Reactive TestBench

Generation Option

User's Manual

www.syncad.com

Reactive TestBench Generation Option User's Manual
(rev 10.0) copyright 1994-2005 SynaptiCAD

Trademarks

For latest product information and updates contact SynaptiCAD at:

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

email: sales@syncad.com

phone: (540)953-3390

Table of Contents

Introduction

The Reactive Test Bench Generation Option bridges the gap between the stimulus waveform test benches that are native to most of the SynaptiCAD product line and the bus-functional model generation of TestBencher Pro. This option allows users to describe single timing diagram test benches that react to the model under test and generate pass/fail reports. The Reactive Test Bench Generation Option can be added to WaveFormer Pro, WaveFormer Lite, DataSheet Pro, VeriLogger Pro, and BugHunter Pro.

With "Reactive Test Bench Generation" users have the option of drawing "expected" waveforms on the MUT output ports and adding "samples" to the waveforms to test for specific cases. During simulation the code generated by the samples would watch the output from the model under test and compare it to drawn states. The samples can perform a variety of functions such as pausing the simulation to debug a problem, reporting errors and warnings, user-defined actions, and triggering other samples.

The Reactive Test Bench Generation also includes markers that can be used to wait for activity from the model under test and/or loop over a section of a diagram. Markers can also be used to call user-written HDL functions ands tasks from within a diagram.

Reactive test bench generation also allows the option of creating "clock-based" test benches as well as the "time-based" test benches currently supported by the stimulus based generation models. Clock-based test benches delay in clock cycles instead of times, allowing the user to change his clock frequency without needing to change his timing diagram. Clock-based test benches are also required when testing using high-speed "cycle-based" simulators.

Chapter 1: Waveforms and Signals

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

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

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

Generation of Reactive HDL Code

The Reactive Export feature (included in TestBencher Pro and optionally in other products) includes the generation of Boolean Equations with delays, registers, latched signals and behavioral Verilog code. Boolean equations and registered logic are useful for modeling interface glue logic that is not part of the model under test itself. The equations are entered through the Logic Wizard section of the Signal Properties, which can be reached by double-clicking on a signal name. Sections *12.1 Boolean Equations*, *12.2 Advanced Gate Representation* and *12.3 Register and Latch Equations* discuss the methods useful for generating interface glue-logic in a reactive test bench.

1.1 Drawing Waveforms and Bi-Directional Signals

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

To Draw a Waveform:

- Click the type of state that you want to add in the group of 7 states on the right side of the *Signal Button Bar*.

- Click in the waveform section of the *Diagram* window to the right of the signal or bus name at the approximate time that you want the state transition to occur. This will place the transition in the waveform. Waveforms are built from left to right.

- Repeat the first two steps until you have completed the signal's waveform.

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

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

- Double-click on the input segment. This opens the *Edit Bus State* dialog.

- Uncheck the **Driven** check box. This indicates that the test bench does not drive this segment; this segment will be an input to the test bench.

- Click **OK** to close the dialog or use **<Alt>-N** or **<Alt>-P** (or the **Next** or **Previous** buttons) to edit other segments on the same signal. The segment for which you unchecked the driven flag should now be colored blue.

1.2 Driving Waveform States with Variables

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

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

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

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

To edit the state of a valid signal segment:

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

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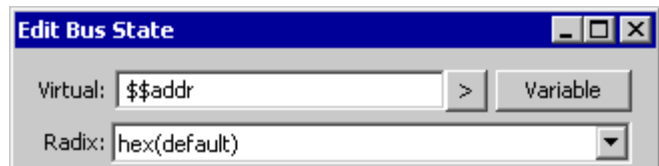
- To hard code a value, type the value in the **Virtual** edit box.

- To add a state variable, type the variable name using a $$ prefix into the **Virtual** edit box. For example, *$$data* might be the name of the variable for the value of data bus. This variable will appear in the timing diagrams apply call.

- To add a diagram-level variable:

    - Click the **Variables** button to open the *Select Variables* dialog. Double-click on the variable and click **OK** to close the dialog. The variable name with a **@** prefix will be added to the Virtual box.

    OR

    - Click the Variables Menu Button [>] to display a list of variables that can be inserted into the equation.

- Click **OK** to close the dialog box.

Note: State values can also be conditionally driven using the *Condition Tree* in the bottom of the *Edit Bus State* dialog. For more information see *Section 1.3: Driving Conditional State Values*.

1.3 Driving Conditional State Values

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

To create a conditional drive for the state value:

- Double-click the segment that is to be conditionally driven to open the *Edit Bus State* dialog.

- Right-click on the **If** row and choose **Add Condition** from the context menu to open an edit box or double click in the **Condition/Action** column of the **If** row.

- Type the text for the condition. The condition must be written in the generated language of the transaction and it must equate to a Boolean equation when evaluated during simulation.

- Next, add the state values to the **Then** or **Else** rows by right-clicking choosing **Add Variable** or **Add State** menu option. Or double-click in the **Condition/Action** column and edit the state.

- Optional: Complex conditionals can be created using the **Add If...Then...Else** context menu. This option is available for any existing **Then** or **Else** row. Selecting this option causes a nested **If...Then...Else** to be added to the branch of tree that was selected.
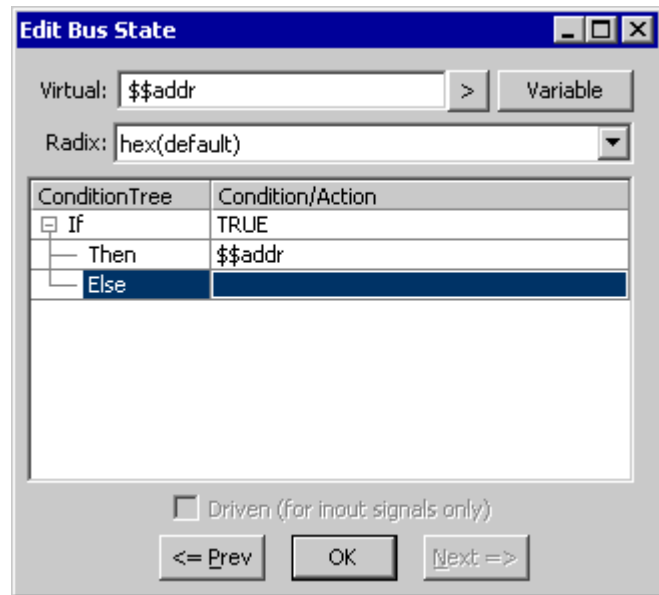
1.4 Adding Signals

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

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

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

To add a Signal, Clock, Bus or Spacer:

- Click the appropriate button in the first group of four. This will add the Signal, Clock, Bus or Spacer to the timing diagram. Spacers are just for adding space to the diagram and do not generate code.

- If you added a signal, clock or bus, then double-click the name of the new object to open the *Signal Properties* dialog.

- Edit the **Name**. If the signal is to be hooked up to a signal in the HDL model, then use the same name.

- Edit the signal type using the **language Type** drop down list box in the bottom of the dialog.

- Edit the signal size using the **MSB** and **LSB** edit boxes. Clocks are always one bit wide.

- Edit the **Direction** using the drop down list box. The following directions are available:

    - **Output** indicates that the signal is output from the diagram.

    - **Input** indicates that the signal is what you expect the model under test to generate during simulation (these signals are inputs to the timing transactions, driven by the model under test). In the timing diagram, Sample parameters usually end on an input signal, indicating that the input signal should be checked for an expected value at that point on the signal.

    - **Inout** indicates that the signal is bi-directional (see *Section 1.1: Drawing Waveforms and Bi-Directional Signals*). Inout signals contain driven and un-driven signal segments. Driven segments act like signals of type **output**.

    - **Internal** indicates that the signal will only be used internally to the diagram component.

- The **Clock** and **Edge/Level** specify the clocking signal for the waveform. In TestBencher, these will be automatically set by the *Project Wizard* options, however, you can pick a different system clock signal and edge using these controls. TestBencher users can also change the default clock using the *Diagram Properties* clock.

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

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

1.5 Boolean Equations with Delays

SynaptiCAD's Boolean equations combinatorially relate one signal to other signals in the diagram. To describe a signal with a Boolean equation:

- Double-click on a signal name to open the *Signal Properties* dialog.

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

- Enter a Boolean equation into the edit box.

> Boolean Equation: ex. (SIG1 and SIG2) delay 5
>
> [▼]

- To view or edit the Verilog HDL code used by the simulator, select the **HDL Code** radio button.

The Boolean Equation edit box accepts Boolean equations in VHDL, Verilog, and SynaptiCAD's enhanced equation syntax. The SynaptiCAD format supports the operators: **and**, **or**, **nand**, **nor**, **xor**, **not**, 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 analysis is the main advantage that SynaptiCAD's format has over the VHDL or Verilog format. Instead of min/max timing, Min-Only or Max-Only simulations can be performed by changing the **Options > Simulation Preferences > Timing Model** drop-down list box. Below are sample Boolean equations:

(SIG0 and SIG1 and SIG3) delay 20ns

> This models a 3-input AND gate with a 20ns delay.

(SIG0 delay 20ns) and (SIG1 delay 10ns)

> This models an AND gate with 2 different input delays.

(SIG0 and SIG1) delay GateDelay

> Assume GateDelay is a delay parameter with a min time of 15ns and a max time of 20ns. This models an AND gate with a delay between 15ns and 20ns. Each edge of the simulated signal will have a grey uncertainty region of 5ns.

