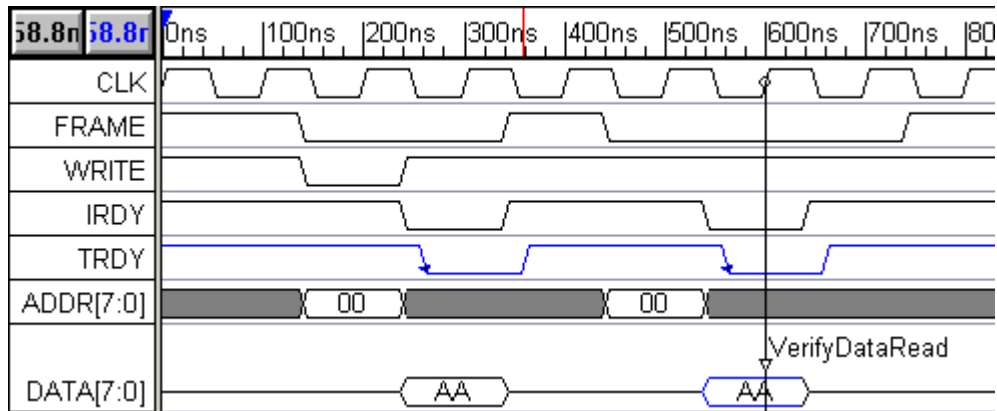
The test bench must not drive the **DATA** bus during the read cycle to avoid contention as the MUT will be driving it then. Since the **DATA** bus is a bi-directional signal, you can specify which parts of the waveform are driven by the test bench and which are not. One way to do this is to draw the bus with the **TRI** state, but in this case we need to specify the expected data on the bus, so the **TRI** state can't be used. Instead, double-click on the waveform segment of **DATA** that happens during the read. Disable the **Driven** check box and click **OK**. 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).

## 7) Add a Sample to Verify Data Read from MUT

Depress the **Sample** button, then left-click on the positive clock edge at **600ns** and 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). This is the default behavior of the Sample. Next, make the following changes to the Sample:

- Double-click on the Sample name and change its *Name* to **VerifyDataRead**
- Click the **HDL Code** button to open the *Code Generation Options* dialog.
- Select **Display Message** for the *Then Action*. Select **Note** for the severity level of this action. This will make the Sample display a note during simulation when it succeeds.
- Click **OK** to close these dialogs.

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



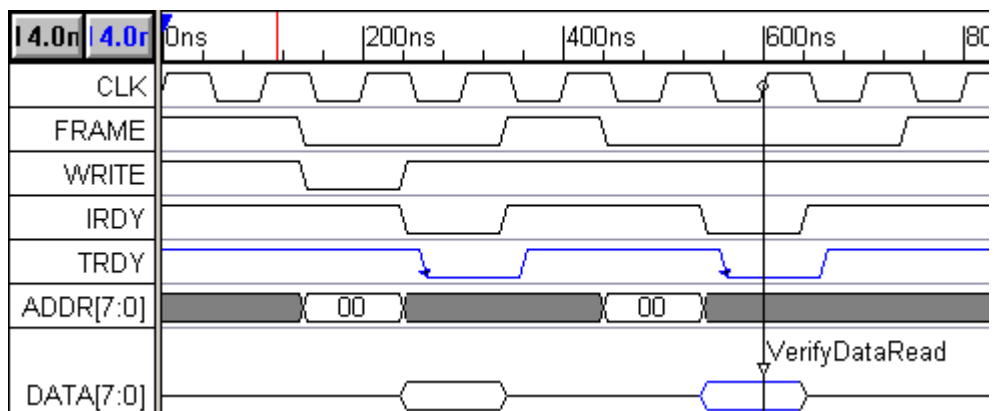
## 8) Drive Data Using a Test Vector Spreadsheet File

This step will use an input file to drive the **DATA** bus during the write cycle. This will increase the effectiveness of the test bench by writing different patterns to different addresses. The basic idea is to create a user-defined array variable that is initialized from a file. Here are the steps to create the 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**.
- Set *Data Type* to **2\_state** then change *MSB* to **7**.
- Enable the **Initialize Structure With File** checkbox near the bottom of the dialog. Browse to the `inputData` directory and select **inputData.txt**. Hit **OK**.

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.



