ECE 551 ModelSim Tutorial

Updated Fall 2022

Dept of ECE, UW-Madison

In this tutorial, you will learn how to setup a ModelSim project, compile your Verilog files, correct compilation errors, and perform design debugging using ModelSim. The example design used within this tutorial is simple Synchronous Serial Port (SSP) that contains both a send and receive module. It has a simple 3-wire interface:

Port Name	Function
SerData	Bi-directional data line used for both sending and receiving
Recv_nTran	Input that indicates if we are receiving (1) or transmitting (0)
StartOp	Input that indicates that an operation should be performed

The design of this unit is broken into a number of separate modules:

Module Name	Function
ssp	Top-level module which instantiates all of the below sub-modules
receive	Contains the shift register and state machine for the receiver
transmit	Contains the shift register and state machine for the transmitter
busint	Contains the logic to control the three wire serial interface

The tutorial also contains testbenches for the receive, transmit, and ssp modules.

The ModelSim Tutorial must be run on a Linux workstation using your CAE account.

IMPORTANT NOTE

It is critical to remember that Verilog is NOT a software language. Verilog is used to describe hardware. While ModelSim may provide the ability to step through the code or insert breakpoints, this is NOT what actually happens when the hardware is operating. In reality, hardware is inherently parallel, with each transistor or gate continuously providing an output signal based upon its input signals.

For example, in software, if there is an **"if (flag) a = b&c else a = b|c"** statement, only one branch of the statement is actually executed. However, if HDL code has an "if-else" statement, hardware must be created for both branches if the value of "flag" isn't constant. When we synthesize to hardware, both an AND gate and an OR gate are created with "b" and "c" as inputs, and a multiplexer is created to choose the result based on the value of "flag", and the output of the multiplexer is sent to "a". Both gates are always present and operating continuously, regardless of the value of "flag". Stepping through the code, however, will make it appear as if only one of the branches is executed. To make the simulation more efficient, the simulator will only perform calculations and event scheduling when necessary. Since it knows the value of "flag", it will only schedule the event relating to the input of the multiplexer that is active.

1 Tutorial Setup

<u>Directory and File Setup</u>

In your root directory, create an ece551 directory:

%> mkdir ece551

Change directory to the ece551 directory:

%> cd ece551

Copy all the tutorial files to your current directory:

%> cp –r ~ejhoffman/ece551/public/modelsim   ./

Change directory to your tutorial directory:

%> cd modelsim/tutorial

<u>Start ModelSim</u>

%> vsim

When you start ModelSim for the first time, a pop-up box will appear (possibly after a short delay) as in Figure 1-1. You should check the **Don't Show…** box and then close the window
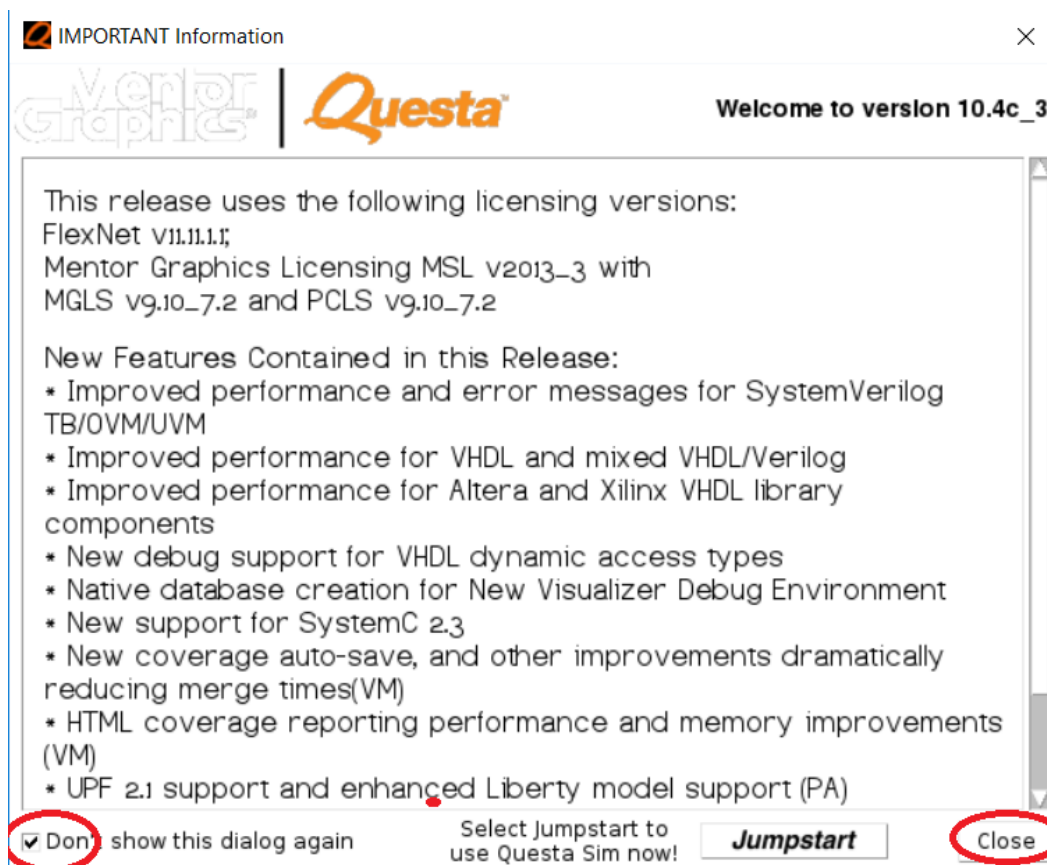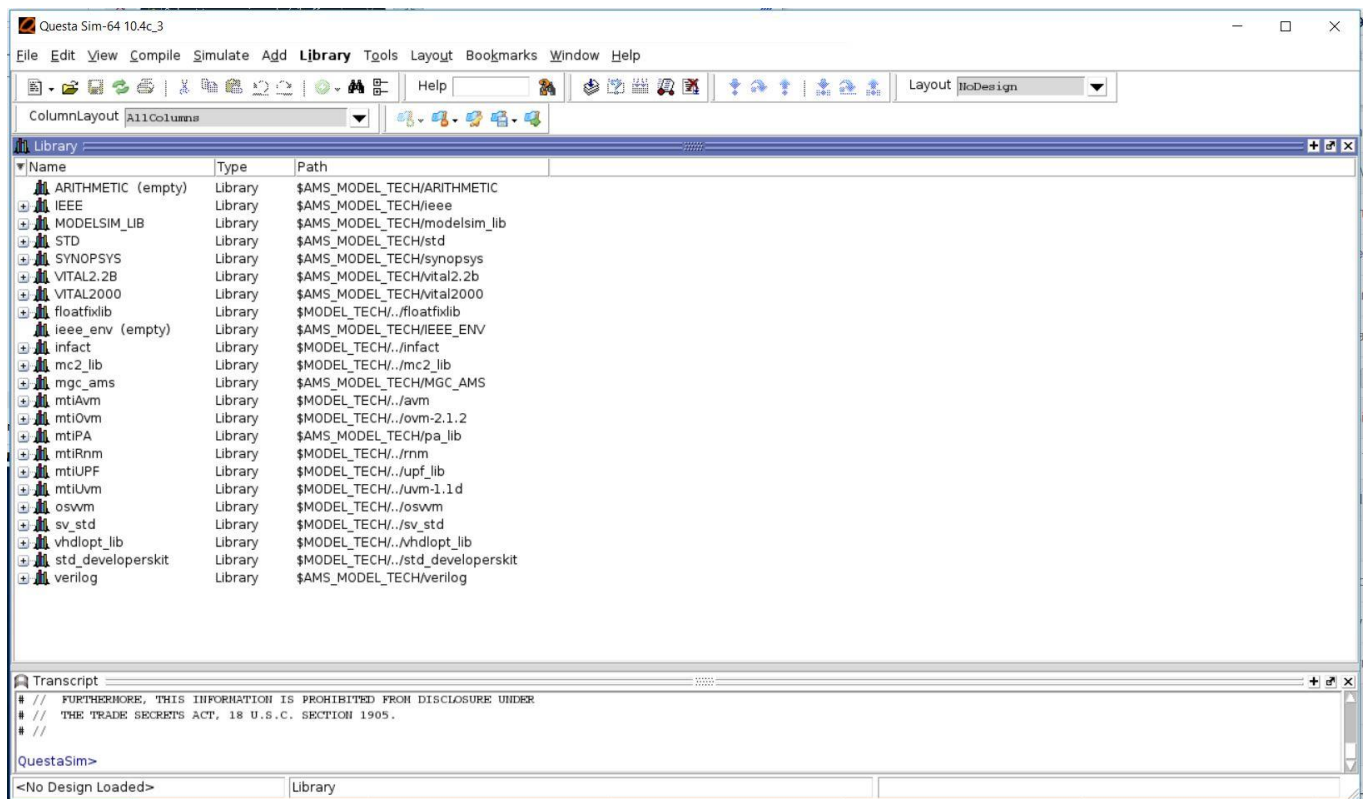

Figure 1-1: Important Information pop-up

**Figure 1-2: ModelSim (Questa) default window.**

Now that ModelSim is open, you should see a default window similar to that in Figure 1-2. There are a few things to note about the window. In the lower left hand corner, it says **<No Design Loaded>** which indicates that there is no current Verilog or VHDL simulation. There is a **Workspace** on the left hand side of the window that currently contains only the **Library** tab. On the bottom of the window is a command line area that can be used either to issue commands, or view the outputs of commands run through the GUI.

Many (although not all) operations performed through the menus will also echo into the command line or transcript area, so you can learn the command-line operation as you go. Knowing the command-line commands is important, for example, if you want to write a script to perform a set of simulations. Help for each command-line command is available by entering **help <command name>** on the command line.

2 Creating Projects

<u>Create a New Project</u>

File->New->Project
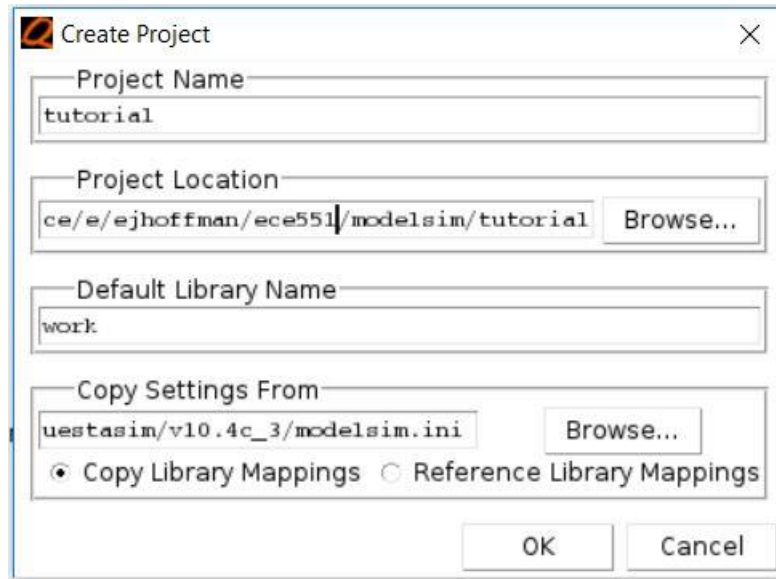
**Figure 2-1: Create Project dialog**

The dialog of Figure 2-1 will appear. If necessary, use the **Browse** button to make sure the **Project Location** is set to your tutorial directory. Name the project "tutorial" as in Figure 2-1 and click **OK**.

<u>Add Existing Files to a Project</u>

When you clicked **OK**, the window in Figure 2-2 appeared. Click **Add Existing File**. The dialog box as in Figure 2-3 appears.
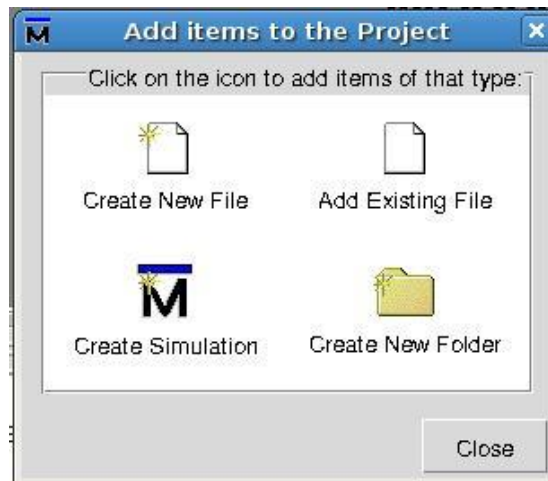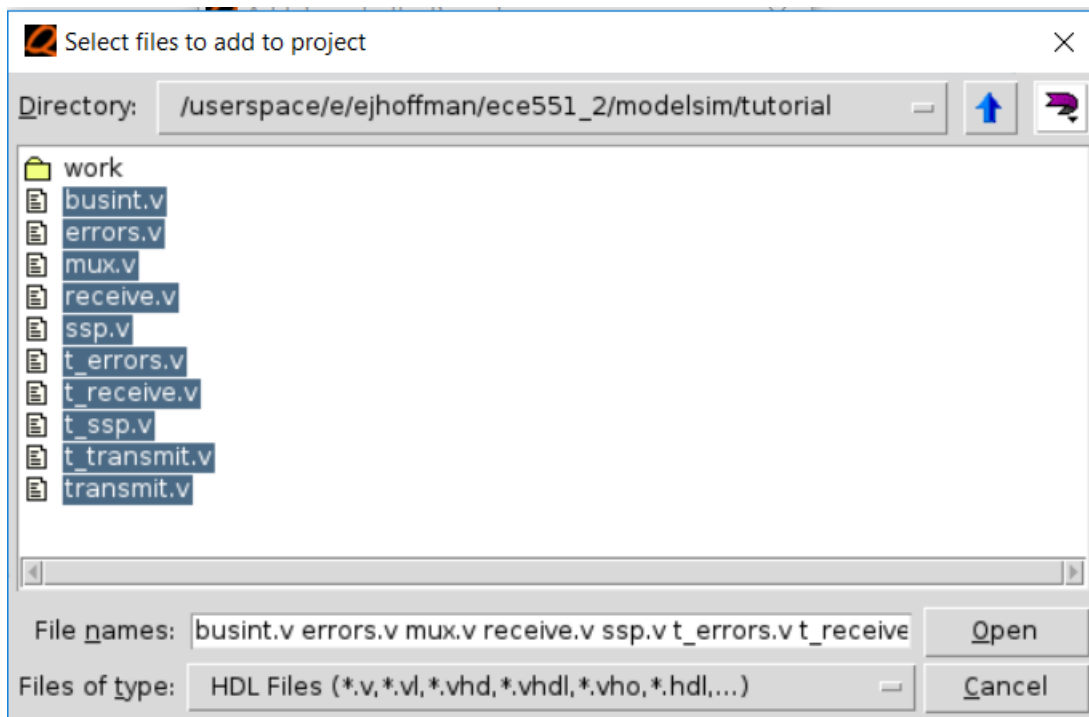

Figure 2-2: Add items to the Project window
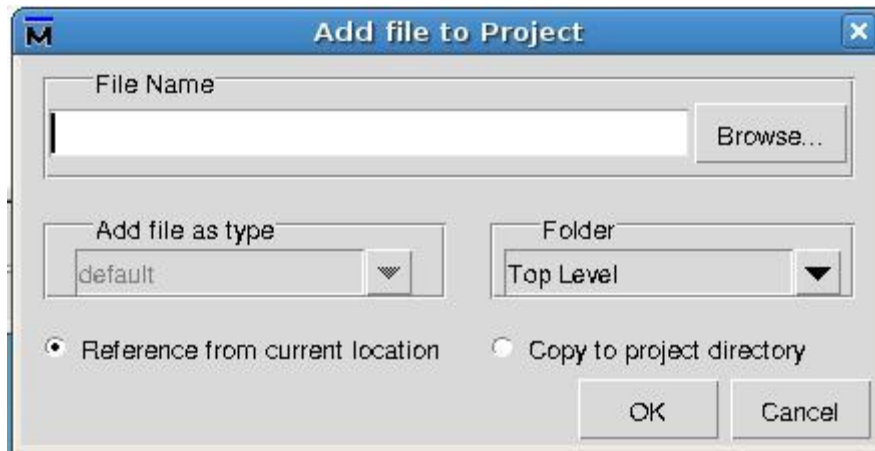
Figure 2-3: Add File dialog box

Click **Browse** and select all of the files as in Figure 2-4. Click **Open** and then **OK**. The files will be added to the project.