1.6 Advanced Gate Representation

The **Boolean equation** edit box of the *Signal Properties* dialog can accept several advanced operators, like conditional expressions and signal concatenation. These operators can be used to model multiplexers, tristate gates, and multi-bit signals. The following demonstrates several of these modeling techniques:

Conditional Expressions for Multiplexers and Tristate gates: The normal C language conditional expression of *conditional ? if_expr : else_expr* can be used inside the Boolean Equation edit box to model multiplexers and tristate gates. Some examples are:

> For a Tristate Gate: **EnableSig ? SIG0 : 'bz**

> For a 2-1 MUX: **S0 ? SIG0 : SIG1**

> For a 4-1 MUX: **S1 ? (S0 ? SIG0 : SIG1) : (S0 ? SIG3 : SIG2)**

Multi-bit Equations are specified by setting the **MSB** and **LSB** of the signal (located at the bottom of the *Signal Properties* dialog). To change a simple 1-bit equation to a 4-bit equation, all you have to do is set the MSB of the signals involved to 3.

Concatenation of Signals is supported using the Verilog concatenation operator. You must set the **MSB** in the *Signals Properties* dialog to the proper size. If the size of the concatenated signal is larger than the receiving signal, then the most significant bits are dropped. Some examples of the concatenation operator:

> - Signal Concatenation: **{SIG0, SIG1}**

> - Concatenating bit-slices: **{SIG0[3:0], SIG1[7:4]}**

1.7 Register and Latch Equations

The generation and simulation of registered and latched equations are supported through the Reactive Export and Interactive HDL simulator features. The logic wizard is used to enter information about the circuit for code generation.
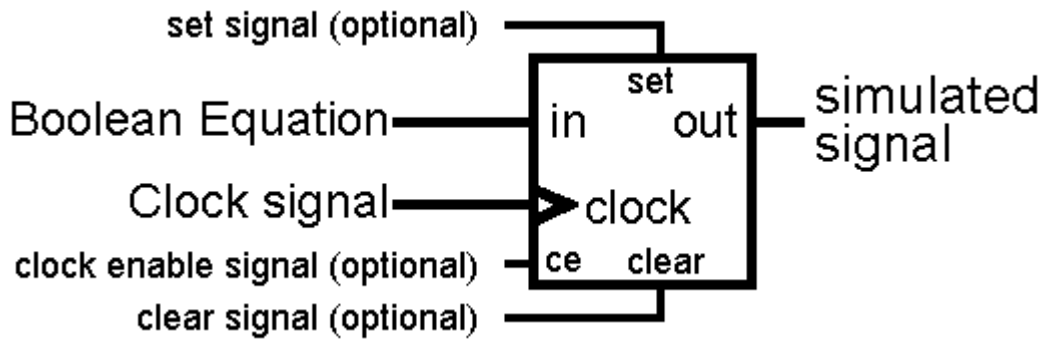
To describe a signal that is registered or latched:

- Double-click on a signal name to open the *Signal Properties* dialog.

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

- Enter the input signal name into the **Boolean equation** edit box. The input Boolean equation can either be the name of the input signal or an equation that conditions the input signal.

- Choose the clocking signal from the **Clock** drop-down list box. The clocking signal can be any clock or signal in the timing diagram.

- Choose the type of edge or level triggering from the **Edge/Level** list box. For a Register circuit choose **neg** for negative edge triggering, **pos** for positive edge triggering, or **both** for edge triggering. For a Latch circuit choose either **low** or **high** level latching.

- The **Set, Clear,** and **Clock Enable** are optional signals that model the set, clear, and clock enable lines of the register or latch. If "Not Used" is chosen for a line, then that line is not modeled. These lines can be active low or high and synchronous or asynchronous depending on the settings in the *Advanced Register and Latch Controls* dialog.

- The **Advanced Register** button opens the *Advanced Register and Latch Controls* dialog which determine how this individual register is generated. The global defaults can be defined using the **Options > Simulation Preferences** menu. This dialog controls the following options:

  - **Clock to Out:** Describes the delay from the triggering of the clock signal to a change on the output edge. This setting supports both a Low to High model and a High to Low model.

  - **Setup:** Describes the time for which the input must be stable before the clock-triggering event. If a min/max time pair is entered, Setup will use the min time. Any violations of this setup time will be reported to the simulation log (see the simulation log information below).

  - **Hold:** Describes the time for which the input must remain stable after the clock-triggering event. If a min/max time pair is entered, Hold will use the min time. Any violations of this hold time will be reported to the simulation log (see the simulation log information below).

  Simulation Log Information:

  Simulation-Enabled products will report timing violations to the **verilog.log** log file, shown in the report window (sample error message shown below). Otherwise, simulation log messages will be reported through the console of the third party simulator that is being used (e.g. ModelSim).

  In the **Clock Enable** area:

  - **Active Low:** If checked, the clock will be enabled when the *clock enable* line is low. If unchecked, the clock will be enabled when the *clock enable* line is high.

In the **Set and Clear** area:

- **Active Low:** If checked, the set and clear lines will control the output when they are low. If unchecked, then the set and clear lines will control the output when they are high.

- **Asynchronous:** If checked, then the set and clear lines will control the output anytime they are active. If unchecked, the model is synchronous and an active set or clear line does not affect the output until the next clock trigger event.

- To view or edit the Verilog HDL code that is used by the simulator, select the **HDL Code** radio button.
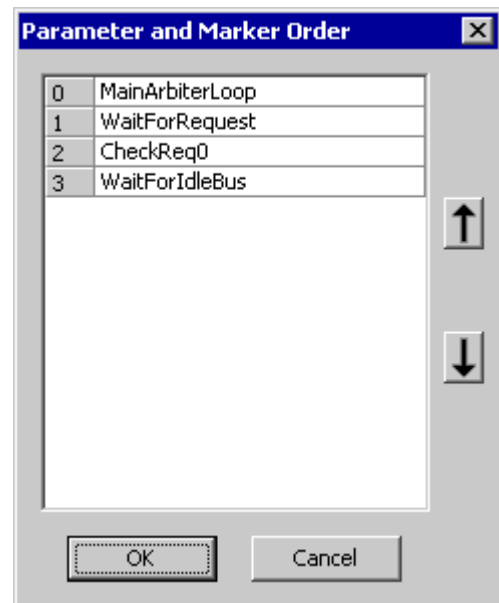
1.8 Controlling the Triggering Order of Parameters

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

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

To order Parameters and Markers attached to the same edge:

- Double-click the edge that triggers the parameters and markers to open the *Edge Properties* dialog.

- Click the **Trigger Order** button to open the *Parameter and Marker Order* dialog.

- Drag and drop the rows to arrange the parameters and markers in the desired order. You can also use the arrow buttons on the right side of the dialog to move selected items up or down.

- If you need to review the properties of an item before setting the order, you can double-click the name of the object in the row to open the *Properties* dialog for that object.

- Click the **OK** button to close the *Parameter and Marker Order* dialog.

- Click the **OK** button to close the *Edge Properties* dialog.

Displaying the order of parameter and markers in the timing diagram

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

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

- Double-click the parameter or marker to open the *Parameter Properties* or *Marker Properties* dialog.

- Select the **Name and Order** option from the *Display Label* dropdown list.

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

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

- Select the **Options > Drawing Preferences (Style Sheet)** menu option. This will open the *Drawing Preferences (Style Sheet)* dialog.

- Select the **Name and Order** selection from the Parameter *Display Label* dropdown list.

- Select the **Name and Order** selection from the Marker *Display Label* dropdown list.

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

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

1.9 Sensitive Edges

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

Sensitive edges cause wait statements to be inserted for that edge. These waits will block the clocking domain that contains the sensitive edge (see *Section 1.10: Transaction Architecture* for more details on clocking domains).

To enable sensitive edges on a signal:

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

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

- Check the **Rising Edge Sensitive** checkbox or the **Falling Edge Sensitive** checkbox. Enabling both checkboxes will cause both rising and falling edges to be sensitive.

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

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

1.10 Transaction Architecture

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

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

Clock domains

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

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

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

Signals

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

Blocking Constructs (Sensitive Edges, Samples, and Markers)

There are three different types of constructs that can be used to block the execution of a clock domain. A *sensitive edge* (*Section 1.9: Sensitive Edges)* will cause its clock domain to wait on the edge, which will block all other items in that same clock domain until the edge is detected. A Sample that has the **blocking** setting checked (*Section 3.3: Interperting Sample Conditions and Blocking Points*) will block its clock domain until the sample completely finishes, including execution of its *then* or *else* action. And a *Wait Until Marker (Section 4.5: Wait Until Marker)* will block its clock domain until the condition specified becomes true. If the marker is attached to an edge it will only check for the condition at each clock edge of the clock domain.
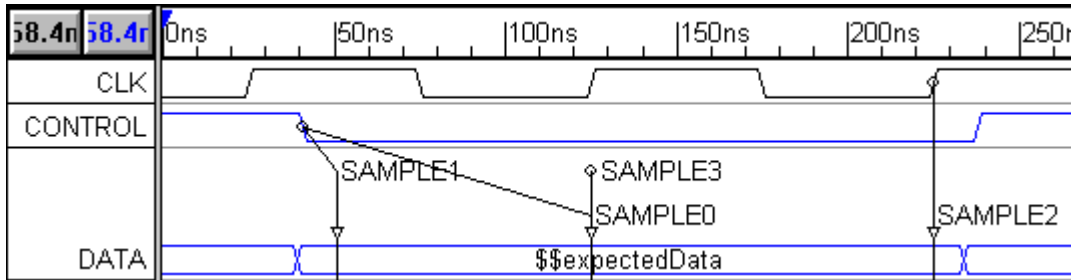
Samples

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

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

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

The example diagram below contains three domains: CLK_pos, CONTROL_neg, and Unclocked.

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

- **SAMPLE2**: triggered from the third clock edge.