## 9) Create For-Loop to Perform Multiple Writes and Reads

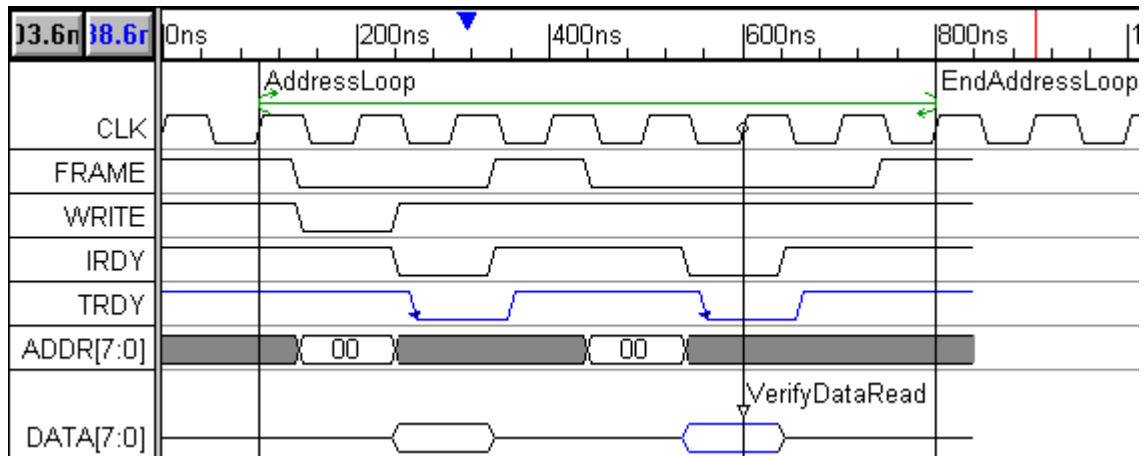
This step sets up the diagram to perform multiple writes and reads.

**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 the first Marker to open the *Edit Time Marker* dialog.
- Select **For Loop** in the *Type* drop down list.
- Set *Name* to **AddressLoop**.
- Set *Index* to **address**.
- Set *End* to **10**.
- Click **OK** to close the dialog.
- Double-click the second Marker to open the *Edit Time Marker* dialog.
- Change *Type* to **Loop End** and click **OK**.



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



## 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.

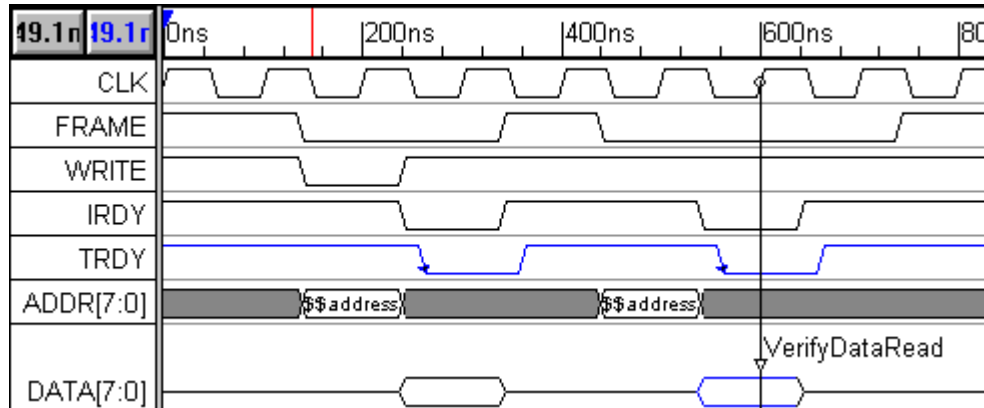
**Note:** The primary purpose of this tutorial is to demonstrate various features available to all Reactive TestBench users. So, there are several steps that may not make as much sense for TestBench users. For instance, two transactors could have been created instead of one: one for the write cycle and 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.



## 11) Alternatives

### 11.1 Consecutive Writes followed by Consecutive Reads

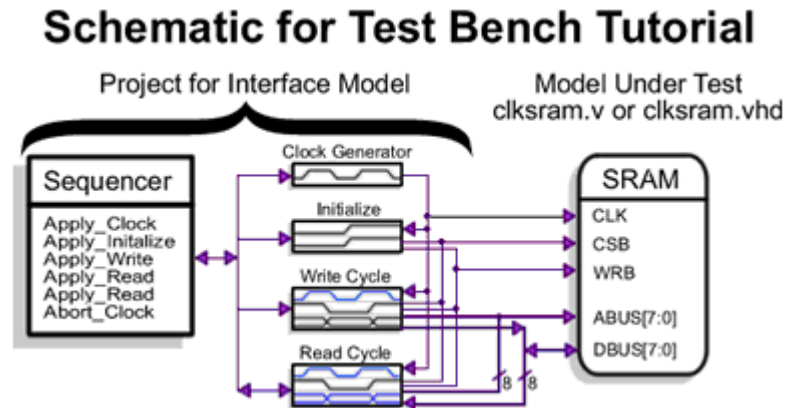
If you want to perform multiple writes concurrently, followed by multiple concurrent reads, then two for-loops are needed. The array of data can be referenced in each loop in the same manner already demonstrated.