Figure 2-4: Select Files dialog box

Existing files can be added to a project at any time by right clicking within the workspace window and selecting **Add to Project -> Existing File.**

Creating a New File in a Project

There are four main ways of creating new design files within an existing project

1.) On the dialog of Figure 2-2, click **Create New File**. The dialog box of Figure 2-5 will appear.
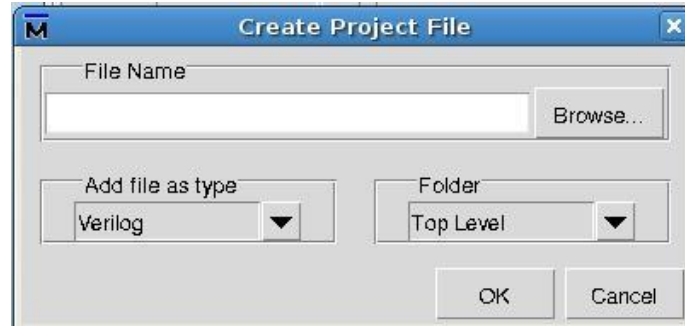
Figure 2-5: Create File dialog box

Change **Add file as type** to "**Verilog"** and type in the name of the new file. Click **OK** then **Close** to return to the main window. The new file will now appear in the workspace window of the project

2.) Click on **File->New->Source->Verilog**.

This will open a blank file within the main ModelSim window for you to edit and save as needed.

3.) Right Click in the "Project Tab" and select "Add to Project" ➜ "New File". This will bring up a a new file dialog similar to Figure 2.5 above.

4.) Create a new *.v file (or .sv is using system Verilog) using an external editor and add it by right clicking on the workspace and selecting **Add to Project -> Existing File**.
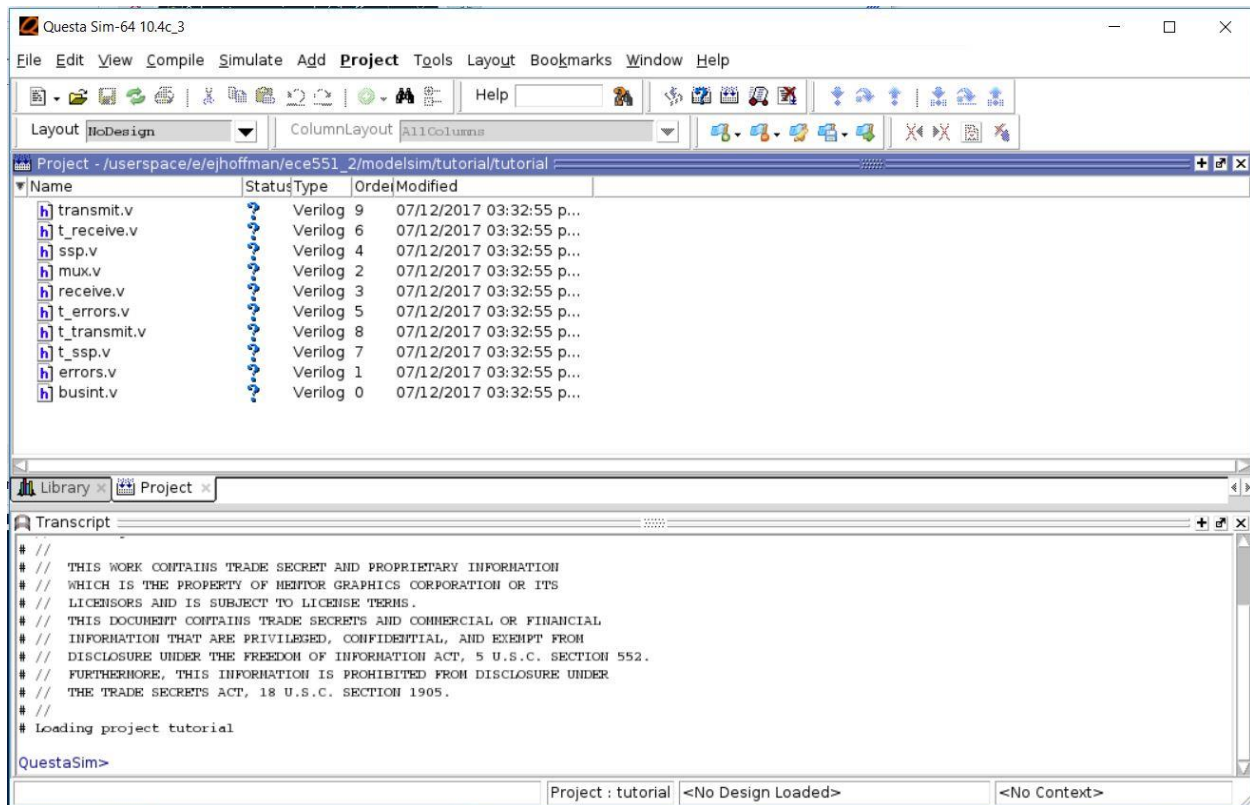
Figure 2-6: ModelSim main window, with project open

All of the tutorial files are now in the project, and a **Project** tab is now part of the workspace. The status of all of the files is "**?**", which indicates that there has not yet been an attempt to compile them. In other words, at this point it is not known whether the file can successfully compile or not.

Set Files as Do Not Compile (FYI)

Some files that are associated with the project are used only as include files, or are not intended to be compiled. To set a file as do not compile, right click on the file in the list and choose **Properties** from the context menu. A dialog appears as in Figure 2-7. Check the **Do Not Compile** box, and click **OK**. Note that there will now be no status icon for this file, which denotes that it will not be compiled. This is for your information only**; no files in this tutorial require this to be set**.
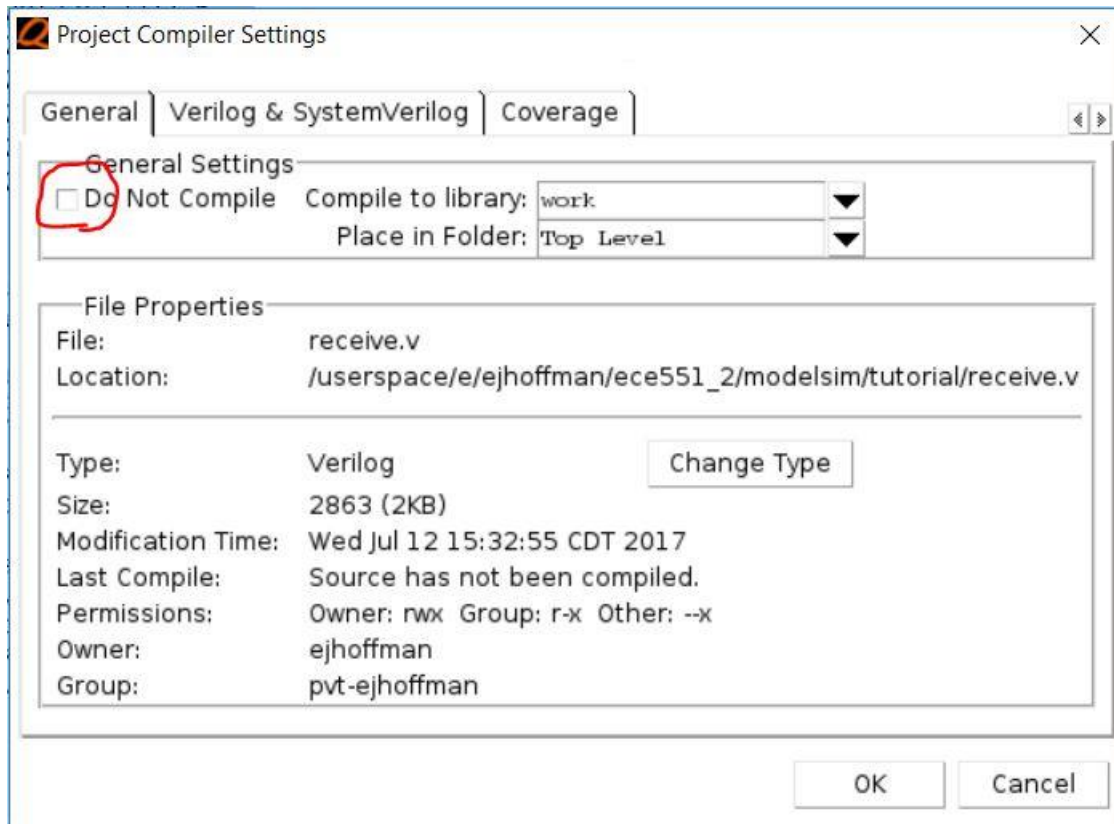
Figure 2-7: File Properties dialog

3 Code Entry and Compiling

<u>Compile All Files</u>

Compile->Compile All

All of the files <u>except</u> for errors.v should succeed. This file has been created to show you some common compilation errors. In the command line area of the main window, double click on the **Compile of errors.v …** failure message. A window will pop up showing the error (Figure 3-1).
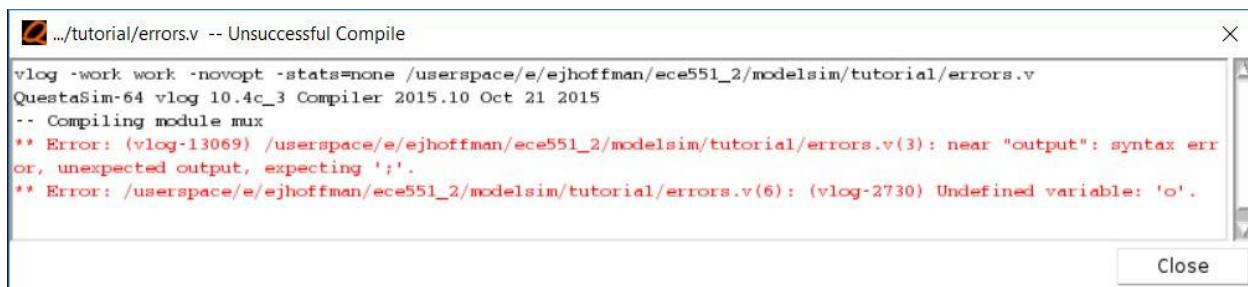

Figure 3-1: The Unsuccessful Compile window provides details on why the compilation failed

This error indicates that a semicolon is missing from one of the lines and is one of the most common errors in code entry. There are only two errors listed, even though there are more than two errors in the code. When there is a missing semicolon, the compiler will not attempt to finish compiling and thus will not report any other errors that exist. Note that the semicolon is missing from line 2, but line 3 is listed

instead because the compiler is only sure that the semicolon is missing once it finds the "output" keyword on the next line. Make sure to check nearby lines when fixing missing semicolons.

Editing the Offending File

Double click on "errors.v" in the project tab of the window. An **Edit** window opens containing the code for errors.v as shown in Figure 3-2 . This file describes a simple two input mux. To fix the error above, add a semi-colon on the end of line 2. When finished, save the file.
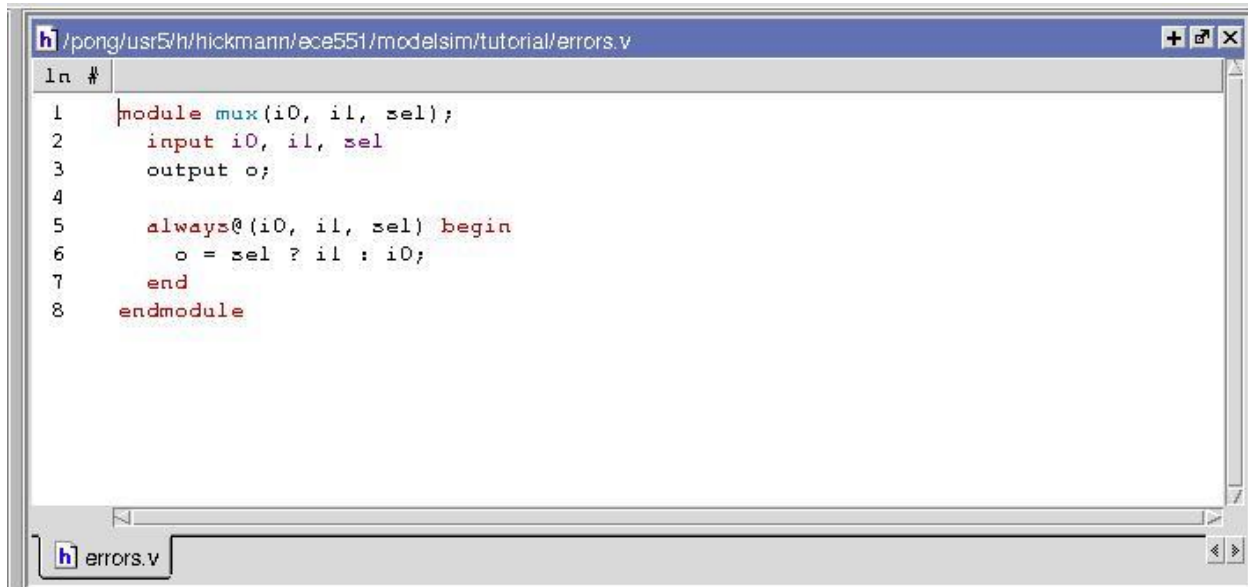
Figure 3-2: Edit window with errors.v code (with code errors)

Recompile Changed Files

Right Click->Compile->Compile Out of Date

This will recompile only the file which have changed since the last compile was performed. There is still an error in the compilation. Double click on the error again to see the new error messages. The errors should be as in Figure 3-3.