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

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

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

Delay Parameters

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

Setups and Holds

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

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

- **CLK_pos** triggers both S0 and H0 at the second positive edge of CLK.

- **CONTROL_B_neg** triggers S1 at the first negative edge of CONTROL_B.

- **CONTROL_B_pos** triggers H1 at the first positive edge of CONTROL_B.
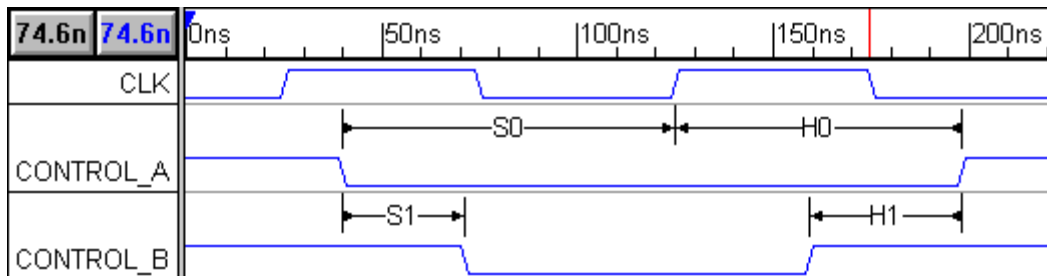
Markers

When looping behavior is needed over a particular set of clock cycles or time, then *Looping Markers* have to be used (see *Section 4.3: Looping markers* for more details on markers). They will only loop over the clocking domain that they are placed in.

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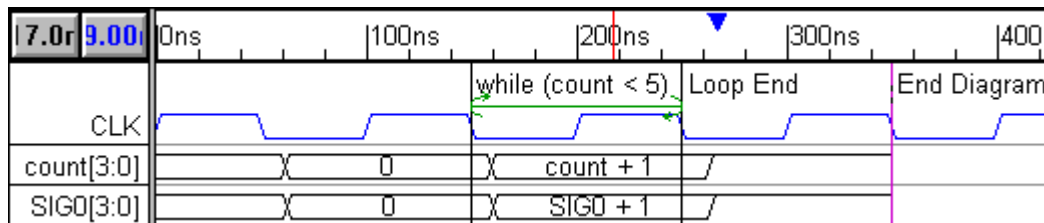
Output Clocks (Clock generators)

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

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

1.11 Diagram Properties

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

To edit the Diagram Properties:

- Open the diagram for which you will be changing the properties.

- In the *Diagram* window, right-click in the signal label area and choose **TestBencher Diagram Properties** from the context menu. This will open the *TestBencher Diagram Properties* dialog.

Including HDL Code Library Files

If you have external code modules that you want to make available to the transaction then you can use the interface in the *Diagram Properties* dialog to make that code available. Files can either be included before the transaction, using the equivalent of the Verilog *include* statement, or files can be included inside the module. The method for including code within the transaction varies by language. If possible the code is included using something like the *include* statement and if that concept is not supported then the code is echoed within the transaction. If you have HDL functions or tasks that you would like to write and use within a transaction then use the **Class Methods** dialog as discussed *Section 5.2: Class Methods*. Class Methods is a newer interface that is more flexible and it makes it easier to modify the code and parameters of the functions.

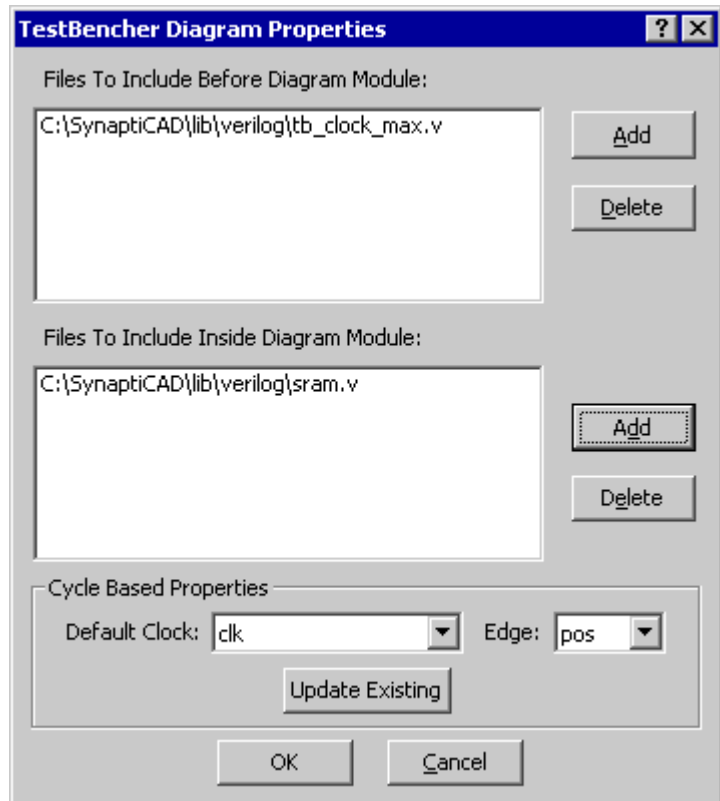**To Add an HDL Code Library File to the Diagram:**

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

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

Cycle Based Properties

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

- The **Default Clock** and **Edge** settings provide default values for the clocking signal and sensitive clock edge in a diagram.

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

TestBencher Diagram Properties

Files To Include Before Diagram Module:

```
C:\SynaptiCAD\lib\verilog\tb_clock_max.v
```

Add
Delete

Files To Include Inside Diagram Module:

```
C:\SynaptiCAD\lib\verilog\sram.v
```

Add
Delete

Cycle Based Properties

Default Clock: clk   Edge: pos

Update Existing

OK   Cancel

Chapter 2: Delays, Setups and Holds

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

This chapter will cover delays, setups, and holds that are parameters that perform actions between two signal transitions. Samples are placed on signal states (not transitions) and monitor the state of a signal. Samples are the main type of parameter used in TestBencher timing diagrams and they are covered in detail in *Chapter 3: Transaction Samples*.

Delays

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

Setups and Holds

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

2.1 Adding and Editing Parameters

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

To add a Delay, Setup, or Hold:

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

> - Click on a transition to select it. For a delay this is the forcing transition. For a setup or hold this is the transition that will be monitored.

> - Right-click on the second transition to add a parameter between the first and second transitions. For a delay this is the transition that will be moved. For a setup or hold this is the control signal.

> - Double-click the name of the parameter to open the *Parameter Properties* dialog for that parameter and edit the properties of the parameter.

The *Parameter Properties* dialog has many settings that control how the parameter is displayed in the timing diagram and these features are covered in the Timing Diagram Editor on-line help *Section 4.4: Parameter Properties* in the TestBencher manual. TestBencher uses only a few controls for code generation and these are discussed below. A few additional controls are available for delays (discussed in *Section 2.2: Delays*) and samples (discussed in *Chapter 3: Transaction Samples*). The following controls are common to all parameters and are used in code generation:

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

- The **Min** and **Max** edit boxes specify the minimum and maximum time for the parameter to execute. Each type of parameter handles the **Min** and **Max** values differently; for more information, see the sections on delays (*Section 2.2*), setups and holds (*Section 2.4*), and samples (*Chapter 3*).

- The **Is Apply Subroutine Input** checkbox, for TestBencher, allows you to generate ports between the Component Model and the timing transaction with which to specify the values to use for the **Min** and **Max** settings of the parameter. If only one of the values is specified, then a port will only be made for that value. If there is no value specified for either setting, then a port will be made for the min value by default.

- The **Enable HDL Code Generation** checkbox allows you to turn the code generation for the parameter on and off without removing the parameter from the timing diagram. This checkbox must be checked in order to produce any HDL code for the parameter.

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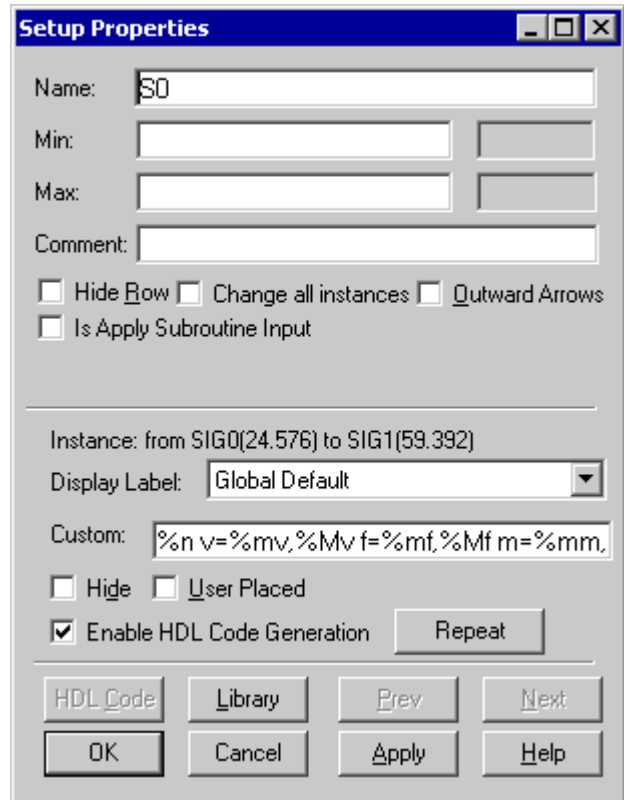
2.2 Delays

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

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

Once a delay is activated, then the delay process will wait for the amount of time (or clock cycles) specified in the min or max value of the parameter, and then drive the second edge. For more information, see *Section 1.10 Transaction Architecture*.

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

- Click the **Delay** button on the *Signal Button Bar*.