### 11.2 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]**.

# TestBencher Pro: Basic Tutorial

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



## Preparation

Before we begin there are few things to setup and understand:

1. This tutorial requires a full version license or an evaluation license. If you are evaluating then you can obtain a license by completing the form under **Help > Request License** menu item and contacting our sales department. To check that you have a good license, verify that you can save a timing diagram.
2. This tutorial assumes that you are familiar with the SynaptiCAD timing diagram editing environment. If you would like more information on the drawing environment then work through the short **Help > Tutorial > Basic Drawing and Timing Analysis** tutorial.
3. This tutorial can be use to generate VHDL, Verilog, TestBuilder, OpenVera and e code. Sometimes a file name will be written as *filename.<language extension>*. This means that the file extension will be different depending on the language used: Verilog \*.v, VHDL \*.vhd, TestBuilder \*.cpp, OpenVera \*.vr, and e code \*.e.

## 1) Create a Project

TestBencher Pro uses a project file to represent and to control the generation of a bus-functional model (BFM) component. The information in the project file is displayed in the Project window and context sensitive menus provide a list of actions that can be performed for the elements in the project tree. In this section the project will be created, the Model Under Test (MUT) file will be added to the project, and the template diagram will be constructed.

### 1.1 Use the New Project Wizard to Create a Project

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

To create a new project:

1. Select the **Project > New Project** menu option to launch the *New Project Wizard* dialog.
2. Enter **sramtest** in the *Project Name* edit box. The actual project directory will be a subdirectory below the displayed path in the *Project Directory* edit box. This subdirectory will have the same name as the project.

**Unix users:** Make sure that you have read/write access to the directory specified in the *Project Directory* edit box.

3. From the *Project Language* dropdown, select the code generation language.
4. Check the **Transaction-based Test Bench Generation** checkbox.
5. Click the **Next** button to move to the second page of the *New Project Wizard*.
6. Note that the name of the *New Template* is **sramtest** (the name of the project). TestBencher will use this file to generate the top-level module of the test bench. The *Original Template*, named **tbench**, is copied into the *New Template* file.
7. Type **CLK** into the **Default Clock** dropdown, and choose **neg** from the **Edge** dropdown box. Selecting a default clock causes the test bench to be cycle-based; if no clock is specified, the test bench will be event-based.
8. Check the **Create Default Clock Generator** box. This will cause TestBencher to create a slave timing diagram called **Clk\_generator.btim** that will drive the *CLK* signal.
9. Click the **Finish** button to close the *New Project Wizard*, create the project, and populate the *Project* window.

## 1.2 Add the MUT to the Project

Next we will add the clocked SRAM model file to the project. TestBencher uses the model under test files to extract the signal and port information for use in the transaction diagrams. TestBencher also uses the MUT file information to instantiate it in the component model (template file).

**Note for Remote Simulators:** If your simulator or HVL tools are running on a different computer than TestBencher Pro, then the external simulator integration feature requires that all files used for the project be in the project directory, or a subdirectory thereof. If you are working a remote simulator, copy the appropriate MUT file (**clksram.v** or **clksram.vhd**) from the **SynaptiCAD > Examples > TestBencherBasicTutorial** directory into the project directory prior to adding the MUT to the project.


To add the MUT to the project:

1. Right-click the *User Source Files* folder in the *Project* window and select the **Add File(s) to User Source File Folder...** from the context menu option. This will open the *Add Files...* dialog.
2. Select the file to use as the MUT from the **SynaptiCAD > Examples > TutorialFiles > TestBencherBasicTutorial** directory (or from the project directory if your simulator is on a remote machine):  
Verilog model file is **clksram.v**.  
VHDL model file is **clksram.vhd**.
3. Click the **Open** button to close the dialog and add the file to the *User Source Files* folder in the *Project* window.

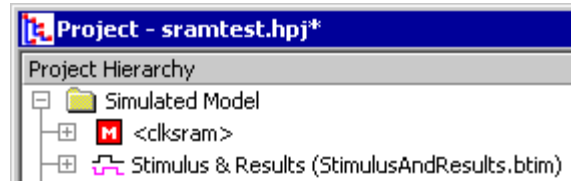
## 1.3 Extract Port Information from the MUT into the Template Diagram

When 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 this diagram and we are going to add the port signals for the clocked SRAM.

To extract the ports from the SRAM into the template diagram:

1. In the *Project* window, under the *Template Diagram* folder, double click on **sramtest\_templateDiagram.btim** to open the template diagram window.
2. Click the **Extract Ports from MUT**  button. This will build the MUT and insert the signals for the MUT ports into the template diagram.

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




**Note:** If `<clksram>` was not generated as the MUT, then change the simulation preferences by choosing the **Options > Diagram Simulation Preferences** menu. Check the **Auto-create test bench and tree** check box. Press the **Extract Ports from MUT** button to rebuild the MUT.