Figure 3-3: The Unsuccessful Compile window showing a different error

The first error is due to the fact that the output "o" is not listed within the module's port list despite being declared as output on line 3. The second error indicates that a variable ("o" in this case) is being used in

an always block but was not declared as a "reg" variable. You may also receive this error if you attempt to use a continuous assign statement to assign a value to a "reg variable. Fix the code by adding ", o" on the first line, and "reg" on the third line. The full corrected code appears in Figure 3-4. The errors.v file should now compile without errors after these last changes.
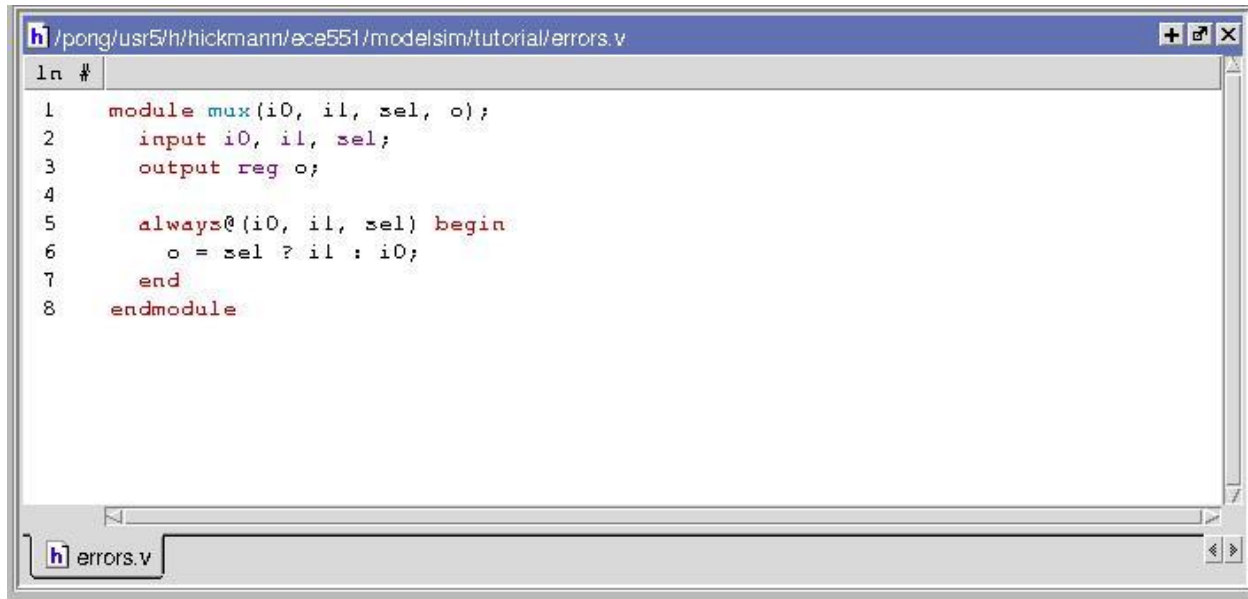


```
ln #
1    module mux(i0, i1, sel, o);
2      input i0, i1, sel;
3      output reg o;
4
5      always@(i0, i1, sel) begin
6        o = sel ? i1 : i0;
7      end
8    endmodule
```

Figure 3-4: Corrected errors.v code

4 Load the Testbench

We have now successfully compiled all of the files for the project. We wish to simulate the compiled files to determine correctness. First, a testbench must be loaded. To do this, perform the follow command.

Simulate->Start Simulate

The dialog appears as in **Figure 4-1**Error! Reference source not found.**.** Expand the work library (click + on work) and select *t_receive*. To get us started, we are fist going to test the receive sub-module by itself to ensure that it works separately from the rest of the project. Testing sub-modules before using them in a larger design is good habit to get into and will reduce the amount of time you will spend debugging. Select *t_receive* from within the work library.
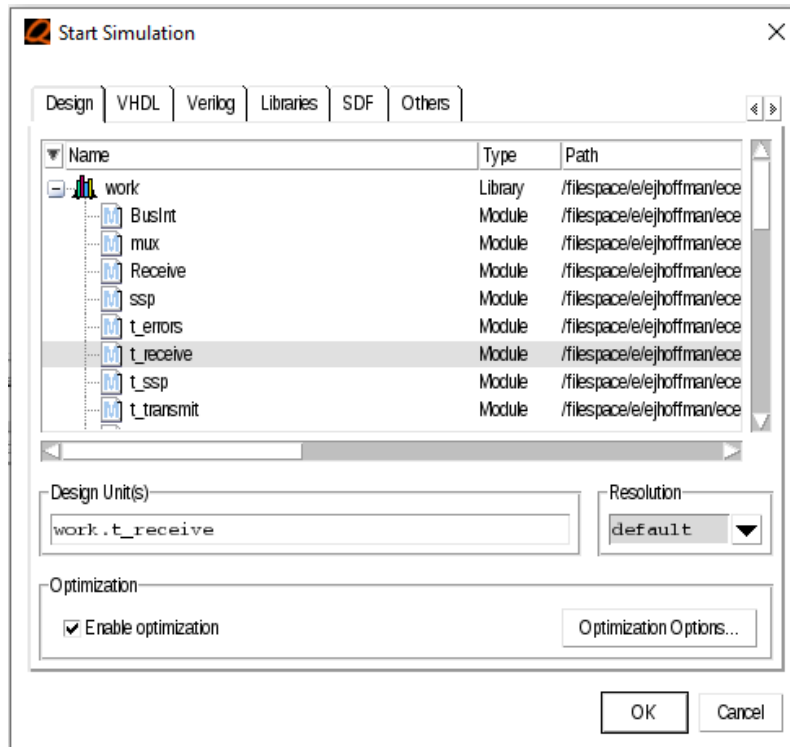
Figure 4-2 (Start Simulation Form)

Next you should click the "Optimization Options" button and select "Apply full visibility to all modules(full debug mode)" See Figure 4-3 below.
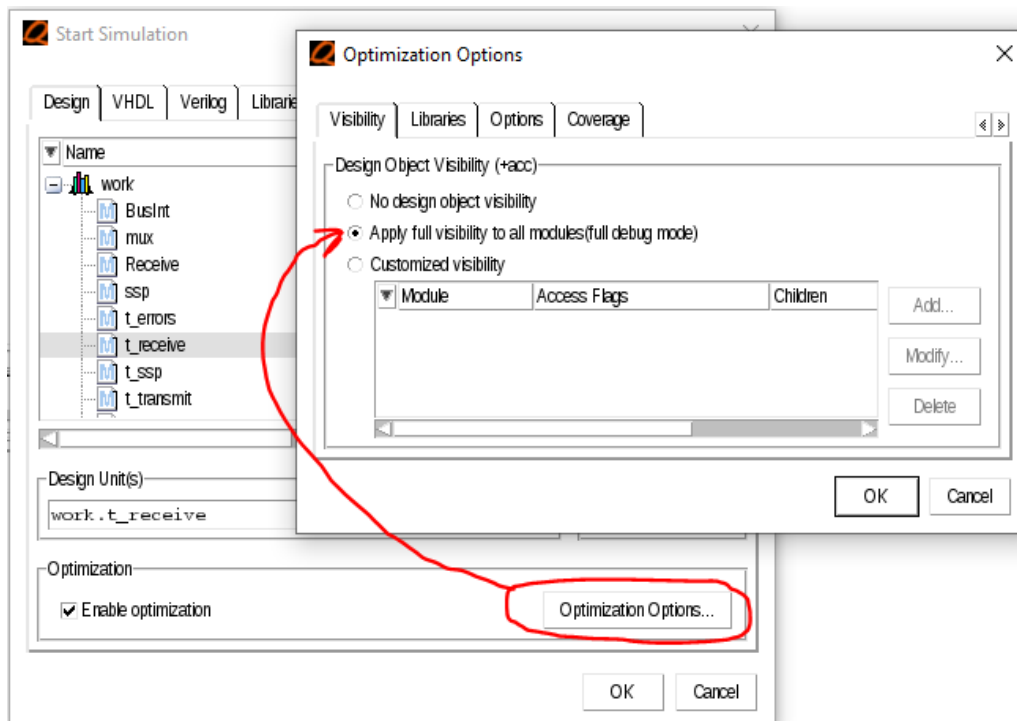
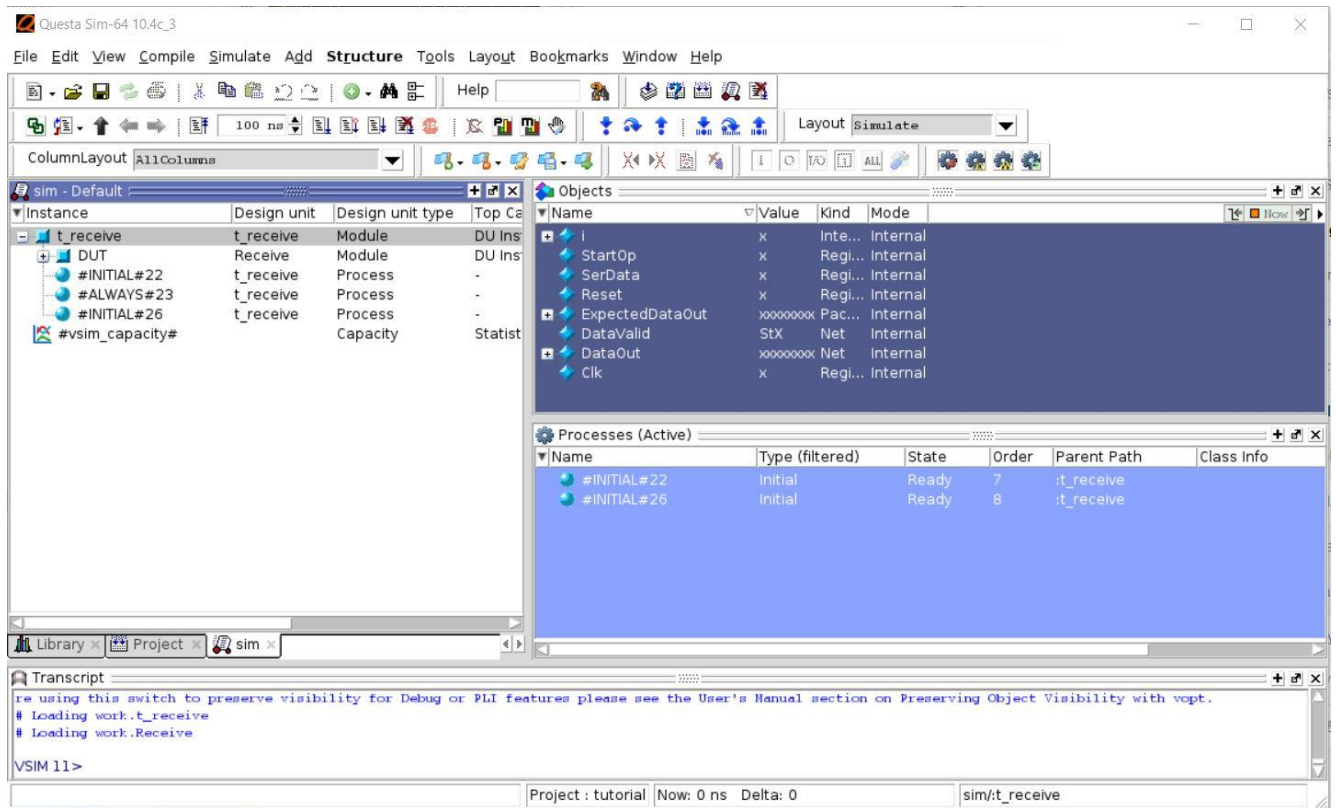**Figure 4-3: Choosing Optimization Option**

Figure 4-4: Main ModelSim window after loading the testbench

If loading of the testbench is successful, the main window should now appear similar to Figure 4-4, though you may have to expand "DUT" and its sub-modules. Note that there is now a **sim** tab in the workspace. In the figure, all of the design instances are expanded so that everything is visible.

In the **sim** tab, there are three primary columns. The first column, **Instance**, shows the instance name of each structure that is currently under simulation. The second column, **Design unit**, shows the module name for each of the instances. The third column, **Design unit type**, classifies the instances into different types according to their function. In this example we see there are **Module** types and **Process** types.

To the right of the **sim** tab in a separate embedded window is the **Objects** window. This window shows all of the signals for the currently selected instance in the **sim** tab and their values at the current timestep or the timestep indicated by the waveform cursor. This window is discussed next.
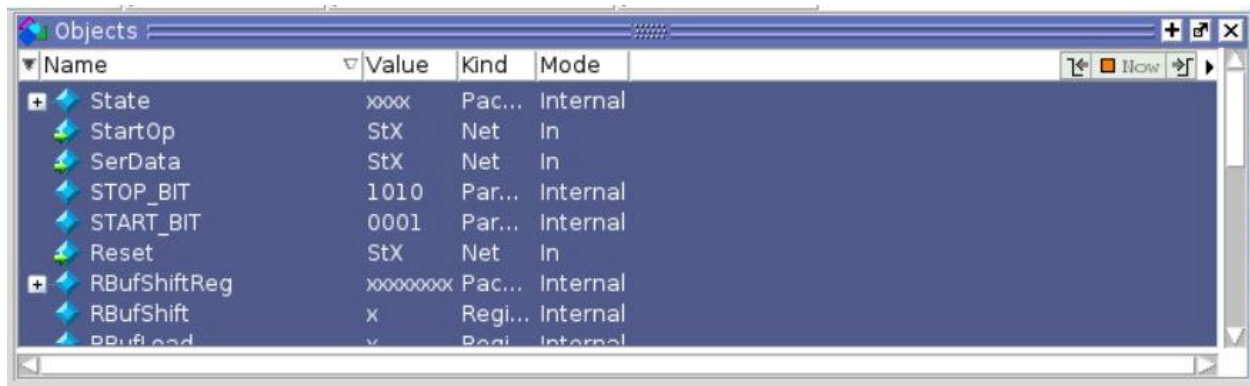
5 The Objects Window

The **Objects** window is context sensitive to whatever instance you currently have selected. The **Objects** window allows you to view what signals are defined in different areas of your design, as well as choose which signals to add to the **Wave** window. The **Objects** window may also be used to force signals to a specific value for debugging. If this window ever gets closed, you can reopen it by selecting **View -> Objects**.

Using the Objects Window

Select instance "DUT" in the **Workspace** of the main window. You may need to expand the tree to see "DUT".

The **Objects** window now shows all signals (Inputs, Outputs and internals) that are related to the DUT instance of the receive module, as shown in Figure 5-1. Note how values are displayed in the value column. Registers larger than 1-bit are displayed as multiple bits without any space. Multi-dimensional registers use curly braces to indicate the different rows and columns within the array.

Figure 5-1: Objects window

6 Running a Basic Simulation

Now that we have successfully loaded our simulation and investigated the default ModelSim simulation window, it is time to run our simulation and view the results. The basic commands for controlling your simulation are described below.

Running for a Fixed Amount of Time

Running your testbench for a fixed amount of time is a good option if your testbench does not have a $stop or $finish statement or if you only want to view the first few tests instead of running your testbench in its entirety.

Run your simulation for 100 ns. There are two ways to do this

Method 1

Typing 100 ns in the Run Length box in toolbar and click the **Run** button (the button directly to the right of the Run Length box)

Method 2

Type **%> run 100ns** in the transcript window and press enter.

Once the simulation is run for 100 ns, you will notice that the signals within the **Objects** window have changed from *x*'s to actual binary values. Also, in the bottom bar now reads "Now: 100 ns" to indicate that the simulation is currently at 100 ns.

Running the Simulation to Completion

Another option is to run your simulation until your testbench reaches a $stop or $finish statement. This is the easiest option if you want to run all your tests with one command.

Run your simulation to completion. There are two ways to do this.

Method 1

Click the **Run –all** button on the toolbar.

Method 2

Type **%> run -all** in the transcript window and press enter.