- Click on a transition to select it. This transition is the forcing transition.

- Right-click on the second transition to add a delay between the first and second transitions. This transition is the transition that will be delayed.
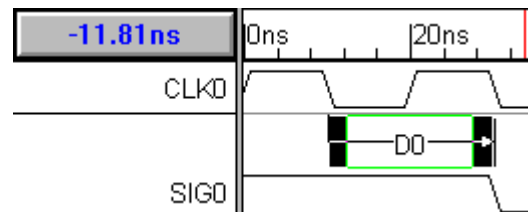
- Double-click on the delay to open the *Delay Properties* dialog. Most of the controls in the *Delay Properties* dialog were covered in *Section 2.1 Adding and Editing Parameters*.

The following controls are specific to delays:

- **Count Clock Edges** determines if the Min and Max settings are time or cycle based values. If the delay is *Unclocked* then the values are time. If a clock is specified then the values are numbers of clock cycles.
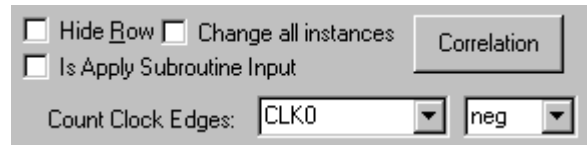
- **Min** and **Max** set the minimum and maximum time or number of clock cycles to be used for the delay. At simulation time only one value min, max, or typical (average of min & max) will be used. In TestBencher, the *Diagram Settings* dialog (discussed in *Section 3.7: Diagram Settings Overview* in the TestBencher manual), has the settings that determine which value will be used during simulation. If only one of the two settings has been given a value (min or max), the other setting will internally be given the same value.

- **HDL** code button opens the *Boolean Condition for Delay* dialog, that stores the condition that is checked before the delay drives the second edge. By default the condition is TRUE. You can type in the text for a new condition in the generated language. The condition can be any equation that evaluates to a TRUE or FALSE at simulation time. If the condition is not true after the triggering edge is detected, then the second edge will not be driven. The condition must be written in the generated language of the transaction. Note: If the condition is based on state values that occur during simulation, a graphical conditional delay can be constructed by triggering the delay from a sample parameter (see *Section 3.4: Samples Triggering a Delayed Transition or Another Delay*).
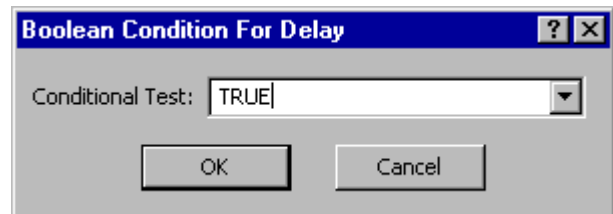
2.3 Resolving Multiple Delays

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

To open the *Edge Properties* dialog:

- Double-click on the edge to open the *Edge Properties* dialog.

- In the **Multiple Delay Resolution** section, choose one of *Transition Settings*:

  - **Earliest Transitions** uses the delay that will place the edge as early in the diagram as possible.

  - **Latest Transitions** uses the delay that will place the edge as late in the diagram as possible.

  - **Max Uncertainty** and **Min Uncertainty** are not currently supported for TestBencher Code generation.
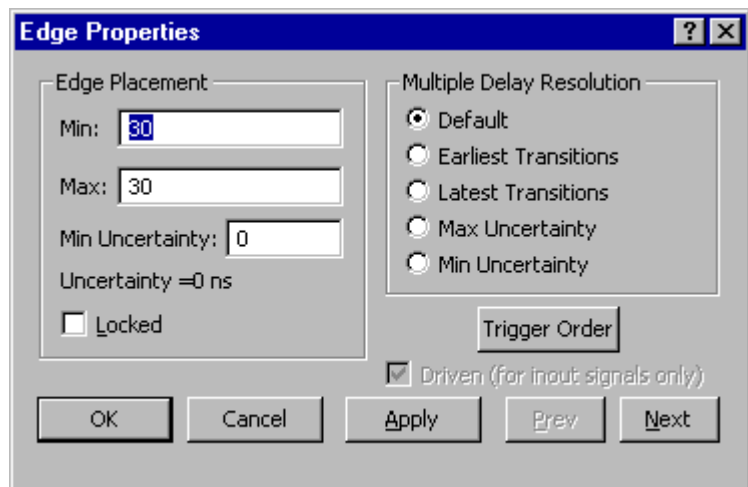
2.4 Setups and Holds

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

To create a setup or hold:

> - Click the **Setup** or **Hold** button.
>
> - Click on a transition to select it. This is the transition that will be monitored.
>
> - Right-click on a second transition to add a setup or hold between the first and second transitions. This is the control signal.
>
> - Double-click on the setup or hold to open the *Parameter Properties* dialog. Most of the controls in the *Parameter Properties* dialog were covered in *Section 2.1 Adding and Editing Parameters*.

The following controls are specific to setups and holds:

> - The **Min** field sets the minimum time that the data transition can occur before a setup or after a hold on the control signal.
>
> - The **Max** field sets the maximum time that the data transition can occur before or after the control transition. This field is optional and usually not specified for setups and holds.
>
> If a time is specified for the **Max** field, then the data transition must occur between the **Min** and **Max** times.
>
> - The **Outward Arrows** checkbox changes the direction that the arrows on the parameter are drawn. This does not affect code generation but it is a popular graphical feature.

Chapter 3: Samples

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

Monitoring Time

Samples that monitor a signal at a specific time are called ***Point Samples***. And samples that monitor a signal over an interval of time are called ***Window Samples***. Window samples are useful for testing that the value of a given signal does not change over a specified time frame, or for verifying that the signal goes through a specified sequence of states. Window samples draw themselves with a box indicating the monitoring interval. If you need to sample over a large window and you do not want to display it graphically then you can use the *Multiplier* control in the *Code Generation Options* dialog described in *Section 3.2 Sample Condition and Actions*.
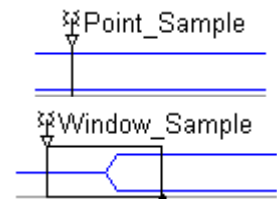
Triggering Process

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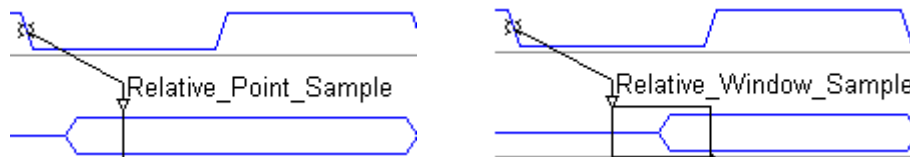
Check for Condition and Trigger an Action

The Sample's *Code Generation Options* dialog is used to define the condition the sample checks for and the actions it performs on the success and failure of the condition. *Section 3.3: Interpreting Sample Conditions and Blocking Points* describes how to control how the sample's condition is tested.

Sample Variables and Files

Samples generate several diagram-level variables that can be accessed by other graphical elements in the diagram (*Section 3.5*). Sampled values can also be written out to a file (*Section 3.6*).

3.1 Adding a New Sample

To create a sample you will define the triggering event by how you draw the sample. The monitoring time or interval will be set using the *Samples Properties* dialog. The sample actions to take if it succeeds or fails will be set using the *Code Generation Options* dialog discussed in *Section 3.2*.
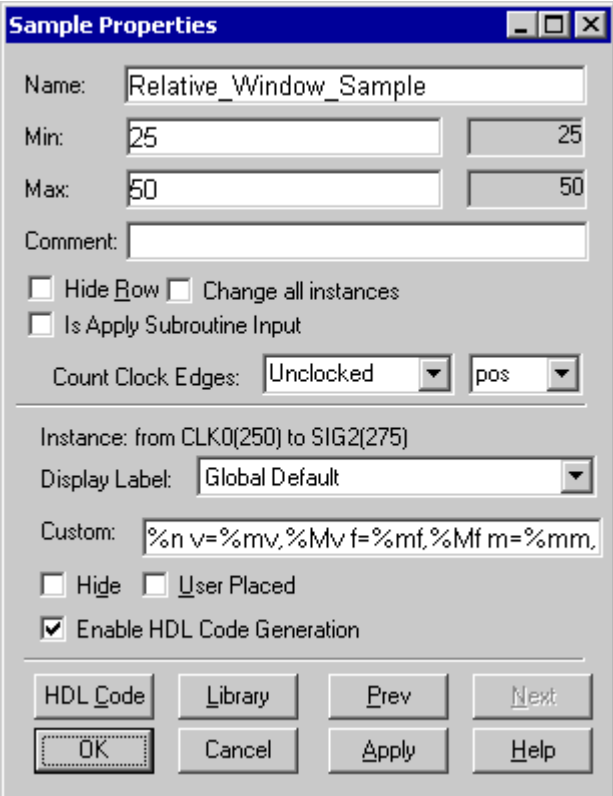
To add a new sample:

- Click the **Sample** button [Sample] on the *Signal Button Bar*.

- If you want the sample to be relative, then click the edge that you want the sample to be relative to.

- Right-click on the signal to be sampled. This will add the sample to the timing diagram. The exact time at which the sample is placed can be changed using the *Samples Properties* dialog discussed in the next step.

To edit the monitoring time and properties of the sample:

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

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

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

- The **Min** and **Max** edit boxes are used to specify the beginning and ending times or clock cycles for a sample window.

- Checking the **Is Apply Subroutine Input**, for TestBencher, generates input ports to the timing transaction that can be used to specify the values to use for the min and max settings of the sample. (*Section 3.2* describes how the monitored value can be made to be an output port of the transaction.)

- Samples can be cycle-based instead of time-based. The **Count Clock Edges** settings allow a clocking signal and edge to be specified for the sample.