### 1.4 Modify the Template Diagram

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

To add an end diagram marker:

- Click on the **Marker** button  on the diagram button bar.
- Click on the fourth falling edge of the CLK signal (at 350 ns) to select it. Then right-click to draw a marker that is attached to the edge.
- Double-click on the marker to open the *Edit Time Marker* dialog.
- Select a *Marker Type* of **End Diagram** from the drop down list box. This end diagram marker will force the transaction to end at the fourth falling edge of the CLK signal.
 

Marker Type:
- Select **Type** from the *Display Label* list box. This will cause the marker to display its type rather than its name.
- Click **OK** to close the *Edit Time Marker* dialog.
- Use the **File > Save All Files** menu option to save the project and the template diagram.

The template diagram should look like the following:

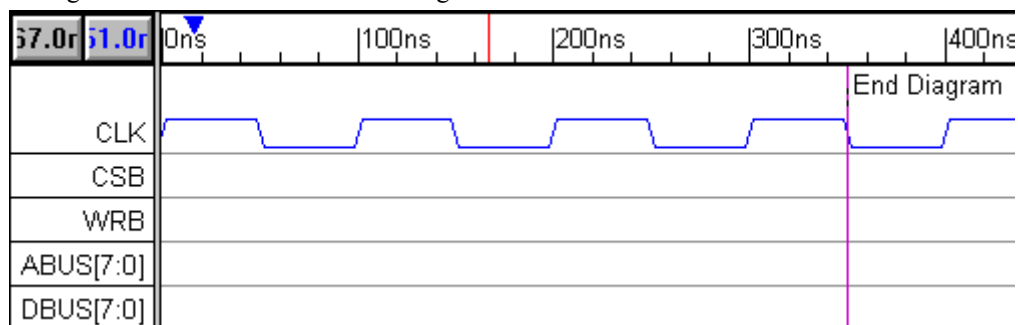


Figure 2: Completed Template Diagram

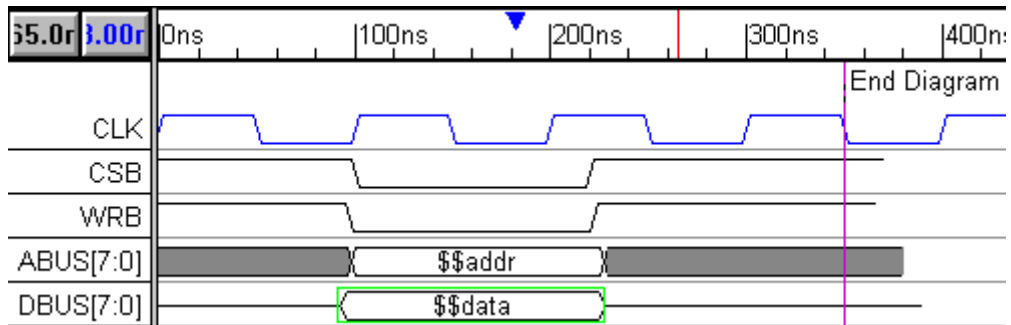
## 2) Create the Write Cycle Transaction Diagram

TestBencher Pro uses timing diagrams that represent reusable bus transactions to generate the test bench. This tutorial will use two timing diagrams, `tbread.btim` and `tbwrite.btim`, to represent the read and write cycles used in testing the memory module. First, the write cycle diagram will be created. Then this diagram will be used as a basis for creating the read cycle diagram. Variables will be used in the diagrams so that values can be passed into the address and data buses.

### 2.1 Draw the Timing Diagram for the Write Cycle

This section explains how to create the timing diagram that represents the write cycle transaction.

1. In the *Project* window, right click the *Transaction Diagrams* folder and select **Create a new Master Transaction** from the context menu. This will cause the *Save As* dialog to open.
2. Name the file **tbwrite** and press the **Save** button. This will copy the template diagram to the new file, list the file in the *Transaction Diagram* folder, and open the new diagram.
3. Draw the following waveforms (the state values will be added in the next section):




**Figure 3: Completed Write Cycle Diagram**

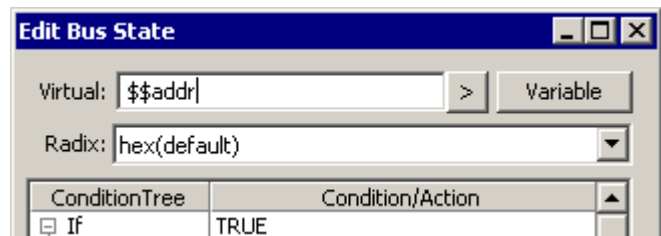
Note: If you have trouble drawing the waveforms, then refer to the **Basic Drawing and Timing Analysis** tutorial.

## 2.2 Add Parameterized State Values for Write Cycle

The next step is to add state variables to the timing diagram so that values for the address and data buses may be passed into the test bench transaction. Parameterized state values, called **state variables**, are passed into the transaction call in the top-level template file, and are used to provide state or comparison values during transaction execution. The write cycle diagram will have a state variable for the value on the address bus, and a state variable for the value on the data bus. When the top-level template file is modified, values will be passed into the state variables.