Your simulation time should now be 7,905 ns. When a $stop or $finish statement is reached, a window appears showing you where this statement is in the code.
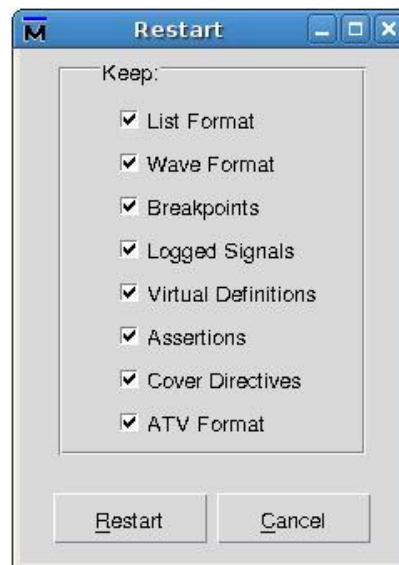
Restarting the Simulation

Restart resets the simulation time to 0 and removes all of the data from your list or wave windows. It does not change any of your other setup so it is an important time-saving feature so that you do not need to perform the same setup multiple times.

Restart the simulation now.

Method 1

Click the **Restart** button 📋 in either the main window or the **Source** window.

The dialog shown in Figure 6-1 will appear. Leave all boxes checked, and click **Restart**.

Figure 6-1: Restart dialog

Method 2

%> restart –f

Note that the –f option is used to force restart of the simulation, which will not show up when using method 1 since you have manually checked restart options.

Your simulation time should now once again be at 0 ns.

<u>Ending the Simulation</u>

Once you are finished with a simulation you must end it before starting a new simulation or changing ModelSim projects. There are two ways to do this:

Method 1

Select **Simulate -> End Simulation** from the menu bar.

Method 2

Type **%> quit -sim** in the transcript window and press enter.

**DO NOT END THE CURRENT SIMULATION AT THIS TIME**

7 The Wave Window

The wave window is your main tool for interactive debug of your design. It allows you to view how different signals within your design change over time. Any signal within your design may be placed within the wave window and several different options are available for formatting this window.

<u>Add Signals to the Wave Window</u>

With "t_receive" selected in the **Sim** tab, in the **Objects** window select "Clk", "SerData", "DataOut", and "ExpectedDataOut" (shift-click for a range, ctrl-click to select multiple). With all these signals selected, right click, and select **Add to Wave->Selected Signals**. The waveform window will now appear with three signals.

All the signals for a module can also be added to the waveform window. In the **sim** tab of the main window, select "DUT", right click and choose **Add->Add Wave**. Now all the signals within the receive module should appear in the waveform window.

When selecting **Add Wave** from the **sim** tab, all signals will be added, whereas choosing **Add Wave** from the **Signals** window allows us to select specific signals. Note that multidimensional (2+) signals will not be added when adding from the **sim** tab, so you must add them manually.
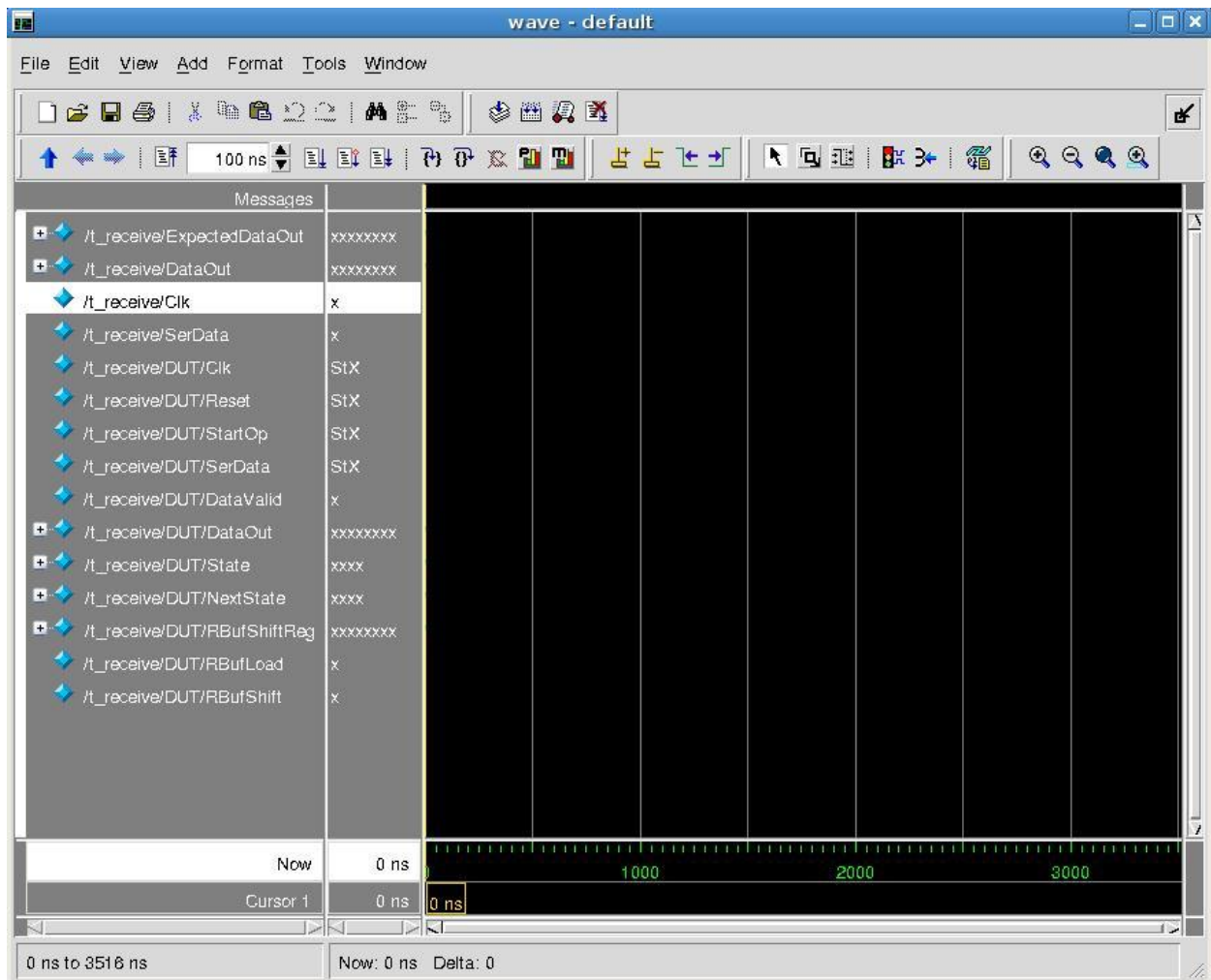
Figure 7-1: Wave window

You may need to expand the size of the window and certain columns, but your wave window should now look like Figure 7-1. The names of the signals are currently several levels deep. In larger designs this problem becomes even worse. We can change the number of levels that are seen.

Detach the wave window from the ModelSim main window by clicking the button to the left of the X in the window. The button looks like a box with an arrow pointing to the upper right. Doing this will enable additional options for the waveform window.
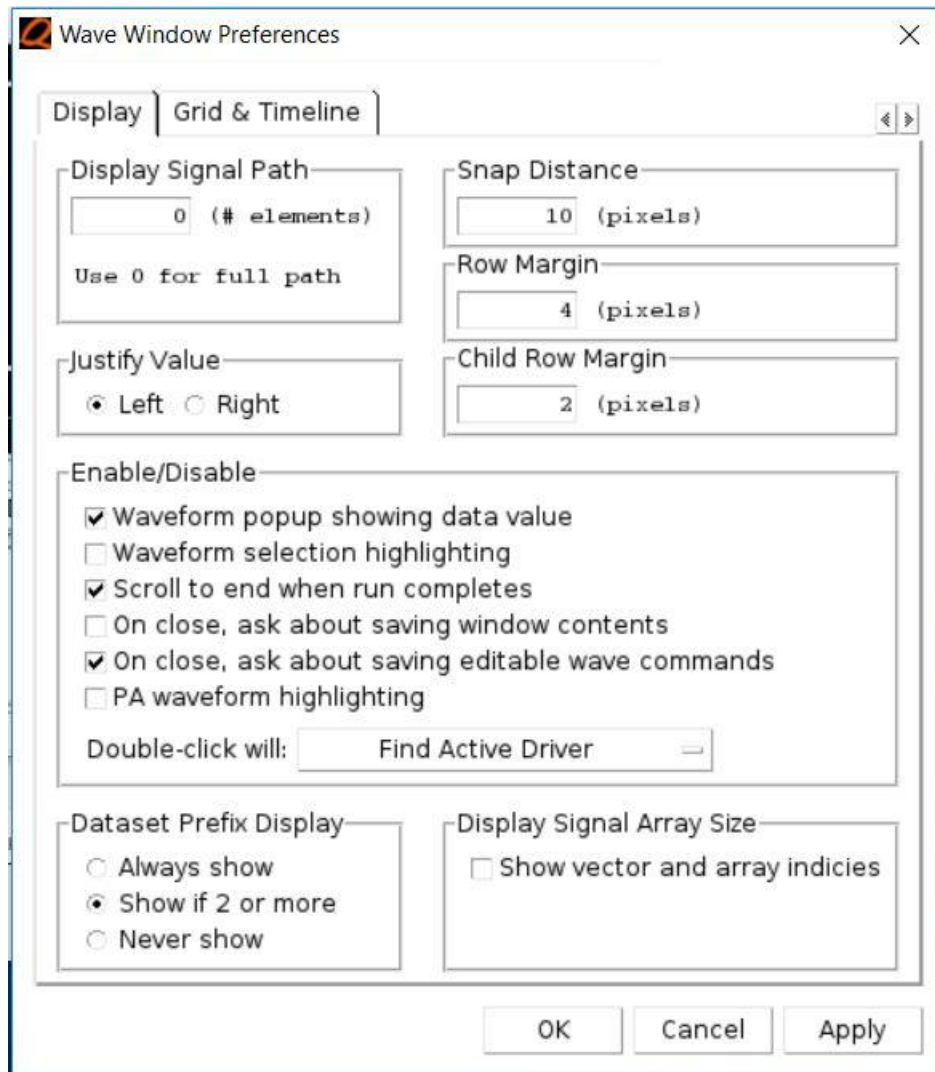
MAKE SURE TO DO THIS OTHERWISE THE TUTORIAL WON'T MATCH.

Change Signal Display Options

Tools->Window Preferences *(In the wave window, not the main window)*

The window preferences dialog appears as shown in Figure 7-2.

Figure 7-2: Wave Window Preferences dialog

We can change the **Display Signal Path** to 2 so that we can see the name of the signal and the module it is contained in. Some other useful values are *0* which will show the full path and *1* which will show only the signal name.

Run the Simulation

Using the methods described above, run your simulation for 2000 ns.

Your wave window should now look like Figure 7-3.
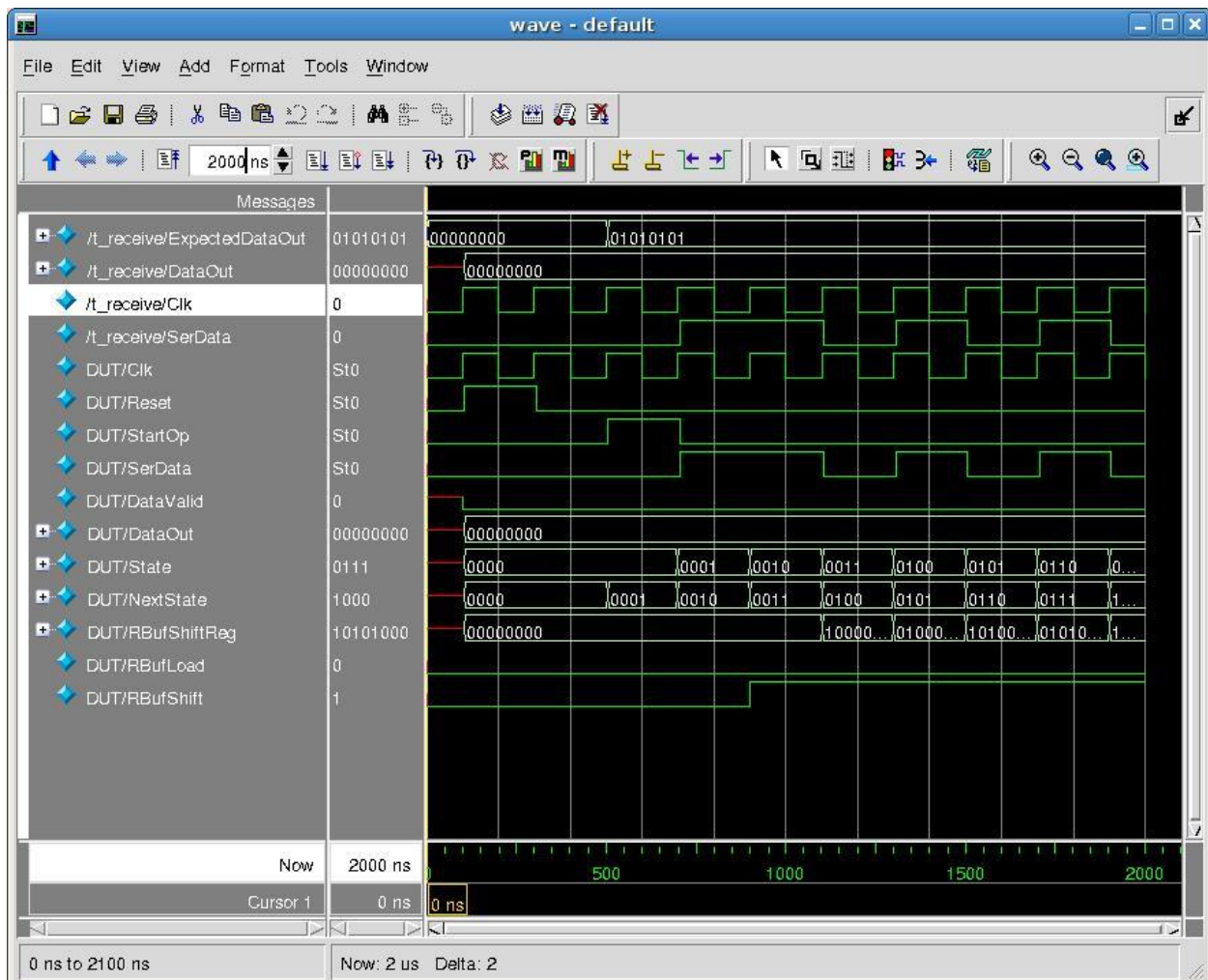
Figure 7-3: Wave window after running the simulation with the given commands

Zoom

You will find at times that you need to **Zoom In** on a signal value in a particular time span, or that you need to **Zoom Out** to get a better overall picture of the operation of the hardware. Also, the **Zoom Full** option is very useful to allow you to see the entire operation of the circuit, and from there you can **Zoom In** to the area you need to examine. There are three graphical buttons that can be used.

Zoom In 2x

Zoom Out 2x

Zoom Full

You can also use a menu option

View->Zoom->Zoom Full

Yet another technique is to center click, and draw a box from lower right to upper left. The box defines the new area visible in the window.

Change the Signal Radix

It is helpful to change the radix of signals to view the data more easily.

Method 1

Select "ExpectedDataOut".

View->Properties

Change Radix from **Default** to **Hexadecimal**. Note that in **Properties** window you may also change the **Wave Color** and **Name Color**, discussed later.

Method 2

Select "DataOut" (all instances in the wave window). Right click, and from the context menu, select **Radix -> Hexadecimal**. Note that using this method, you may select several signals at once and change all of the radices at the same time.