- The **Enable HDL Code Generation** checkbox must be checked for any code to be generated for the sample.

- The **HDL Code** button opens the *Code Generation Options* dialog that defines the actions of the sample. This is covered in *Section 3.2: Sample Condition and Actions*.

3.2 Sample Condition and Actions

When a sample is triggered, the sample will test for a condition and then perform an action based on the success or failure of the condition. Both the condition and the actions can be changed using the *Code Generation Options* dialog. You can choose from several predefined conditions and actions or directly enter the HDL code. The user defined condition and action usually call class methods that have been defined for the transaction (*Section 3.3: Transaction Level Variables* in the TestBencher manual) or are short HDL expressions that make use of the internally generated sample variables (*Section 3.5: Using Sample Variables*).

To define the condition and actions of a sample:

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

   - Click the **HDL Code** button in the lower left-hand side of the dialog to open the *Code Generation Options* dialog.

   - The **If Condition** drop down list box controls what the sample checks for. Select one of the following conditions:

      - *Sample state matches:* If the monitored value matches the expected value the *Then Action* will be taken otherwise the *Else Action* will occur.

      - *Sample State doesn't match:* If the monitored value matches the expected value the *Else Action* will be taken otherwise the *Then Action* will occur.

      - *User-defined condition:* Directly enter the HDL code to execute see *Section 3.5: Using Sample Variables*

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

      - *Do nothing:* take no action if this branch is executed.

      - *Display Message:* Display a message in the simulation log using the severity level defined by the radio buttons below the action.

      - *Restart Diagram:* Resets and restarts the transaction execution.

      - *End Diagram (set status to Done):* Ends execution of this particular transaction. The bus-functional model will continue to execute as if this transaction had normally ended.

      - *Pause Simulation (Verilog only):* Stops the entire simulation.

      - *Do Delayed Transition:* Creates a delayed state transition (see *Section 3.4: Samples Trigger Delayed Transition or Another Sample*) that triggers based on the results of the *If Condition*.

      - *Trigger Sample:* creates a triggered sample (see *Section 3.4: Samples Trigger Delayed Transition or Another Sample*) that will fire based upon the results of the *If Condition*.

   - *Break Loop*: stops the loop that immediately surrounds the sample.

   - *Continue Loop*: returns to the beginning of the loop immediately surrounding the sample, skipping the last part of the loop

   - *User-defined action (enter below):* lets the user directly enter VHDL or Verilog code for the action into the edit box below the action drop-down list box. See *Section 3.5: Using Sample Variables* for more information.

- When the **Full Expect** checkbox is checked, every transition that occurs within the window range is checked. If the option is not enabled, the sample will check for the condition to be true (*Simple Expect*), or the event to occur (*Restricted Expect*) depending on how the waveform is drawn. See *Section 3.3: Interpreting Sample Conditions and Blocking Points* for more information.

- The **Multiplier** property extends the window of time for a sample. The difference between the **min** and **max** values will be multiplied by the value of the multiplier to determine the length of the sample. This also provides a method to indirectly specify a timeout for a sample. Since this method of extending the sample window does not appear graphically it can be used for very large windows that would not be very pretty to look at.

- In TestBencher, the **Enable Variable** control enables the sampled value to be output to a file in a spreadsheet-like format or stored in a variable. Set to *Then* if you want the data to be stored if the sample condition succeeds or *Else* if you want the data to be stored if the sample fails. Select *Always* if you want the data to be stored regardless of the condition. See *Section 3.6 Storing Sample Values in User Defined Variables* for more information.

- In TestBencher, the **Store Sampled Value As Subroutine Output** checkbox creates an output port to the transaction and when the transaction terminates it passes the sampled value out to the port. How this is implemented depends on the generation language:

  - In Verilog, TestBencher will automatically create a variable in the top-level project that the transaction is stored. The variable is named ***transactionName_sampleName*** and at the end of the transaction the sample value will be passed out to this variable.

  - In all the other languages, you must create a variable in the calling project that has the same type as the signal that is being sampled. This variable is then passed into the transaction apply call. The variable will be set during the transaction execution.

- The **Blocking** option determines whether or not the triggering process or sequence of the sample will wait for the sample to complete before continuing execution. If the option is enabled, the triggering process will trigger the sample and then wait until the sample process is complete before continuing execution. Otherwise, the two processes will execute concurrently once the sample is triggered. Samples, by default, are non-blocking. *Section 3.3: Interpreting Sample Conditions and Blocking Points* discusses this feature.

3.3 Interpreting Sample Conditions and Blocking Points

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

Full Expect Samples

If the **Full Expect** box is checked then the sample will wait until the end of the sampling window to determine if all conditions where met. This is called a **Full Expect** sample. This type of sample will test every state transition drawn within the sample window. For instance, if a Full Expect sample has a condition of *Sample state matches*, it will test that every expected transition matches what is drawn in the sample window. The appropriate Full Expect sample action is executed at the end of the sample window. This is indicated visually by the dot at the end of the sampling window.
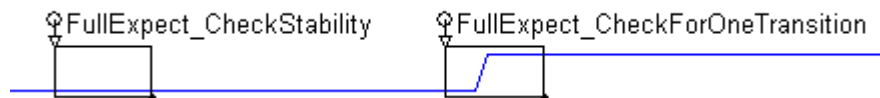
If the sample is clocked then the value of the waveform will be sampled at each clock edge in the sample window. For OpenVera, clocked Full Expect samples check for the last state in the drawn waveform.

Simple Expect and Restricted Expect Samples

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

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

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

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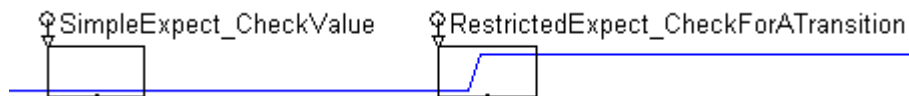
Blocking Sample

The **Blocking** setting in the *Code Generation Options* dialog controls whether or not a sample blocks other constucts in the same process. Samples with this setting enabled prevent other constructs from proceeding until the sample condition triggers an action. *Section 1.10: Transaction Architecture* discusses how blocking samples will pause a portion of the timing transaction. If a sample has **Blocking** enabled, then the clocking domain will pause until the *Then* or *Else* action is executed. Blocking samples are shown visually with a solid arrowhead. Non-blocking samples display with a hollow arrowhead.
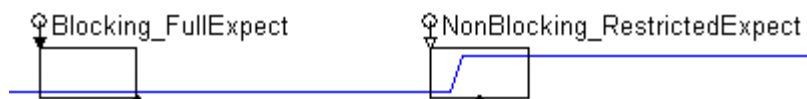
3.4 Samples Triggering a Delayed Transition or Another Sample

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

If a sample is triggering a delay, then that sample conditionally controls the signal transition. This is especially useful if several conditions must be met prior to a transition on a control signal. An alternate, non-graphical method for conditionally triggering transitions is discussed in *Section 2.2: Delays*.
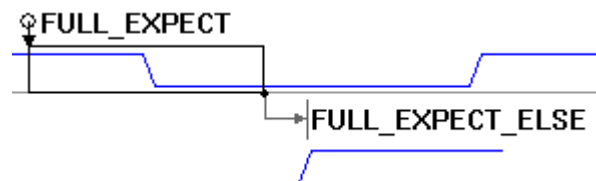
There are two different methods you can use to add a parameter to a sample. The recommended way is to add a parameter that is relative to a sample. This is the fastest way to add samples and delays to the *Then* and *Else* actions. The other method is to use the Sample's *Code Generation Options* dialog. Either method will set one of the sample actions to **Do Delayed Transition** or **Trigger Sample** and attach a graphical parameter to the sample.

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

- Click the **Delay** button `Delay` or the **Sample** button `Sample` in the *Diagram* window.

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

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

- For samples, right-click on the waveform you want to sample. This will open a context menu. Choose either **Then Triggered Sample** or **Else Triggered Sample** to create a chained sample.

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

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

- Click the **HDL Code** button to open the *Code Generation Options* dialog.

- For Delays, choose **Do Delayed Transition** from the **Then Action** or **Else Action** drop down list box.

- For Samples, choose **Trigger Sample** from the **Then Action** or **Else Action** drop down list box.

- Click the **OK** button to close the *Code Generation Options* dialog.

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

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

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

3.5 Using Sample Variables

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

In TestBencher, the sample value, *sampleName*, can also be exported from the transaction by checking the **Store Sampled Value as Subroutine Output** checkbox in the *Code Generation Options* dialog as described in *Section 3.2: Sample Condition and Actions*.

### Example of using Sample Flag Conditions

It is frequently desirable to define a sample condition in terms of previously executed samples. For example, you might wish to execute an action if two different previous samples were both true. This can be accomplished by writing HDL code accessing the flag variables that store information about previously executed samples. Assume you have a diagram with three samples (SAMPLE0, SAMPLE1, and SAMPLE2) where the first two samples test the values of two signals. To make SAMPLE2 true if both SAMPLE0 and SAMPLE1 are true, you would enter the *User-Defined Condition* of ***SAMPLE0_Flag and SAMPLE1_Flag***.

Example of using Sample Values in the Diagram

You can also use the sample values to build user-defined conditions for samples. For example, to test that the value sampled by SAMPLE0 is equal to the value sampled by SAMPLE1, enter the following *User-Defined Condition* for SAMPLE2.
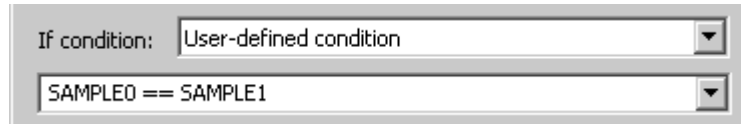
Note: the types of the signals sampled by SAMPLE0 and SAMPLE1 must be the same, or you will get a type mismatch error when you compile your test bench.