To add the address and data state variables to the diagram:

1. Double click on the valid segment in the center of *ABUS* to open the *Edit Bus State* dialog.
2. Type **\$\$addr** into the **Virtual** edit box. The "\$\$" in front of the variable name indicates that this is a state variable. If the "\$\$" is missing, TestBencher Pro will assume that this is the value of the address rather than a variable that will accept a value at a later time.
3. Click on the valid segment in the center of *DBUS* to move the focus of the *Edit Bus State* dialog to the new segment.
4. Type **\$\$data** in the **Virtual** edit box.
5. Click **OK** to close the *Edit Bus State* dialog. The two edited segments will display the state variables.
6. Click the diskette icon  on the main toolbar to save the timing diagram.



## 3) Create the Read Cycle Transaction Diagram

The read cycle will initiate a read with the clocked SRAM and monitor the data bus to verify the result of the read. For the read cycle, the data bus will be an input signal (not driven like the write cycle), and the \$\$data variable will be used for comparison with the actual value driven by the SRAM.



### 3.1 Create Read Cycle Diagram and Add it to the Project

Since the signals for the read diagram are so similar to the write diagram, a modified copy of **tbwrite.btim** can be used to create the read diagram.

Create the read cycle timing diagram and add it to the project:

1. In the **tbwrite** diagram window, right-click in the Label window and select the **Save As...** menu option to open the *Save As* dialog.
2. Name the file **tbread**, and press the **Save** button. This will create a new file, but you still will need to add the file to the project.
3. Right-click in **tbread**'s Label window, and select **Add Master Diagram to Project** from the context menu. This will add **tbread** to the *Transaction Diagrams* folder in the *Project* window.

### 3.2 Edit the Waveforms for Read Cycle

The WRB and DBUS signals need to be changed for the Read cycle. The write control signal, WRB, should stay high (inactive) for the duration of the read. And during the read the DBUS signal will be driven by the SRAM, so the data segment of the signal needs to be set to input. Also since our SRAM is clocked the data comes out on the clock cycle after the chip select signal, CSB, goes active.

To edit the waveforms:

1. Make the WRB signal high for the entire read cycle. Select the center segment and press the delete key to remove the low signal segment.
2. Shift the start of the DBUS data segment to 200ns. Hold down the <2> key (the number 2 key) on the keyboard, while dragging the starting transition to 200ns. The <2> key causes transitions to the right of the selected edge to move with the dragged edge.
3. Set the DBUS data segment to be a blue input segment. Double click on the data segment to open the *Edit Bus State* dialog, uncheck **Driven (Export to source code)** checkbox.