Now make these signals displayed in hex:

DataOut, ExpectedDataOut, RBufShiftReg

Now make these signals displayed as unsigned:

State, NextState

Add a Window Pane

In designs with very large numbers of signals it may be useful to have multiple window panes so that signals that are common throughout the design are visible while you scroll to see other signals in the window. To add a window pane:

Add->Window Pane

A second window pane appears in the bottom of the wave window. Drag all signals except "clk" and "rst" to the bottom pane. Resize the window panes so that the top pane is no larger than the 2 signals remaining. The **Wave** window should now look like Figure 7-4. Note that you may have as many window panes as you wish, although fewer panels are probably easier to work with.
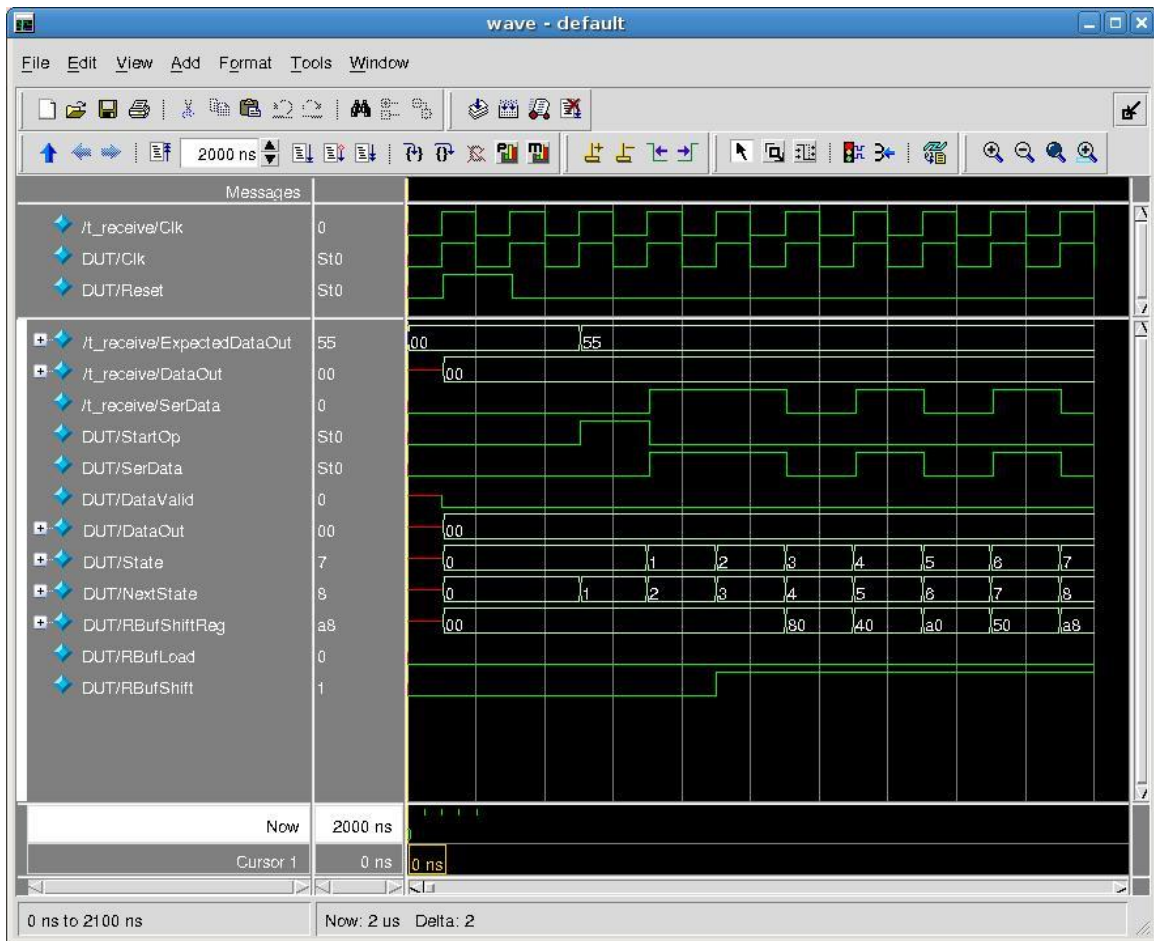
Figure 7-4: Wave window with multiple panes

Add a Divider

Another feature to help organize many signals is dividers. You may insert a divider to separate logical groups of signals and make the wave window easier to read. You can right-click on any divider you have created to open a context menu, where you can select **Delete** if you no longer need the divider.

Method 1

Select "StartOp".

> **Add->Divider**

Figure 7-5: Wave Divider Properties dialog

Change the **Divider Name** to "DUT signals" and change **Divider Height** to 30. You now have a divider between the signals. You may change the name of the divider or the divider height at any time by right clicking and choosing **Divider Properties**.

Method 2

Select the signal that you want to create a divider above. Use the right-click contest menu to **Insert Divider**. You get the same dialog as Figure 7-5.

Change the Signal Color

In a design it may be useful to distinguish certain signals from others by changing their color.

Change the color of the signals in the top window pane, by first selecting both signals ("clk", "rst").

Format->Color

The dialog shown in Figure 7-6 should appear. Click on **Choose Color** and select **Orange**, or just type in Orange. Click **OK**.

Figure 7-6: Wave Color dialog

The signals in the top pane are now orange, as shown in Figure 7-7. Note that the signal coloring here overrides the default coloring, so do not change the color of signals if you wish to use the default coloring to tell the difference between x's, z's and binary values.
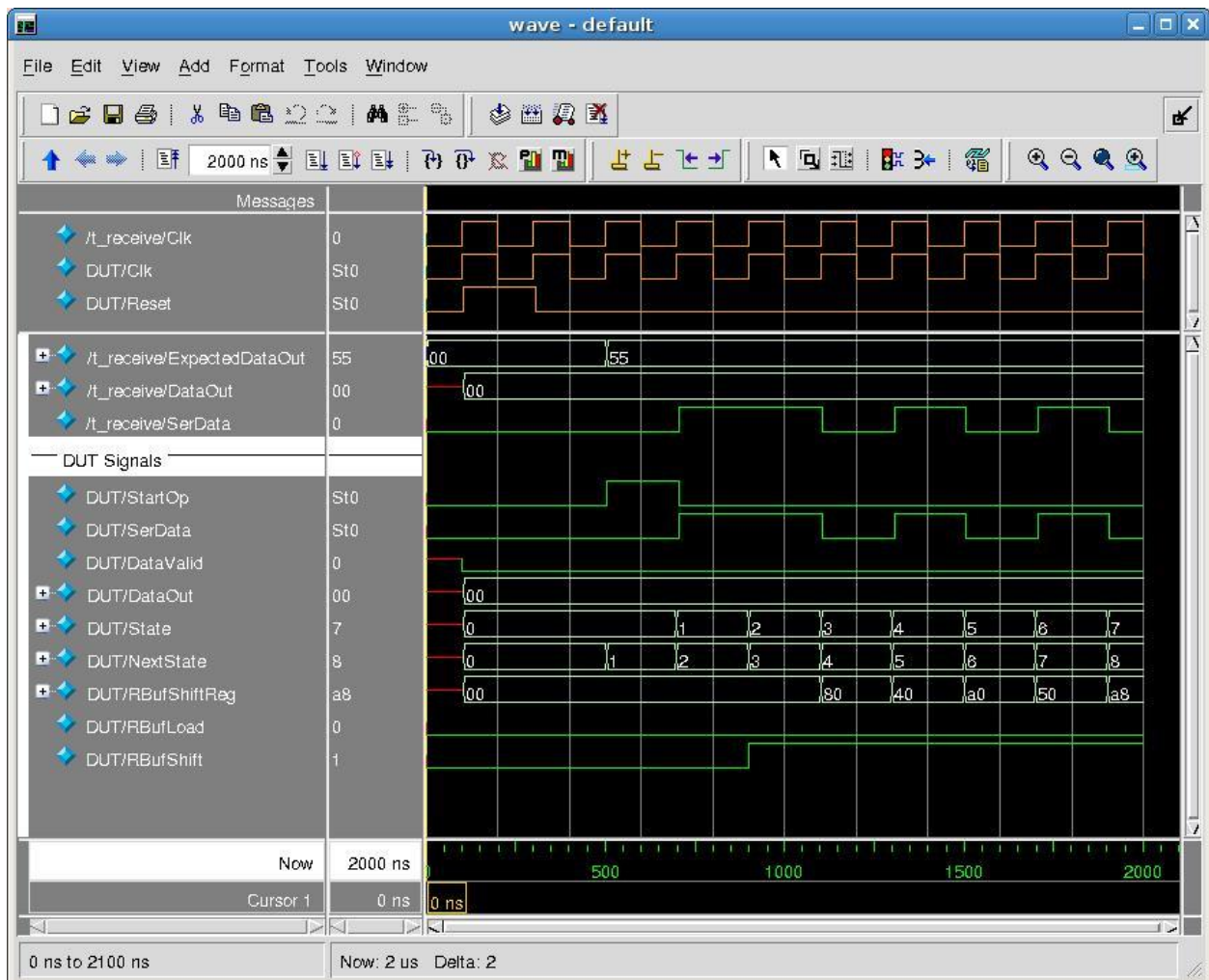
Figure 7-7: Wave window showing orange signals in the top pane

Cursors

Cursors are used within ModelSim to specify the time at which you want to view a signal's value and also measure the time between two events. By default, the wave window begins with one cursor. Wherever you click in the wave window, the cursor will be placed at that time. There are a few cursor buttons:

Add Cursor:

Delete Cursor:

Previous Transition:

Next Transition:

Move a Cursor: Part I

Move the cursor to 700ns by dragging the cursor within the wave window until its time is 700ns.

Alternative ways to move the cursor:

- Right click on the time next to "Cursor 1" (in the signal column, not the value under the cursor), and then enter 700 ns.

- Right click on the cursor (at the bottom of the screen) and choose **Cursor Properties**. Change the time to 700 ns.

Change a Cursor Name

Right click on the cursor's name in the lower left hand corner. It is now editable. Change the name to "StartOp".

Add a Cursor

You can either use the graphical **Add Cursor** button, right click on the bottom part of the wave window and select **New Cursor**, or

Add->Cursor

A new cursor has been inserted, and a measurement of the amount of time between the two cursors is shown at the bottom of the screen. You will also see that there is a row for each cursor in the wave area. Right clicking in one of these rows will open the context menu for the corresponding cursor.

Move a Cursor: Part II

Select the signal that you want to examine. You can use the **Previous Transition** and **Next Transition** to move the cursor along that signal. Alternately, you can press **Tab** for the next transition and **Shift-Tab** for the previous one. The cursor will move along the selected signal.

Save Settings

Now that we have our wave window set up to appear as we want it to, let's save the format so that we don't need to reproduce all of the changes we made every time we want to run this simulation.

File->Save Format…

This will bring up a dialog box with a path to save a "wave.do" file. You could give this a more meaningful name if you like, but keep the .do extension. Click **OK**. Now, when we wish to run this

simulation, we can choose **File->Load…** so that all of the settings we have specified will be there (note that the data itself is not saved, just settings such as what order the signals are in, their radix and color etc.).

Save Datasets

If we have a known good dataset, or we simply want to save the data to analyze later, the data set is automatically saved in a file called **vsim.wlf**. We can open this data any time by doing

> **File->Open...**

When we re-open the dataset, we must either have a saved format as well, or we must manually create a wave format as we did earlier.

8 Simulation Debugging

Now that we have learned how to setup our waveform window, we will look at how to use it to test the receiver module that we are simulating. Within t_receive, we have a number of tests which input data to the receiver as a serial signal on the "SerData" signal. We want to make sure that the parallel byte of data output by the receiver on its "DataOut" signal matches the serial data put into the module.

Debugging Using an Expected Value Signal

To verify that the output data matches the input data, the testbench stores the byte it has input on the serial link in a signal called "ExpectedDataOut". To see that the receiver works correctly, we need to ensure that the "DataOut" signal matches the "ExpectedDataOut" signal at the end of each operation. To do this, simply move these 2 signals next to each other on the **Wave** window as shown in Figure 8-1.

Press the **Run –all** button to finish your simulation of the t_receive modulation

Next, select the "DataValid" signal from the receiver and place "Cursor 2" at time 0. Next, simply use the **Next Transition** ( ) button to find each time this signal goes high. Each time this signal is high we simply need to verify that the "DataOut" and "ExpectedDataOut" signals match. If they match for all the tests, then the receiver is working correctly. An example of this process is shown in Figure 8-1.
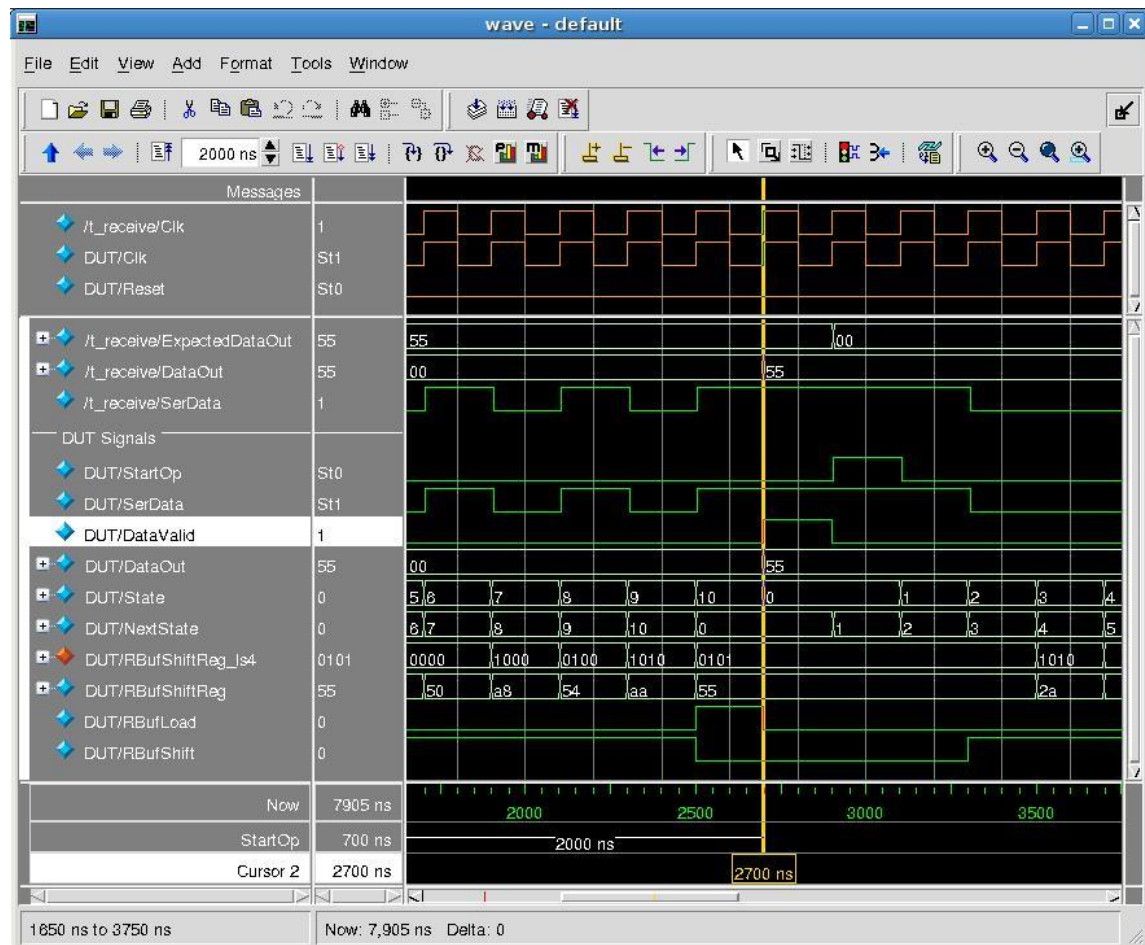

Figure 8-1: Wave Window Using the ExpectedDataOut Signal for Debugging

Self-Checking Testbenches

A more advanced debugging technique is to add code to your testbench to check the values for you. This simplifies the process of checking your simulation output for correctness. A self-checking testbench is illustrated in the "t_ssp" testbench.