3.6 Storing Sample Values in User Defined Variables

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

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

To store a sample value in a user defined variable:

- Open the *Sample Properties* dialog by double-clicking the name of the sample.

- Click the **HDL Code** button to open the *Code Generation Options* dialog.

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

- Click the **Select Variable** button to open the *Select Variable* dialog.

- Click a field name or the variable name in the **Name** column of the selection tree to select a variable. Note that default *Index*, *MSB*, and *LSB* values are defined.
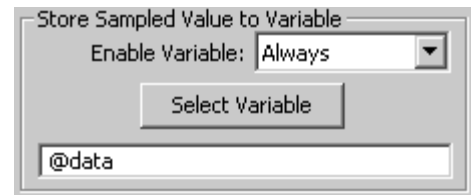
  Note: For any given transaction, only one sample can output to a specific column in the file. If more than one sample is using the same field name within the same timing diagram, only the last instance to occur during simulation will output to the column.

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

  - Double-click the text that needs to be edited.

  - Edit the text

- Click the **Insert Into Equation** button to set the variabel property for the sample.

- Click **Close** to close the dialog. This will set the variable property for the sample.

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

Chapter 4: Markers

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

Markers are triggered either by the unclocked process or by the clocked process of edge they are relative to. Loop and Wait Until markers act on their triggering process so it is important when using these types of markers to setup the triggering event correctly (*Section 1.10 Transaction Architecture*).

4.1 Adding a Marker to a Diagram

As with samples, markers can be absolute or relative. An absolute marker is attached to a specific time, while a relative marker is attached to a specific edge. Relative markers will be triggered by the process associated with the clocking domain (See *Section 1.10 Transaction Architecture*). Double clicking the marker opens the *Edit Time Marker* dialog that is used to control the code generation options for the marker.

To place a marker in a diagram:

- Click the **Maker** button `Marker` on the *Signal Button Bar*.

- If you want the marker to be relative, then select the edge that you want the marker to be relative to.

- Right-click in the *Diagram* window to place the marker. This will add a documentation marker to the diagram window.

To Edit a Marker:

- Double-click on the marker line or on the marker name to open the *Edit Time Marker* dialog.

- The **Marker Type** controls the function of the markers. The rest of the chapter is devoted to the details of each of the marker types:

  - *End Diagram* causes the transaction to terminate at that point.

  - *Pause Simulation (Verilog only)* stops the entire simulation.

  - *While Loop, For Loop, Repeat Loop, Loop End*, and *Exit Loop When* are used to create loops for a single process in the transaction.

  - *HDL Code* marker inserts user written source code.

  - *Wait Until* causes the process that triggers the marker to block until the condition becomes true.

  - *Semaphore* used to define critical regions in a transaction. This Marker type is used by TestBench only.

  - *Pipeline Boundary* is used to specify a pipeline region in a transactor. This is used when multiple instances of a trans- actor are running in parallel. This Marker type is used by TestBench only.

  - *Documentation* markers are used to annotate the timing diagram.

  - *Time Break Markers* are used to hide sections of the timing diagram but do not cause code to be generated.

- The **Attach to time** controls are used to change the attachment or placement of a marker.
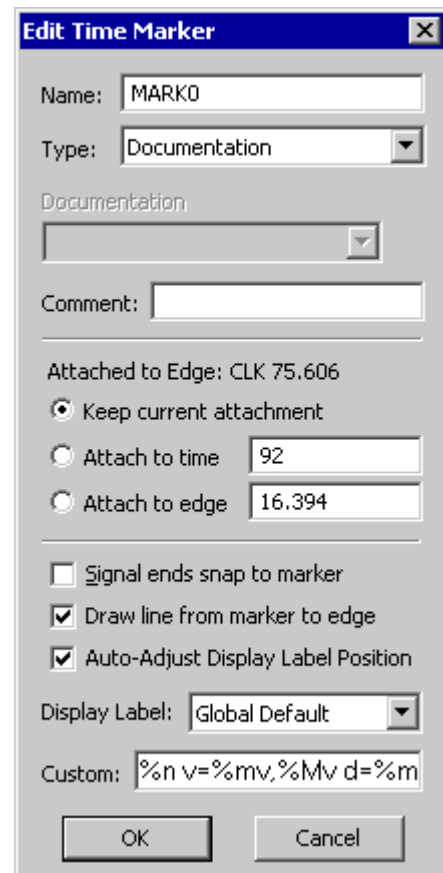
- To move a relative marker to the exact edge time, type **0** into the **Attach to edge** edit box.

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

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

- The **Snap Signal Ends to Marker** feature is generally for documentation purposes - the ends of all drawn waveforms will be attached to the marker and move with it.

- **Draw Line From Marker To Edge** will cause a dotted line to be drawn for markers that are attached to an edge. This is a nice feature to be able to quickly see that the marker is attached to an edge and not a time, and also which edge the marker is attached to.

- **Auto Adjust Display Label Position** allows the diagram editor to automatically adjust the position of the marker display to ensure that it does not over-write or get overwritten by other items in the diagram window.

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

4.2 End Diagram Markers

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

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

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

- Add a marker and then double-click on the marker to open the *Edit Time Marker* dialog.

- Select **End Diagram** from the *Marker Type* drop down list.

- If the Marker is not located at the exact location or attachment that is needed, then use the **Attach to** radio buttons to move the marker. In this example the edge is attached to the CLK0 edge at exactly time 250ns.

- (OPTIONAL) Choosing **Type** from the **Display Label** control causes the marker to display the words *End Diagram* instead of the marker name.

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

4.3 Loop Markers

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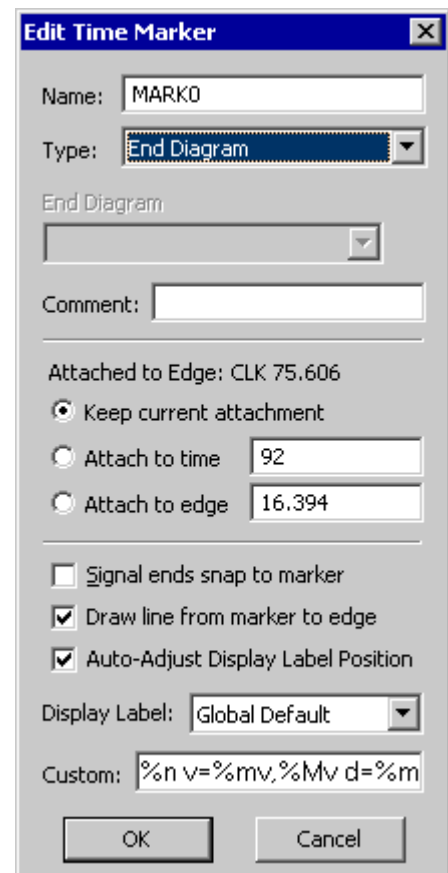
was met. TestBencher supports while loops, for loops, and repeat loops. The *Exit Loop When* marker can be used to terminate a loop in the middle of a cycle. Loop Markers can also be used with samples whose *Break Loop* and *Continue Loop* actions affect the operation of the loop.

The same process must trigger both the beginning loop marker and the end loop marker (see *Section 1.10 Transaction Architecture*). For clocked transactions, this means that the begin marker and the end marker need to be attached to the same edge type of a given signal. When TestBencher recognizes the beginning and ending of a loop it will draw a green loop line between the markers. In the example below the bad loop will not work because the while marker is triggered by the rising edge clk process while the loop end is triggered by the unclocked process.

Often the signal edges that trigger the beginning and end of loop markers are also triggering other markers and samples. When several graphical elements are triggered off of the same edge then the order determines whether the other graphical elements occur inside or outside of the loop. The order is set by double-clicking on the edge and using the *Edge Properties* dialog (see *Section 1.8: Controlling the Triggering Order of Parameters*).
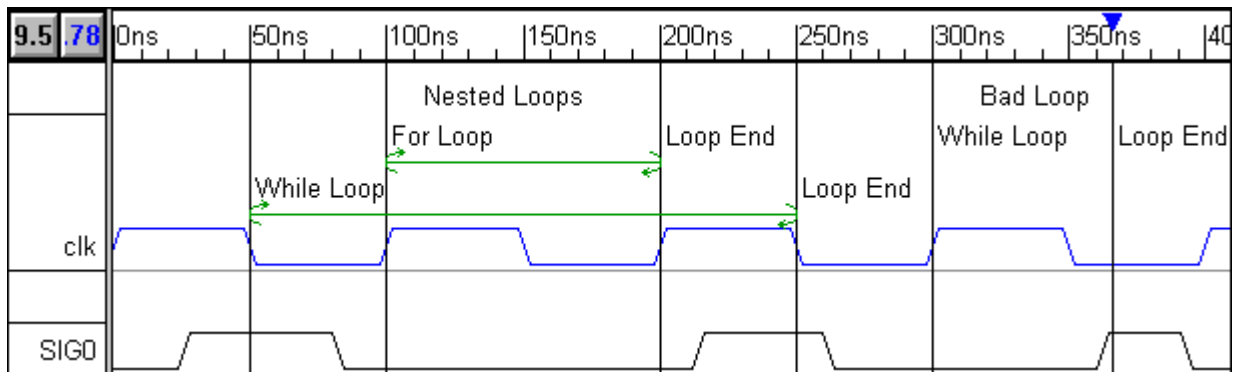
To add a loop to a timing diagram:

- Add two markers to the timing diagram. Both should be relative to the same signal and edge type, or both should be absolute time markers.