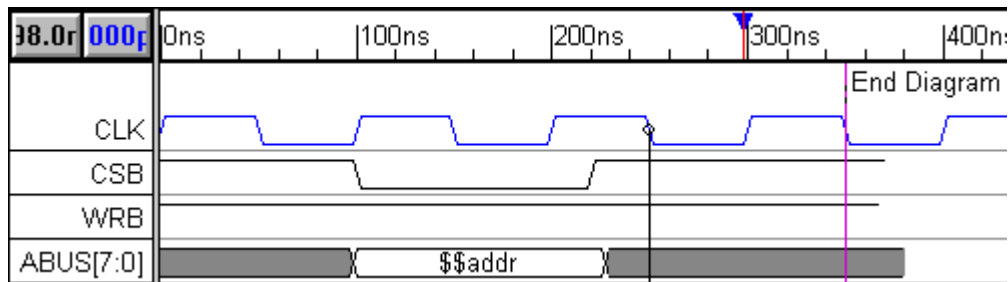



Figure 4: Completed Read Cycle Diagram

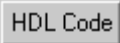
### 3.3 Add a Sample to Verify Data

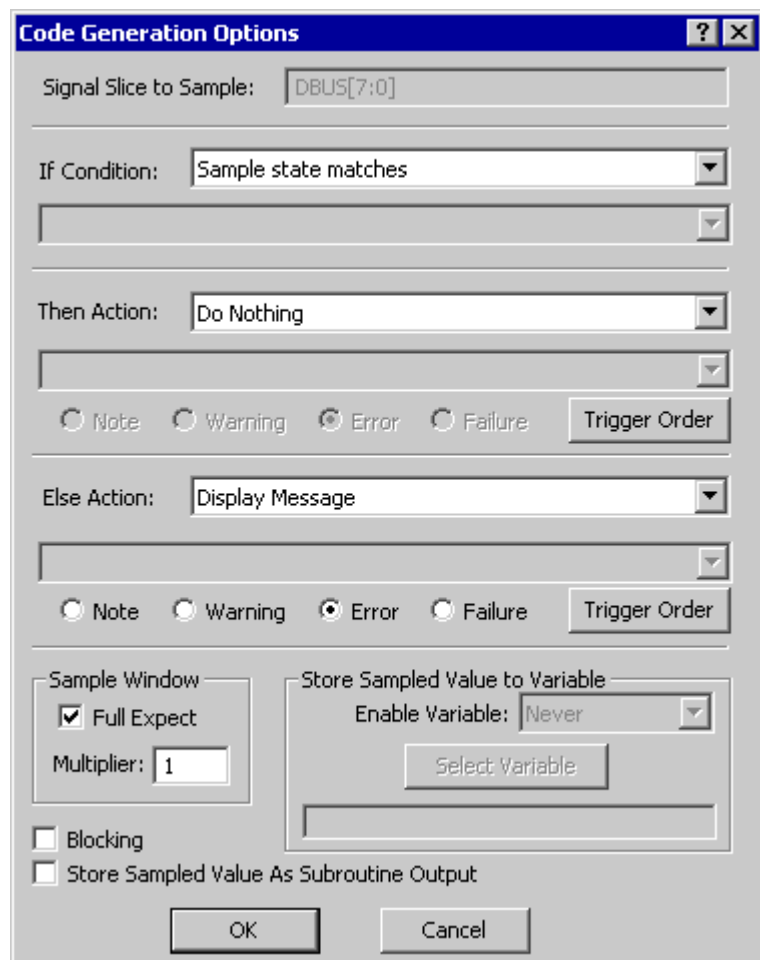
Next a Sample will be added to the timing diagram. Samples compare the actual state value of an input signal to the expected state value, and conditionally react to the results of the comparison.

To add a Sample:

1. Click on the **Sample** button  on the button bar.
2. Click on the **third falling edge** (250ns) of **CLK** to select the edge.
3. Right-click near the end of the **blue valid segment** on **DBUS**. This adds a Sample parameter named **SAMPLE0** that lines up with the third neg edge of the **CLK** signal. Refer the image in the previous section.

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

1. Double-click on the sample name **SAMPLE0** in the drawing window to open the *Sample Properties* dialog.
2. Press the **HDL Code** button in the dialog to open the *Code Generation Options* dialog. 
3. In the *If Condition* dropdown, select **Sample state matches**. This means that during simulation, the test bench will compare the actual value on the data bus with the value passed into the timing diagram (\$\$data).
4. In the *Then Action* dropdown, select **Do nothing**. If the value on the data bus matches the value of \$\$data, then the circuit is working properly and no action should be taken.
5. In the *Else Action* dropdown, select **Display Message**. This means that if the values don't match, a message will be displayed during the simulation.
6. Below the *Else Action* dropdown, choose the **Error** radio button. These radio buttons allow a severity level to be defined for the message that is displayed.
7. Click **OK** to close the *Code Generation Options* dialog.
8. Click **OK** to close the *Sample Properties* dialog.
9. Save the timing diagram by selecting **File > Save Timing Diagram** from the main TestBencher menu.



The image shows the 'Code Generation Options' dialog box. It has a title bar with a question mark and a close button. The dialog is divided into several sections:

- Signal Slice to Sample:** A text field containing 'DBUS[7:0]'.
- If Condition:** A dropdown menu set to 'Sample state matches'.
- Then Action:** A dropdown menu set to 'Do Nothing'.
- Severity Level:** Radio buttons for 'Note', 'Warning', 'Error' (selected), and 'Failure'. A 'Trigger Order' button is to the right.
- Else Action:** A dropdown menu set to 'Display Message'.
- Severity Level (Else):** Radio buttons for 'Note', 'Warning', 'Error' (selected), and 'Failure'. A 'Trigger Order' button is to the right.
- Sample Window:** A checkbox for 'Full Expect' (checked) and a 'Multiplier' field set to '1'.
- Store Sampled Value to Variable:** A section with an 'Enable Variable' dropdown set to 'Never' and a 'Select Variable' button.
- Other Options:** Two unchecked checkboxes: 'Blocking' and 'Store Sampled Value As Subroutine Output'.
- Buttons:** 'OK' and 'Cancel' buttons at the bottom.

## 4) Create the Initialize Transaction Diagram

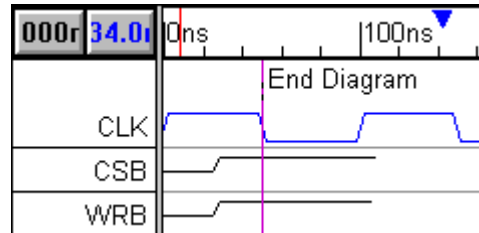
When drawing the waveforms for a transaction diagram it is important to remember that transactions do not automatically include an event at time zero and that only the drawn events are driven. This is a feature that allows transactions to be reused any time during simulation without implying any initialization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

### 4.1 Draw the Initialization Waveforms

Create the Initialization diagram by first copying the template diagram, removing the extra signals, and drawing the waveforms.