Please end the current receiver simulation by typing **quit –sim** in the transcript window and close the current wave window. Or by choosing **Simulation → End Simulation** in the menus of the main modelSim window.

Start a new simulation using the "t_ssp" module as described above.

Add all of the testbench signals by right clicking on "t_ssp" on the **sim** tab and selecting **Add -> Adds Wave**.

Type **run –all** in the typescript window to run the simulation of the entire project until completion. The simulation should end at 1,280,505 ns.

Click the **Zoom Full** button to view the entire waveform. It should look like the window pictured in Figure 8-2.

The t_ssp testbench is self-checking and generates a signal called "error" that will go high only if an error is detected during the simulation. A message will also be printed to the transcript window by a $display statement if an error is found.

Find the "error" signal on the wave and ensure that it never goes high during the simulation. While this testbench is slightly more complicated to write, it allows for easier checking of your design, especially as design get more complex or you run a large number of test cases. This testbench runs 256 test cases to exhaustively test sending every possible byte of information.
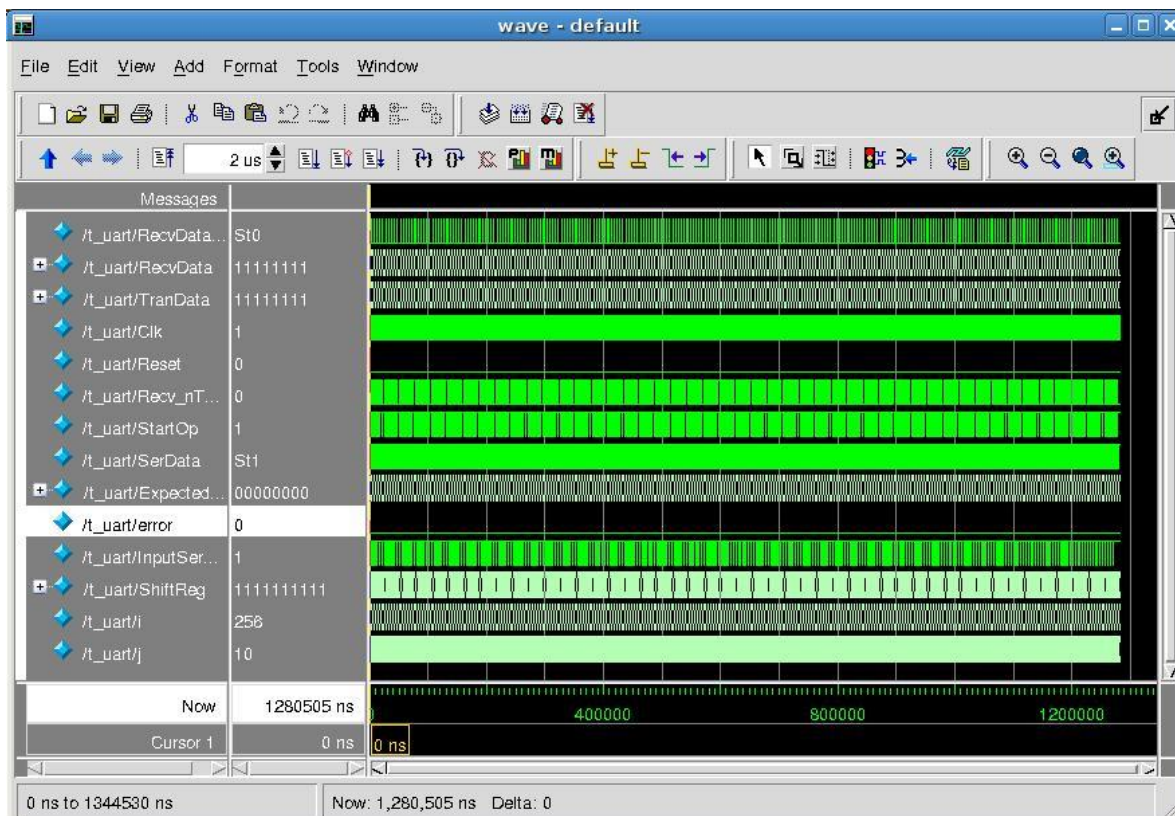

Figure 8-2: Wave Window for the Self-Checking t_ssp Testbench

HW1 Submission Check Point.

At this point Overlay the wave window on the main ModelSim window in a manner such that we can see your username in the transcript window (see image below) and take a snapshot of this. You will need to submit this snap shot as part of HW1 to prove you completed the tutorial.
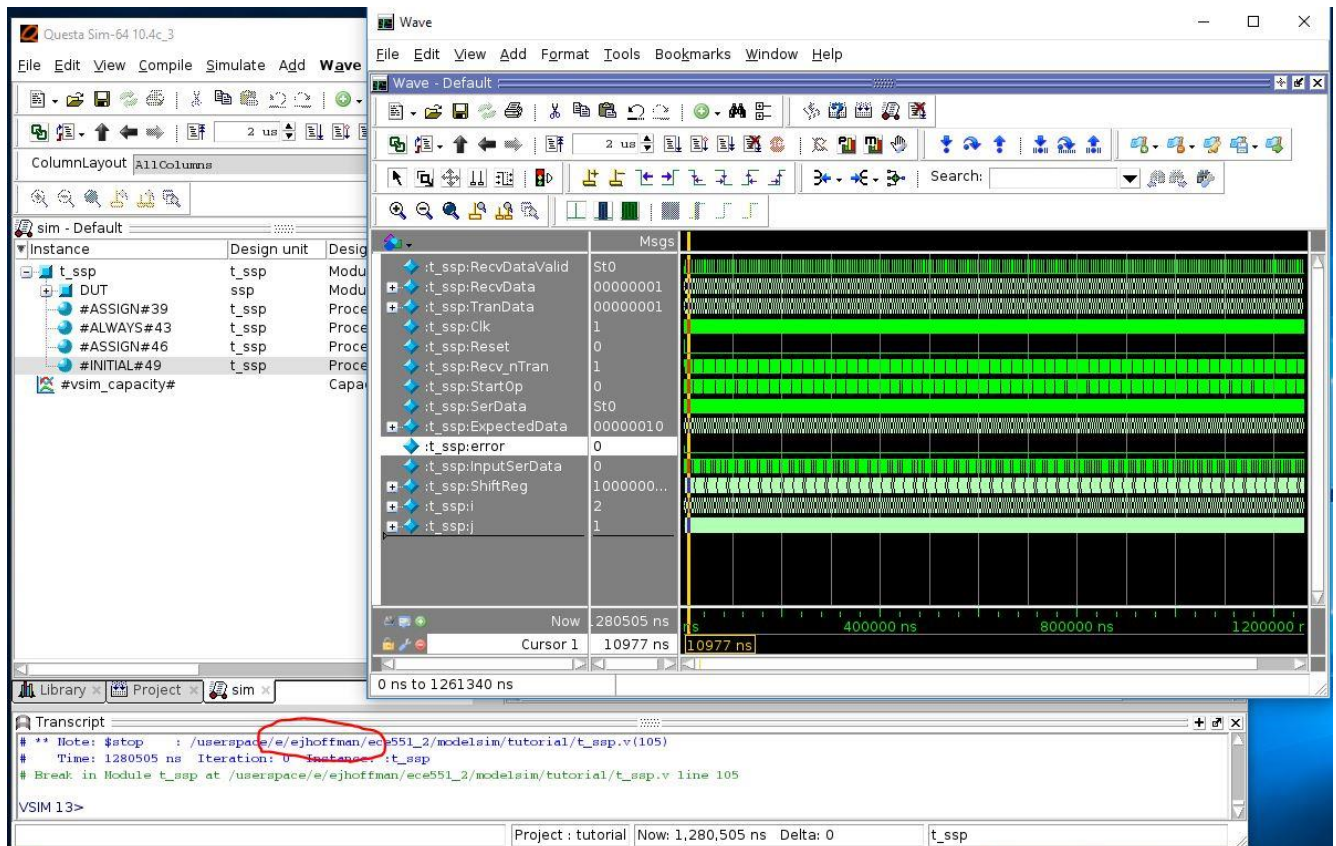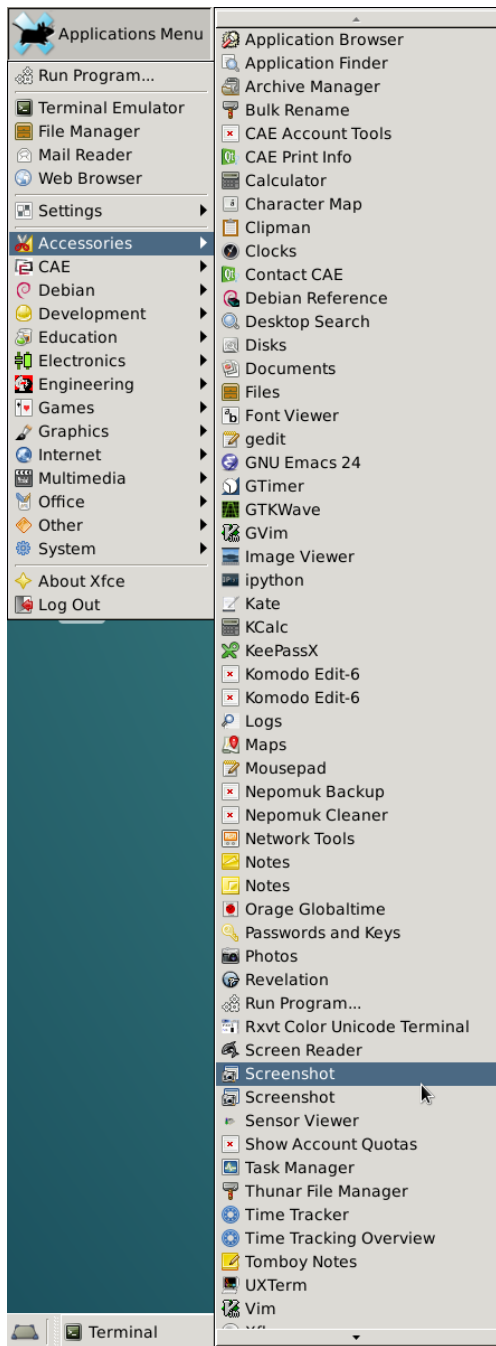

Figure 8-3: HW1 Problem 1 Checkpoint

Screenshots in Linux can be done through the applications menu in the upper left corner.

Applications Menu → Accessories → Screenshot

(See image below)

9 Breakpoints and Code Stepping

ModelSim provides **Breakpoints** as well as **Step Into** and **Step Over** buttons for use in debugging, similar in appearance to debuggers for software languages such as C or Java. One important difference to note is that Verilog is inherently a parallel language. When you step through Verilog, **you are actually stepping through time.** This means that it is possible (and very likely) that the next line to be executed is not the next line in the source code, with no explicit call or return. This also means that based upon other activity in the system, execution may or may happen in the same order twice in a row. Good

programming style will help make debugging easier to follow, but be warned that stepping through a Verilog program may be more difficult than the C or Java debugging you may be used to.

<u>Creating a Breakpoint</u>

All breakpoints are created by using the **Source** window, the same window used to edit the source files. While a simulation is running, some of the line numbers are highlighted in red. These are the "executable" lines of code. On these lines of code breakpoints can be created. These appear as red dots which denote a currently enabled breakpoint, while a black dot indicates a breakpoint that has been created but is not currently enabled. However, at this point in the tutorial you have not yet created the breakpoints or started to step through the code, so they will not appear on your window.

For this tutorial, we would like to create a breakpoint at line 88 of the t_ssp testbench. This will stop the simulation after each test of the receiver. To do this, open the **Source** window for t_ssp.v by double-clicking on this file in the **Project** tab.

There are three way to create a breakpoint at this point in the code

Method 1

Right click over the desired line of executable code in the "BP" column and select **Set Breakpoint** to set and enable a breakpoint on that line
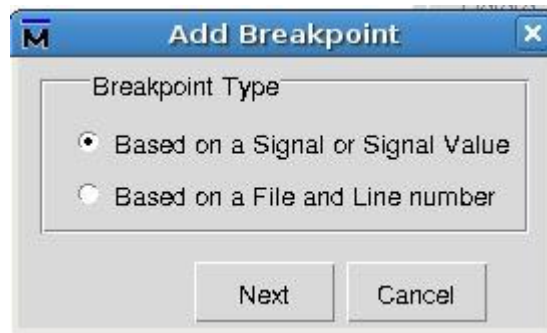
Method 2

Left-click once on the desired line number in the "BP" column within the **Source** window to create and enable a break at that line number (it must be an executable line of code). Click once more to disable the breakpoint. The breakpoint can be removed by right-clicking on it and choosing **Remove Breakpoint**.

Method 3

Select **Tools->Breakpoints**… from the menu**.** The dialog box in Figure 9-1 will appear. Note that this dialog may be accessed from the Source, Signals or Wave window in the same way. Click **Add** to create a new breakpoint. The dialog in Figure 9-2 appears.
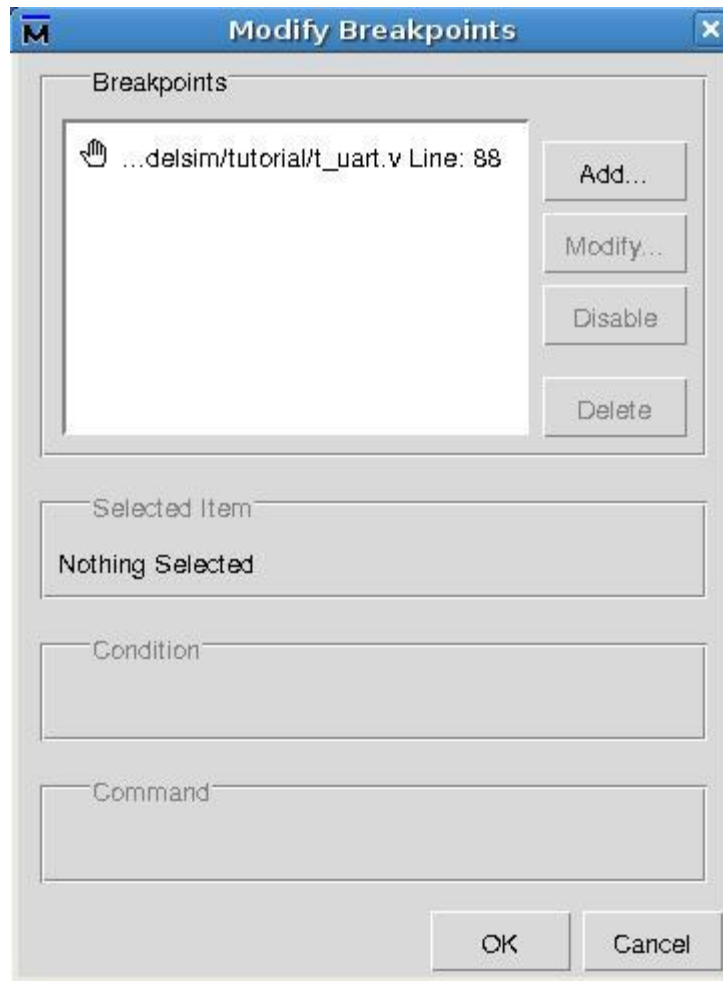
Figure 9-1: Modify Breakpoints dialog

Figure 9-2: Add Breakpoints dialog

Using this method, you may create breakpoints based upon a **Signal or a Signal Value** (discussed next) or a **Line Number** (like those in the first method). You can also use the **Modify Breakpoints** dialog to modify line number breakpoints made using the first method.