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

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

   - **For loop** marker will execute for a specified number of iterations. The *Index* variable will be automatically created. Each loop the index variable will be incremented by the *Inc* number. The loop will end when the index becomes greater than the *End* number.

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

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

- Double-click on the marker on the right to open the *Edit Time Marker* dialog. Choose the **Loop End** marker type.

- Click the **OK** button to close the dialog. If the markers are triggered by the same process, TestBencher will draw a loop line between the markers. If there is no loop line then check the attachments of each marker.

Exit Loop When

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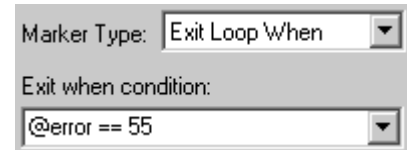
4.4 HDL Code Markers

HDL code markers are used to make calculations and execute code that is not represented graphically. HDL code markers have a limited amount of space for typing, so it is usually just used to type in the name of a function to call. The code box accepts direct HDL code in the transaction generation language. You can make calls to class methods (*Section 5.2: Class Methods*), library subroutines (*2.7 Libraries and Use Clauses* in the TestBencher manual), or insert any code that is valid within the context of a process (VHDL), always block (Verilog), TCM (e), Task (OpenVera), or method (TestBuilder).

To add an HDL Code marker:

- Add a marker to the diagram and double-click on the marker to open the *Edit Time Marker* dialog.

- From the **Marker Type** drop-down list, choose **HDL Code**.

- Type in the source code into the **HDL Code** edit box.

   OR

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

- Click **OK** to close the dialog.
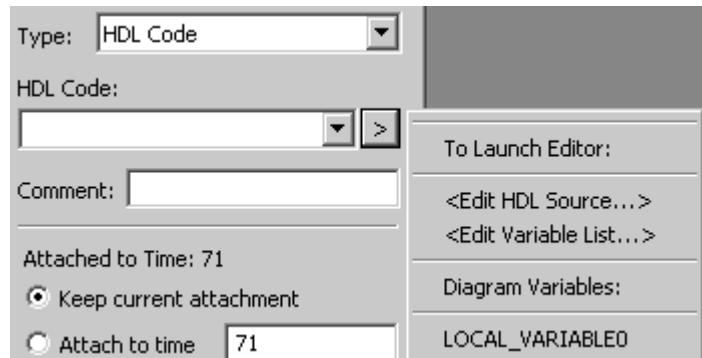
4.5 Wait Until Marker

Wait Until markers provide a mechanism for indefinitely pausing the execution of one clocked process within a transaction. This type of marker pauses the transaction until its condition becomes true. Blocking samples also pause the execution of a process, but they have a time out built into the window and multiplier settings. Wait Until markers will not time out. The process that gets paused will be the triggering process of the Marker (see *Section 1.10: Transaction Architecture*).

To specify a **Wait Until** condition:

- Add a marker that is attached to some signal transition in the diagram.

- Double-click on the marker to open the *Edit Time Marker* dialog.

- From the *Marker Type* drop-down list, choose **Wait Until**. This relative marker it will pause the execution of all signals and graphical elements that are relative to the same signal and edge type.

- In the **Wait Until Condition** edit box, enter a condition. The condition can be any equation in the generation language that evaluates to a TRUE or FALSE at simulation time.

- Click **OK** to close the *Edit Time Marker* dialog.
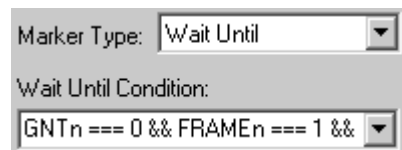
When this transaction is applied, it will now pause execution (of the transaction, not the simulation) at the time that the marker is placed until the specified condition has occurred.

4.6 Pause Simulation Marker (Verilog Only)

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

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

To specify a **Pause Simulation** marker:

- Add a marker to the timing diagram. The exact placement or attachment does not matter because the marker will pause all processes in the entire model.

- Double-click on the marker to will open the *Edit Time Marker* dialog.

- From the *Marker Type* drop-down list, choose **Pause Simulation (Verilog only)**.

- Click **OK** to close the *Edit Time Marker* dialog.

4.7 Documentation and Time Break Markers

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

In addition to the graphical elements of the timing diagram, the Reactive Test Bench Generation option also supports the generation of user-defined variables and class methods. These elements let you manipulate data from the model under test and compose algorithmic functions that are not ease to define graphically.

Chapter 5: Variables and Class Methods

In addition to the graphical elements of the timing diagram, the Reactive Test Bench Generation option also supports the generation of user-defined variables and class methods. These elements let you manipulate data from the model under test and compose algorithmic functions that are not ease to define graphically.

5.1 Variables

Variables are used to store data that can be set and accessed during simulation. Variables can be used anywhere in a diagram that you type in HDL code including: Sample values, Loop control variables, HDL code markers, and HDL states. The data type of a variable can be any of the generation language's native data types.

Each variable has various properties including the data type and the structure. The data type can be any of the generation languages native data types. Variables can either be individual instances or arrays of variables.

To Create Variable:

- Press the **View Variables** button, in the *Diagram* window, to open the *Variables List* dialog.

- Press the **New Variable** button near the bottom of the dialog. You can also just click the blank variable line and start typing a new variable. Either method creates a new variable with default properties in the Variables list box.

- Double-click on the column for any property that needs to be edited. This will cause either an edit box or a drop-down list to be opened that can be used to set the field property. The variable properties are defined below.

- When you are done click **OK** to close the dialog.

Variable Column Properties

Most of the column properties for variables do not affect code generation for the Reactive Test Bench Generation Option and are used only by TestBencher's bus-functional model generator. For the Reactive Test Bench Generation feature the following are the column properties that you must set for a variable:

Variable Name: used when referencing the variable or field of the class.

Size: specifies the number of elements in a complex structure type field. This setting is available for arrays. Because the default structure type is an element, the default for this property is 1.

BitSize, MSB, and LSB: are used to determine the bit size for field elements. Depending on the language being used, the bitsize may be specified using an LSB and an MSB. Note that some types, such as a string, may not use a bitsize, and that others, like bool, may have a limited bitsize.

Data Type: determines the type of the elements of the field. The possible settings for this property are the available Language Independent Types (such as bool, 2_state, or 2_state_vector) and may also include other classes that have been defined in Class Libraries that are included in the project. The default for this property is int. If a language independent type is selected it will be converted to the appropriate type for the generated language in the generated test bench. *5.3: Language Independent Types* provides more information about these types as well as a chart showing the conversion values from the language independent type to the generated language types.

Structure Types: Either an *Element* or an *Array*. An element is a single data item (like a single integer). And array is a series of elements of the same data type.

Initial Value: allows an initialization value to be specified. The variable will be initialized on creation during simulation. The string entered in this field will be placed directly in the generated code without modification.

5.2 Class Methods

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

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

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

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

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

- Open the *Class Methods* dialog from the object that you want to define the class method for:

    - For diagram-level methods, click the **Class Methods** button located in the *Diagram* window.

    - For class-level methods, click the **Class Methods** button located *Classes and Variables* dialog for the selected class.

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

- Create a new class method.

    Select the Language that the method is to be defiend for from the **Language** drop-list.

    - Click the **New Method** button to create a new method with default values.

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

        - Method Name: the name of the method.

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

	Method Name	Method Type	Return Type	BitSize
0	SetHEC	Task	void	
1	SetCRC8	Task	void	
2	ComputeHEC	Function	integer	32
3	ComputeCRC8	Function	integer	32
4	CompareCell	Function	integer	32
5	BriefReport	Function	integer	32
6	Display	Task	void	

Functions and Tasks:

New Method   Delete Method
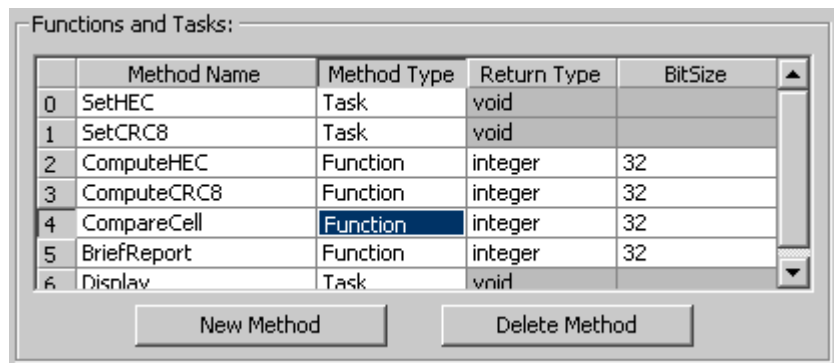
turn any value. *Functions* perform an operation on the parameters that they are passed and return a value.

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

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

- Add parameters to the class method:

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

Method Parameters:

	Method Name	Type	BitSize
0	cell_to_compare	Invalid	0

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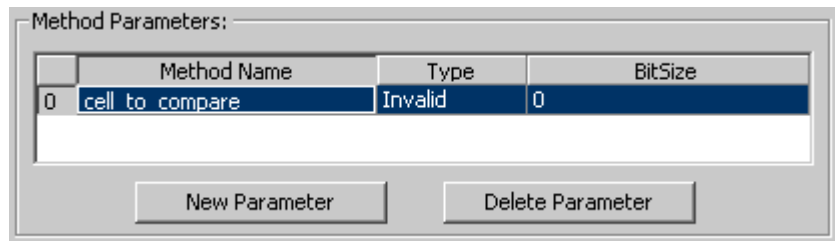
- Click the **New Parameter** button to create a new parameter with default properties.