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

2. Name the file **tbinitialize** and press the **Save** button. This will save the diagram, add it to the *Transaction Diagram* folder, and open the new diagram.
3. Remove the **ABUS** and **DBUS** signals, because the tri-state bus signals do not need to be initialized. Select the **ABUS** and **DBUS** signals by clicking on them, and then press the **<delete>** key to delete the selected signals.
4. Draw the following waveforms:




**Figure 5: Completed Initialization Diagram**

#### 4.2 Move the End Diagram Marker for Initialization Diagram

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

To move the **End Diagram** marker:

1. Double-click on the marker to open the *Edit Time Marker* dialog.
2. Select **Attach to Edge** from the radio buttons.
3. Click **OK** to close the *Edit Time Marker* dialog. This will put TestBencher into a special select mode.
4. Click on the first negative clock edge (at 50ns) to attach the marker to that edge.
5. Click the diskette icon  on the main toolbar to save the timing diagram.

### 5) Modify the Sequencer Process

Inside the primary template file for the project is a *Sequencer Process*. The Sequencer Process is the place in the top-level test bench that defines the order in which the timing transactions are applied to the model under test.

The *Insert Diagram Subroutine Calls* dialog generates diagram apply calls so you do not need to memorize the function syntax. Each timing diagram generates three task calls: Apply, Apply-nowait, and Abort. Apply runs the transaction in a blocking mode, and Apply-nowait runs the transaction concurrently with other transactions. The *Master/Slave Diagram Setting* determines how many times a transaction executes. Master Transactors, like the Read, Write, and Initialize diagrams run once and stop. Slave Transactors like the Global Clock Generator run in a looping mode until an Abort call is received.

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

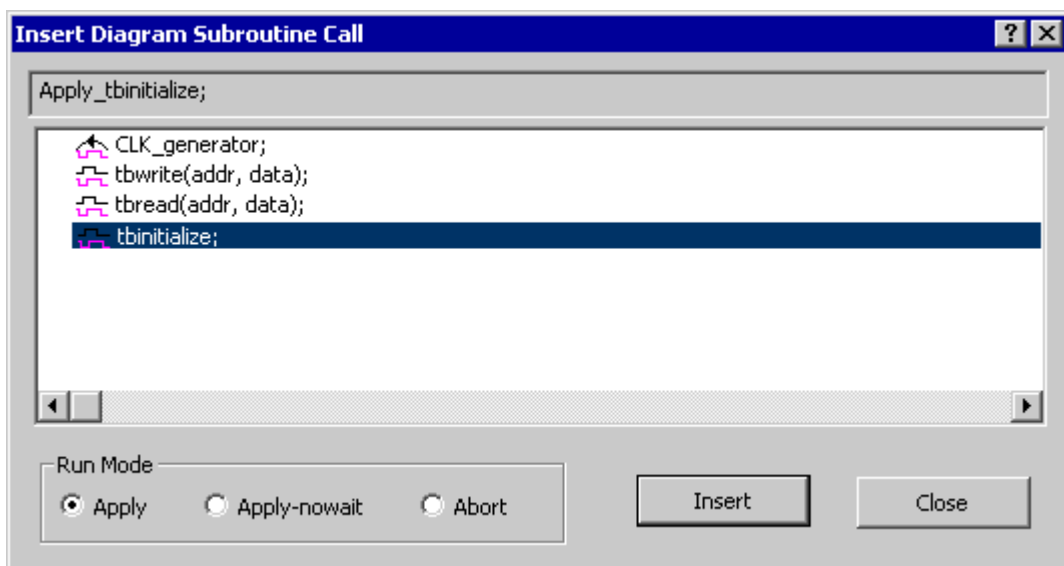
An alternative method to placing transaction calls in the sequencer process is to create a file external to the bus-functional model with transaction calls and during simulation read the transaction calls from a file (see *Section 9.4: Transaction Manager and Test Reader* in the online TestBencher Manual).

#### 5.1 Adding Apply Calls to the Sequencer Process

Use the *Insert Diagram Subroutine Calls* dialog to add apply statements to the Sequencer. We will first start the clock, initialize the control signals, write to the SRAM, the read from the SRAM twice, and then abort the clock.

To edit the sequencer process:

1. In the *Project* window, double click on the *Component Model* folder to open an editor window with the **sramtest** template file.
2. Scroll down in the **sramtest** editor window near the end of the file until you find the comment block that has this line:  
  
Transaction Sequencer - After this comment, define how to apply transactions to the model under test using:
3. Click in the **sramtest** editor window below this comment so that the blinking cursor is in the place where the apply statement should be added.
4. Right-click in the editor window and select **Insert Diagram Calls** to open the *Insert Diagram Subroutine Call* dialog.