If **Signal or Signal Value** is chosen, the **Signal Breakpoint** dialog of Figure 9-3 will appear. You may choose a name for the breakpoint, specify a condition (such as equal to 0) and commands (such as printing a message).
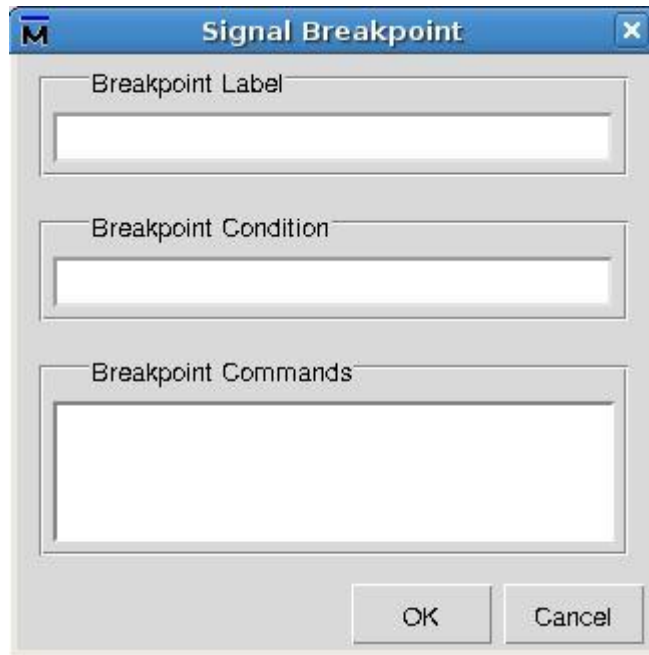
Figure 9-3: Signal Breakpoint dialog

If **File and Line Number** is chosen, the File Breakpoint dialog of Figure 9-4 will appear. You may choose the File (source file), line number, instance name (if the same module is used many times), as well as a breakpoint condition and commands like when creating a breakpoint on **Signal or Signal Value**..
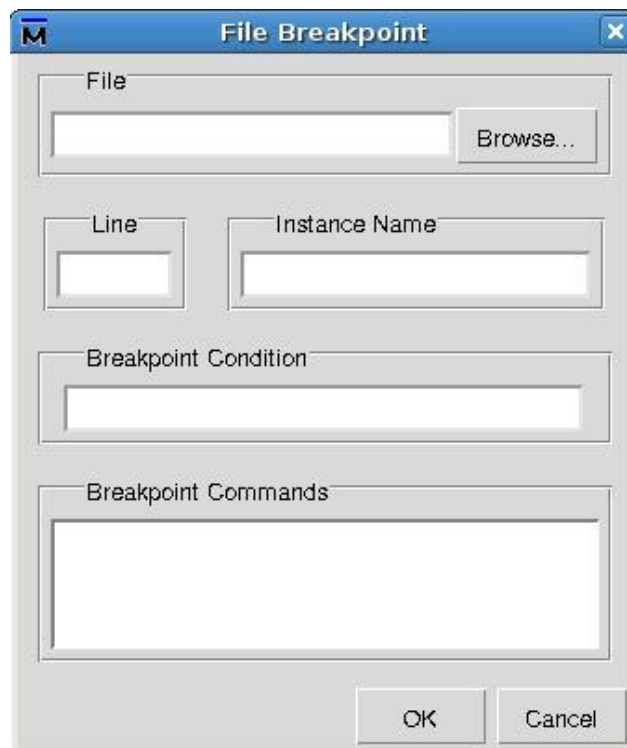
Figure 9-4: File Breakpoint dialog

Now that we have our breakpoint created, first restart the simulation by typing **restart –f** in the typescript window.

Next, type **Run 10 us** into the typescript command-line.

Notice that although we specified run for 10uS, we have reached the breakpoint and the simulation has stopped at 2,905 ns. The run button tells ModelSim to run for a specific amount of time, whereas the continue button tells ModelSim to run until the next breakpoint is reached. In that way, the continue button has a similar effect as run –all.

Click the **Continue** button

You will run again and stop at the same breakpoint. Note that the time listed below the **sim** tab has now changed to 7,905ns.

Disable the breakpoint.

Step and Step Over

ModelSim provides **Step Into** and **Step Over** buttons for use in debugging, similar to debuggers for software languages such as C or Java. Note that step over is used for function or task calls to step over, it does not step over a module instance, as an instance is not a function, it is continuously executing hardware.

Step Into

Step Over

Breakpoints and Stepping in Testbenches

One very useful place to use breakpoints is within testbenches where the Verilog code is stepped through in a sequential manner. It can be used to stop the simulation at a certain test or error condition so the waveform can be viewed at that point.

To illustrate the use of breakpoints in testbench, restart the current simulation of t_ssp. Open up the **Code** window for t_ssp.v and place an additional breakpoint on line 99. Since the 2 breakpoints are within the main test loop, the simulation will stop each time a test, receiver or transmit, is finished. Press the **Continue** button several times to observe this.

Breakpoints and Stepping in Design Code

Using breakpoints within your Verilog design is much less intuitive than in the sequential portion of testbenches. As mentioned above, Verilog is an inherently parallel language and therefore two sequential lines of code need not be executed in order or even in the same order at every timestep. It is important to realize that the next line of code executed is always the next event to happen in <u>time</u> and is not necessarily related to the order of the lines in the code. Also, a single line of code may be executed several times in a single timestep due to dependencies on other events in your code. While using breakpoints within your Verilog can be useful for debugging purposes, keep these caveats in mind while doing so.

Appendix A: Simulation Problems

Errors While Starting a Simulation

While we have seen errors during compilation, errors and warnings can also appear while starting a simulation. These usually involve the connections between modules, such as connecting a signal to a port on a module with a different width or having too many or too few port connections. While ModelSim will sometimes let you continue despite these errors, all of the above errors should be fixed before performing a simulation.

To see these errors, start a new simulation using the "t_errors" module. In the transcript window, you should see the errors as shown in Figure A-1.
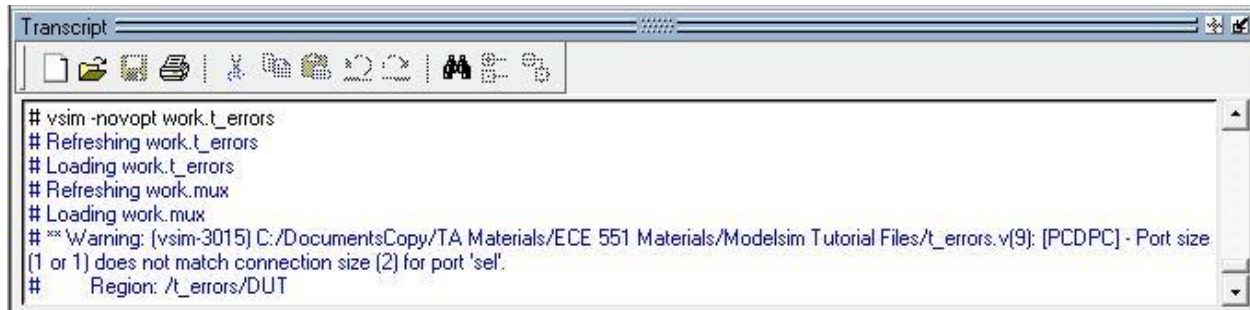
Figure A-1: Simulation Error during Startup

If you look in the code in t_errors.v, it is easy to see that the instantiation of the mux using an input wire which is 2-bits while the port is only 1-bit wide. Errors such as this must be fixed before simulations are run. Note that multiple modules having same name may not create compilation using but can cause simulation errors. Below is a list of common errors encountered when starting a simulation.

Error or Warning Message	Explanation
Too few or too many port connections	This error appears when a module instantiation has too many or too few connections. While this may not be an error in all case (i.e. you wanted to leave an output unconnected) it usually indicates an incorrect instantiation and should be checked.
Port Size does not match	This is an actual error which occurs when a connection to an instantiated module does not match the port's size. This should be fixed before moving on with simulation.
The design unit was not found	This means you forgot to add a module to your project. Make sure and double check that all your Verilog files have been added to the project and are compiled as well as double-check the offending module name for misspellings.
… has a `timescale directive in effect	In most situations this is nothing to be worried about. Only in situations where multiple modules are using delay statements should this be looked into further.

Appendix B: Printing

The wave window <u>can</u> be printed either directly to a printer or to a Postscript file. To print directly to the nearest printer, choose **File->Print Postscript**, then change the **Print command** in the **Write Postscript** dialog to "lp". If you wish to annotate waveforms in software or paste images of the waveforms into a document, it is recommended that you print to postscript <u>first</u> so that the image is converted to black and white (unless you plan to print in color later). You may then open the files in **gv** (GhostView) or convert them to PDF files (www.ps2pdf.com) to copy the images into the program or document you wish to use.

Appendix C: Code Coverage

Code coverage is the only verification metric generated automatically from design source in RTL or gates. It only measures how often certain aspects of the source are exercised while running a suite of tests. ModelSim Code Coverage gives you graphical and report file feedback on which executable statements, branches, conditions, and expressions in your source code have been executed. It also measures bits of logic that have been toggled during execution.

Step 1: Compiling with Code Coverage Enabled

Right click in the *Project Window* and under *Compile* select *Compile Properties*.

In the *Project Compile Settings Window*, under *Coverage* select all the necessary coverage metrics and click *OK*. Figure C-1 shows the above mention window.
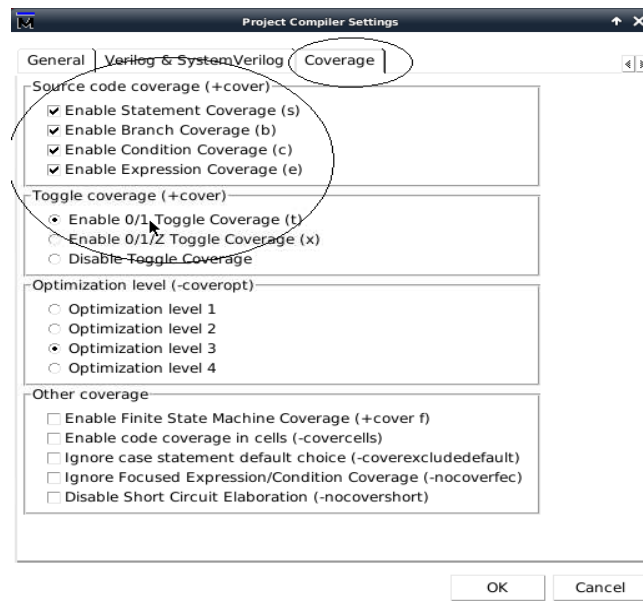
Figure C-1: Project Compiler Settings

Now compile the entire project with code coverage enabled.

Note: In order to exclude file from coverage statistics such as the Testbenches, one should disable all the coverage options for such files in the Project Window.

Step 2: Simulating with Code Coverage Enabled

Once the coverage types have been specified for coverage, enable the simulation to collect the code coverage statistics using the following method:

Simulate > Start Simulation > **Others** > **Enable Code Coverage** checkbox, as shown in Figure C-2.
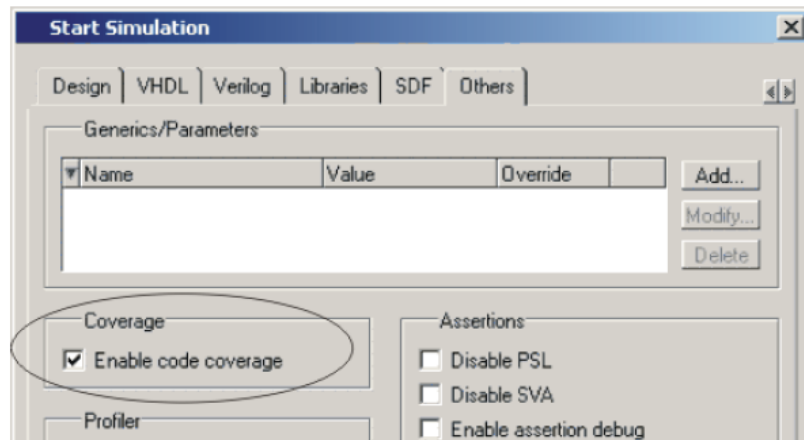
Figure C-2

Step 3: Observing the Coverage Details

When you load a design with Code Coverage enabled, ModelSim adds several columns to the Files and sim tabs in the Workspace as shown in figure C-3. ModelSim also displays three Code Coverage panes in the Main window as shown in figure C-4:

• **Code Coverage Analysis**
Displays the selected file's un-executed statements, branches, conditions, and expressions and signals that have not toggled.

• **Instance Coverage**
Displays statement, branch, condition, expression and toggle coverage statistics for each instance in a flat, non hierarchical view.

• **Details**
Shows details of missed coverage such as truth tables or toggle details.

Figure C-3: Coverage columns in the Main window Workspace
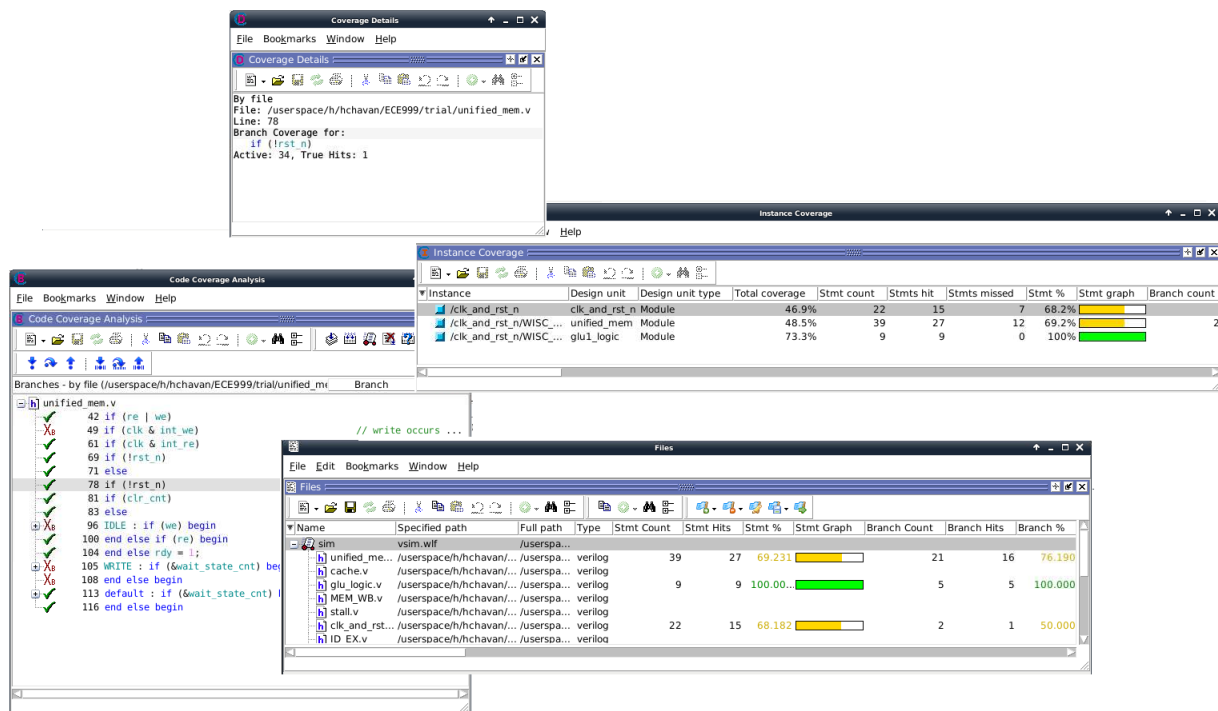
Figure C-4: Coverage Panes

Source window statistics

Code coverage statistics are displayed in the Source window when coverage is invoked. You can view the source code for specific modules or entities by double-clicking an item in the Files or sim tab of the Main window Workspace, or by selecting any item in the Code Coverage Analysis or Instance Coverage panes.