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

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

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

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

- Add the class method source code:

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

- Type your source code into the *Source Code* edit box.
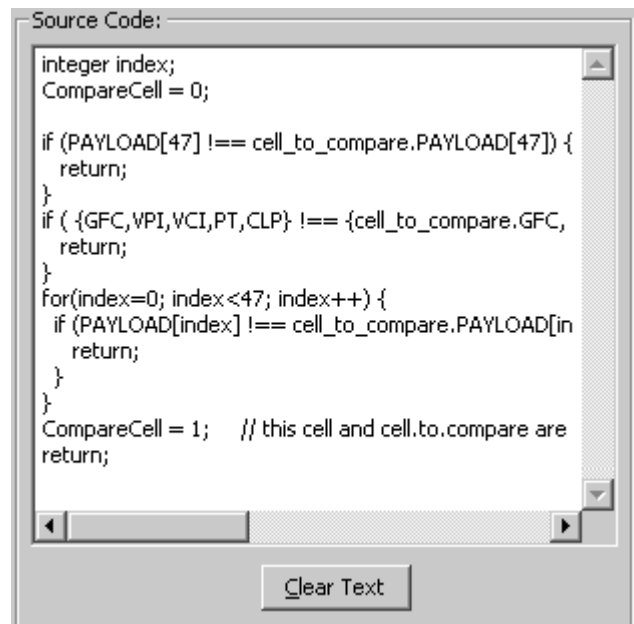
5.3 Language Independent Types

SynaptiCAD has defined a set of language independent types that is used in generated test benches in place of the native types for a given language. This is done to facilitate the development of language independent class definitions and variables. During test bench generation the language independent type is converted to the appropriate native type for the language being generated. Note that not all of the language independent types are supported by all of the generation languages. The dialogs that allow selection of these types, such as the Class Definitions & Variables dialog, will only display

Source Code:

```
integer index;
CompareCell = 0;

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

Clear Text

the language independent types that are supported for at least one of the currently licensed languages. Additionally, these dialogs support a view that will display only the items that are available for the currently selected language.

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

Syncad Type	BitSize	Description/Values
bool	1	Truth values (1 or 0)
2_state	1	0, 1
2_state_vector	variable	0,1 in vector format
byte	8	Unsigned integer represented by 8 bits
int	32	Signed integer represented by 32 bits
unsigned_int	32	Unsigned integer represented by 32 bits
real	64	Floating poing numbers
fixed_len_string	variable	Series of characters enclosed by quotes
variable_len_string	n/a	Series of characters enclosed by quotes
time	64	Simulation time quantities
4_state	1	0, 1, X, Z
4_state_vector	variable	0, 1, X, Z in vector format
event	n/a	Synchronization item
std_logic	1	U, X, 0, 1, Z, W, L, H, -
std_logic_vector	variable	U, X, 0, 1, Z, W, L, H, - in vector format
std_ulogic	1	Unresolved version of std_logic
std_ulogic_vector	variable	Unresolved version of std_logic_vector
signed_logic	variable	Signed version of std_logic_vector
unsigned_logic	variable	Unsigned version of std_logic_vector

Type Conversion

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

Syncad Type	Verilog	VHDL	TestBuilder
bool	reg	boolean	bool
2_state	reg	bit	tbvSmartSignal2StateT
2_state_vector	reg	bit_vector	tbvSmartSignal2StateT
byte	reg	bit_vector	tbvSmartSignal2StateT
int	integer	integer	tbvSmartIntT

Syncad Type	Verilog	VHDL	TestBuilder
unsigned_int	integer	natural	tbvSmartUnsignedT
real	real	real	tbvSmartDoubleT
fixed_len_string	reg	string	char[]
variable_len_string			tbvSmartStringT
time	time	time	
4_state	reg	std_logic	tbvSmartSignal4StateT
4_state_vector	reg	std_logic_vector	tbvSmartSignal4StateT
event	event		
std_logic		std_logic	
std_logic_vector		std_logic_vector	
std_ulogic		std_ulogic	
std_ulogic_vector		std_ulogic_vector	
signed_logic		signed	
unsigned_logic		unsigned	

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

- Verilog reg type is a four state type.

- Verilog integer type is signed.

- VHDL natural is a limited version of the VHDL integer type, so it's max value is 2^{31}, not 2^{32}.

- Some languages do not provide an unsigned integer type.

Chapter 6: Test Bench Techniques

The Reactive Test Bench Generation feature can generate single timing diagram models that perform complex data generation and checking. Here we have gathered some of the techniques that we use to model different types of functionality.

6.1 Testing a Counter Model

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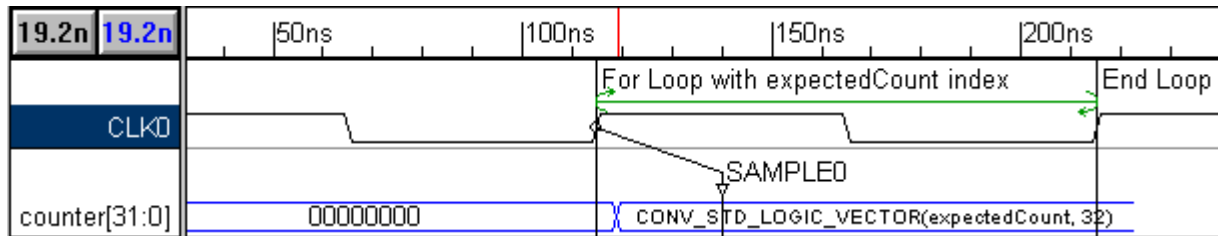
Discussion of the Counter Test Bench:

- **Initialize the counter:** the first cycle in the diagram is used to initialize the starting value of the counter.

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

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

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

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

6.2 Waiting for Signal Transitions

You can use either samples or markers to make a transaction wait for an event or series of events before continuing to execute. Below is a chart of the different methods of waiting and the recommended usage for each method. *Chapter 3: Samples* and *Chapter 4: Markers* have more information about Samples and Markers.

Wait On	How Long	Construct Used	Settings
One or more conditions (e.g., signal states)	Block until all conditions are true	Wait Until Marker	
One event or condition	Block until time out	Sample with Multiplier	*min == max* *multiplier > 1* check *Blocking* uncheck *Full Expect*

Wait On	How Long	Construct Used	Settings
One event or condition	Block until time out	Sample with window	*min != max* *multiplier == 1* check *Blocking* uncheck *Full Expect*
One event	Block indefinitely or until diagram times out	Sensitive Edge	
Several events or conditions across several clock cycles	Each sample may block with time out	Several samples chained together (first samples will block subsequent samples	check *Blocking* uncheck *Full Expect*

Below is an example of timing diagram that demonstrates these techniques for waiting.
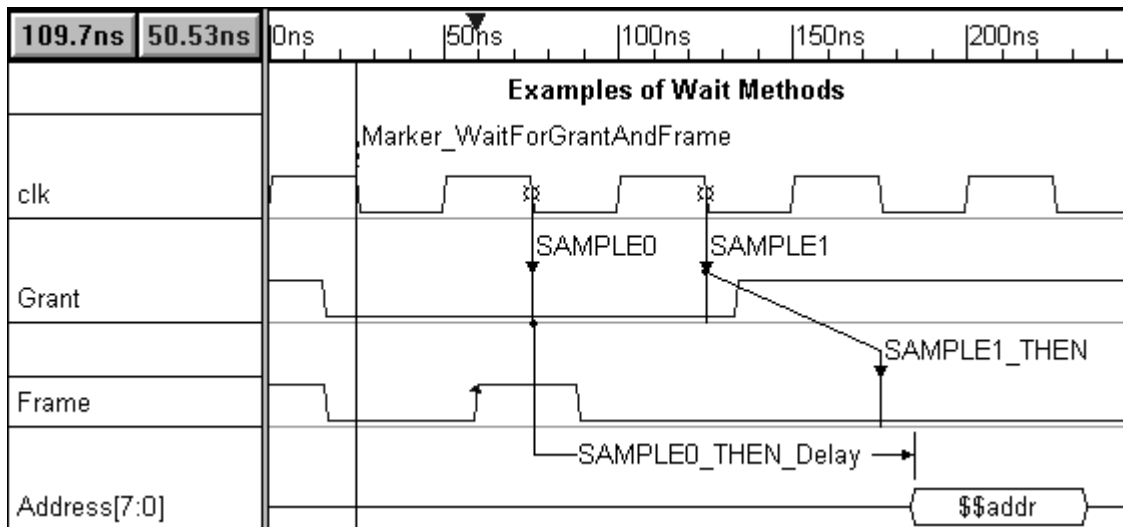
- The Marker called **Marker_WaitForGrantAndFrame** is a *Wait Until Marker* type with the condition of **(Grant===0 && Frame ===0)** {the condition code is in the generated language, in this example Verilog}. This marker will block the transaction until the condition becomes true.

- The rising edge on **Frame** is sensitive. This will cause the diagram to wait for that edge to occur.

- The Sample called **Sample0** is setup as *blocking* and *non-full expect* with a multiplier of 3. The Multiplier is the sample's time out. Checking the *blocking* box causes the sample to block the triggering clocked sequence until it times out or until the condition becomes true. Disabling the *Full Expect* box means that the sample will not expect the drawn condition to be true during the entire window. Instead it will continue sampling as long as the condition is NOT true and the time out has not been reached.

- This sample also has a conditional delay, **SAMPLE0_THEN_Delay**, so that when it passes it will cause the value passed into *$$addr* to be written out to the Address signal. If **Sample0** times out then the Address signal never gets driven.

- The samples **Sample1** and **Sample1_THEN** check for Grant and Frame to be true over successive clock cycles. They are defined using the same settings as Sample0 in the previous example except the multiplier is set to 1.

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

Index