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

Use the *Insert Diagram Subroutine Calls* dialog to add the apply calls. When you select a slave diagram, the dialog will automatically default to **Apply-nowait**, because most of the time slave diagrams will run concurrently with other diagrams. When you select a master diagram, the dialog will automatically default to **Apply**, because most of the time master diagrams run in a blocking mode:

1. Double click on the **CLK\_generator** entry in the *Insert Diagram Subroutine Calls* dialog. This adds an apply call to the editor window.
2. Double click on the **tbinitialize** entry.
3. Double click on the **tbwrite** entry.
4. Double click on the **tbread** entry TWO times to insert the code to add two read calls.
5. Select **CLK\_generator** entry, choose **Abort** radio button, and then press the **Insert** button to insert the code. This will add the abort call to stop the clock generator.
6. The apply call should look similar to the following code block. Different languages may have extra parameters.

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

```

//
// - Transaction calls (Insert Diagram Calls in right-click menu)
// - Source code in Verilog
//*****
Apply_CLK_generator_looping_nowait;
Apply_tbinitalize;
// Apply_tfwrite(addr, data);
Apply_tfwrite(addr, data);
// Apply_tbread(addr, data);
Apply_tbread(addr, data);
// Apply_tbread(addr, data);
Apply_tbread(addr, data);
Abort_CLK_generator;

```

## 5.2 Providing Values for Variables in Timing Transactions

The **tfwrite** and **tbread** transactions have parameterized state values. These values are passed to the transaction in the Apply statements.

To set the values of the state variables in the transaction apply calls:

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

For Verilog type:

```

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

```

For VHDL type:

```

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

```

For OpenVera type:

```

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

```

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

Notice that the **tfwrite** apply statement writes the hex value AE to memory cell F0. The **tbread** diagram calls will then read the value from the same memory cell. The data values provided in the **tbread** diagram calls will be used to compare with the actual value. The first call to **tbread** will expect to find a value of hex AE in the address F0. The second call to **tbread** will expect to find the hex value EE instead. This will cause the sample to report an error during the second execution of **tbread**.

## 6) Generate Test Bench and Simulate

At this point all the timing diagrams have been created and you have edited the Sequencer process. Next we will generate the test bench and simulate the entire design.

### 6.1 Setup the Simulator

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

To configure a third-party simulator:

1. Choose the **Options > Simulator and Compiler Settings** menu option. This will open the *Simulator and Compiler Settings* dialog.
2. From the **Simulator and Compiler tools** dropdown select the appropriate simulator.
3. Enter the directory that contains the simulator executable in the **Simulator Path** edit box.
4. Click **OK** to close the *Simulator and Compiler Settings* dialog.


Select the third-party simulator:

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



### 6.2 Generate the Test Bench and Simulate

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

To generate the test bench:

1. Click on the **Make TB** button  on the main 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.
2. 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:

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

In the bottom right corner, a **Simulation Good** status message indicates that the simulation has reached a successful end.

### 6.3 Examine Report Window Results

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

Examine the log file:

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

```
Running...
TB> Note: In "sramtest_CLK_generator" at 0.000ns: Executing LOOPING
TB> Note: In "sramtest_tbinitalize" at 0.000ns: Executing ONCE
TB> Note: In "sramtest_tbinitalize" 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
```

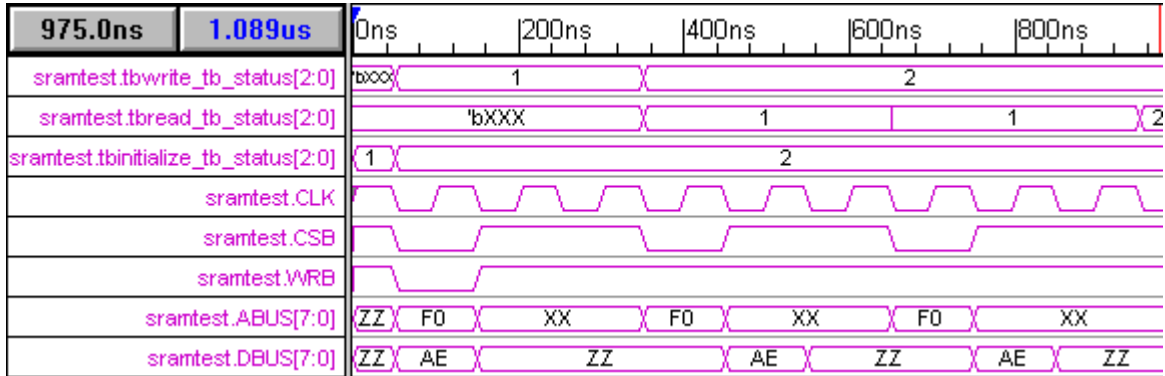
Normal exit

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

### 6.4 Examine the Stimulus and Results Diagram

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

1. 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:



2. 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.
3. 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.
4. The read cycles read back the data from the memory.