- In the Files tab of the Main window Workspace, double-click the file to open it in the Source window.
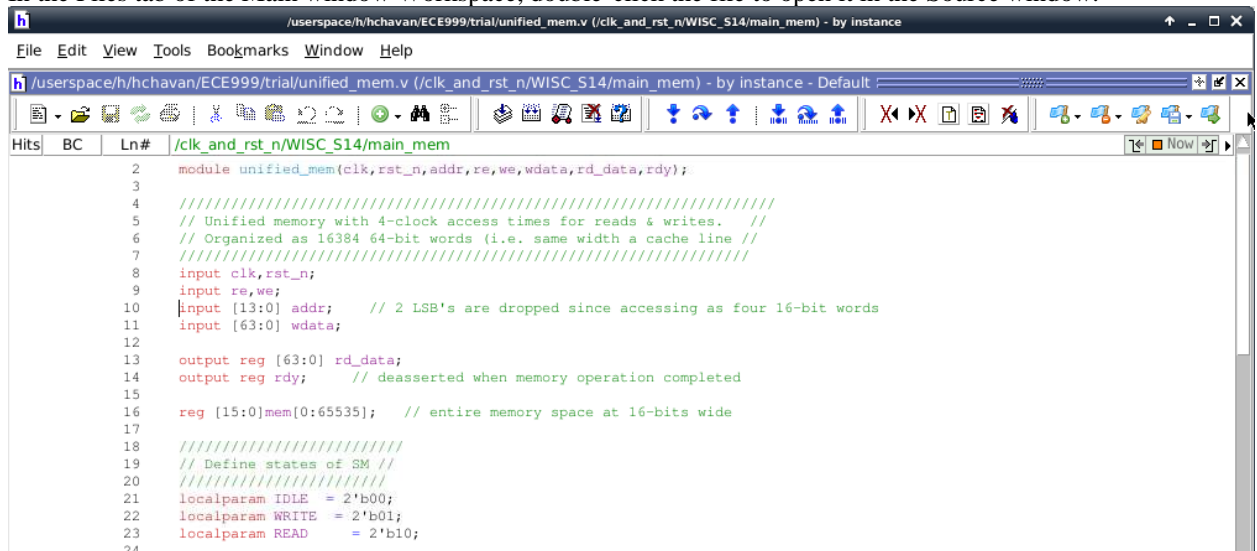
Figure C-5: Source Window

- To go directly to a line of code that contains a statement that has not executed, click the plus sign next to the file name in the Code Coverage Analysis pane (with the Statement tab selected) and select the corresponding line.
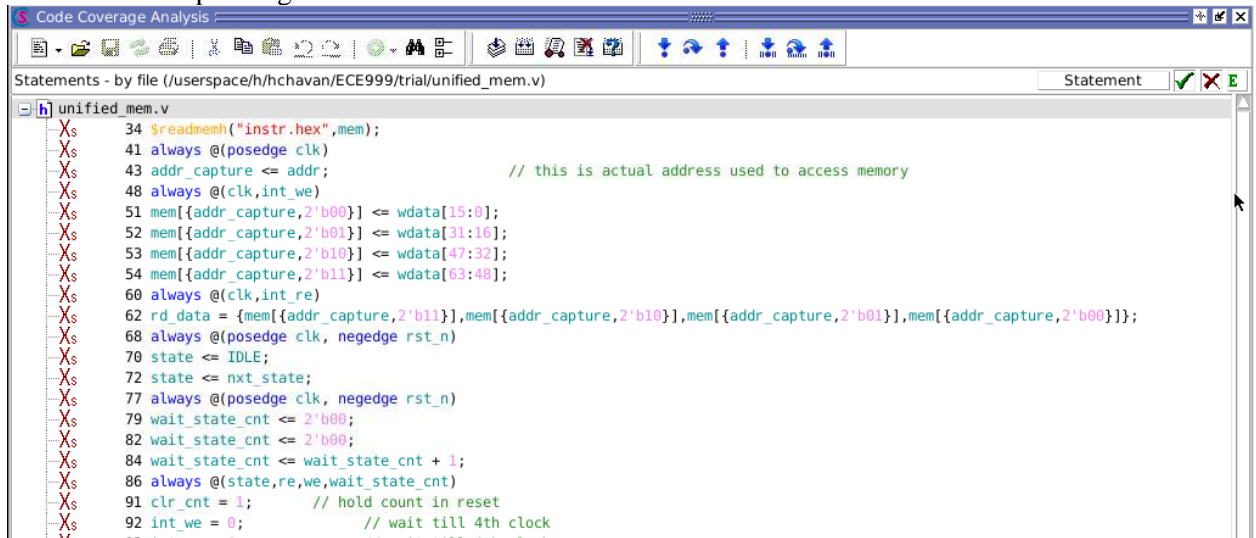
Figure C-6: Statement Coverage before Execution in Code Coverage Analysis Window
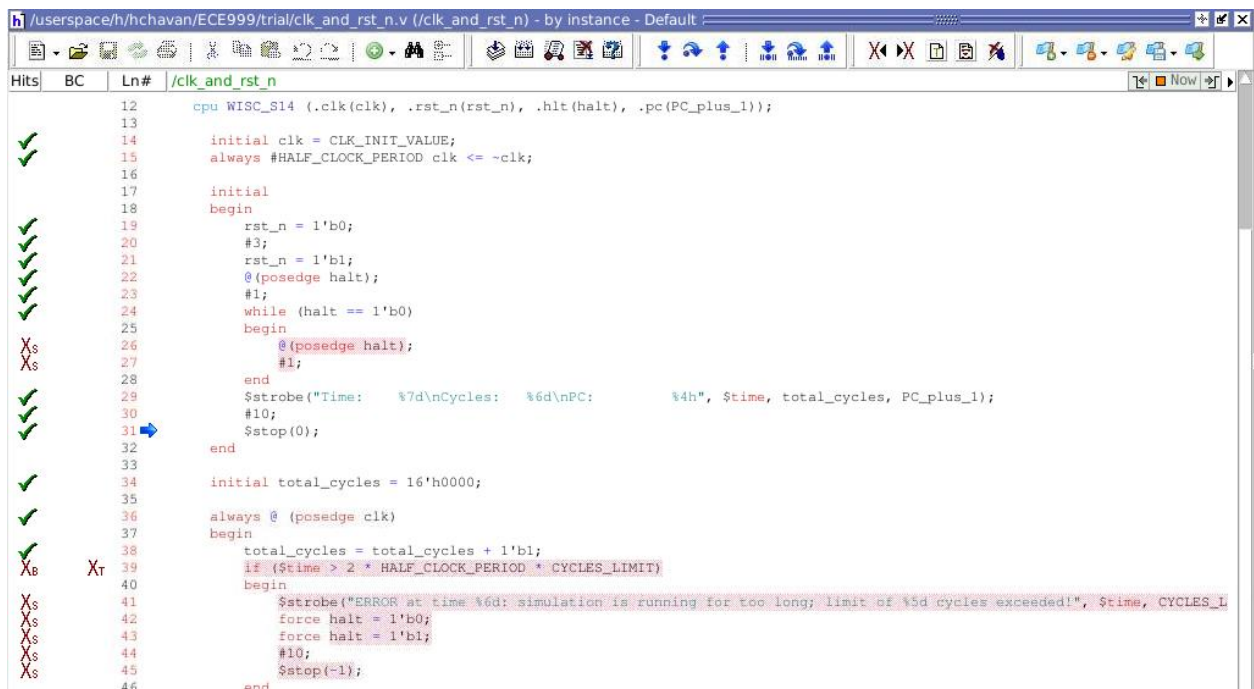
Figure C-7: Source Window after Execution

A red X in the Hits column indicates that a statement in that line has not been executed (zero hits). A green E in the hits column indicates a line that has been excluded from code coverage statistics. A red X_T or X_F in the BC (Branch Coverage) column indicates that a true or false branch (respectively) of a conditional statement has not been executed. Lines that contain unexecuted statements and branches are highlighted in pink.
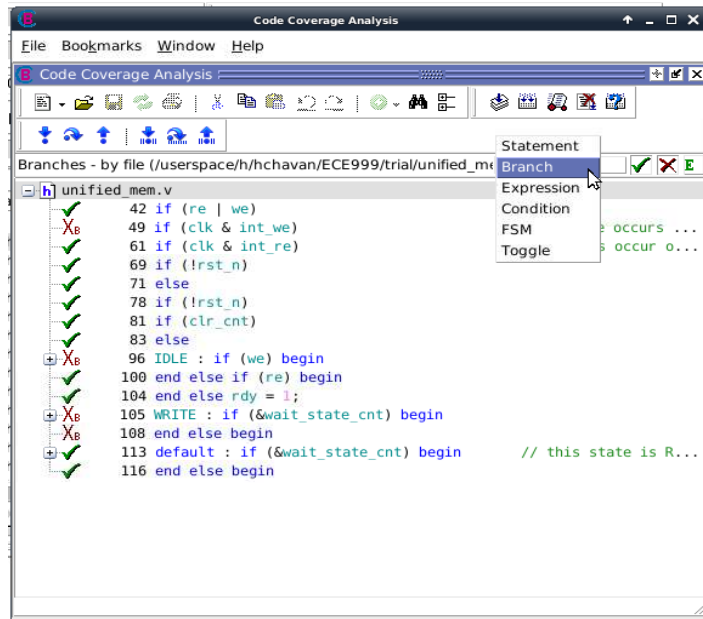
Figure C-8: Branch Coverage after Execution in Code Coverage Analysis Window

In order to change the analysis type, click on the analysis type icon as shown in figure C-8 and select the necessary type of coverage that you wish to observe.

Step 4: Saving the Code Coverage in the UCDB

UCDB stands for Unified Coverage Database. When you run a design with coverage enabled you can save the code coverage that was collected for later use, either on demand or at the end of simulation. By default, even if coverage is enabled, the tool will not save the data unless you explicitly specify that the data should be saved.

Often, users simulate designs multiple times, with the intention of capturing different coverage data from each test for post-process viewing and analysis. When this is the case, the naming of the tests becomes important. By default, the name ModelSim assigns to a test is the same as the UCDB file base name. If you fail to name the test you run explicitly, you can unintentionally overwrite your data.

For saving coverage data dynamically (during simulation) or in coverage view mode
Tools > Coverage Save
This brings up the Coverage Save dialog box, where you can specify coverage types to save, select the hierarchy, and output UCDB filename. Figure C-9 shows the *Coverage Save Window*.
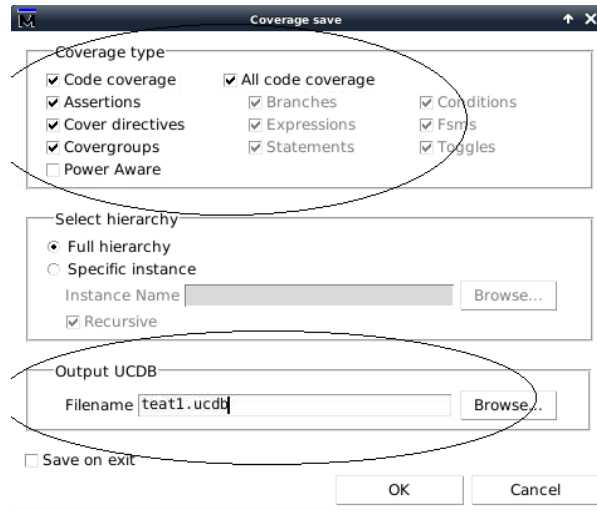
Figure C-9: Coverage Save Window

Step 5: Merging Coverage Test Data

When you have multiple sets of coverage data, from multiple tests of the same design, you can combine the results into a single UCDB by merging the UCDB files. The coverage data contained in the merged UCDB is a union of the items in the UCDBs being merged.

You can merge multiple .ucdb files (including a verification plan) from the Verification Browser window as follows:

1. Select the .ucdb file(s) to merge.

Figure C-10: Verification Management Browser Pane

2. Right-click over the file names and select Merge.

This displays the Merge Files Dialog Box, as shown in Figure C-11.

3. Fill in fields in the Merge Files dialog box, as required. A few of the more important, less intuitive fields are highlighted here.

- Set the Hierarchy Prefix: Strip Level / Add Prefix to add or remove levels of hierarchy from the specified instance or design unit.
- For Exclusion Flags, select AND when you want to exclude statements in the output file *only* if they are excluded in all input files. When OR is selected (default) a statement is excluded in the output merge file if the statement is excluded in any of the input files.
- Test Associated merge is the default Merge Level.

4. Select OK
This creates the merged file and loads the merged file into the Verification Browser window.
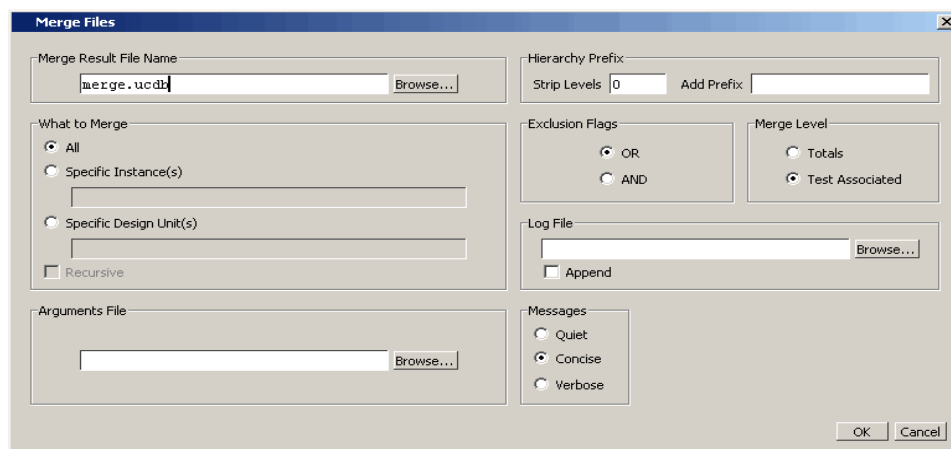
Figure C-11: Merge Dialog Box

The merge.ucdb file can be viewed by adding it to the Browser pane.

Step 6: Create code coverage reports

ModelSim allows you to create code coverage reports using the graphic interface or by entering commands at the command line. ModelSim allows you to create code coverage reports using the graphic interface.

After each individual run, report can be generated by selecting **Tools > Coverage Report > Text** from the Main window menu. Figure C-12 shows the Coverage Text Report Pane.
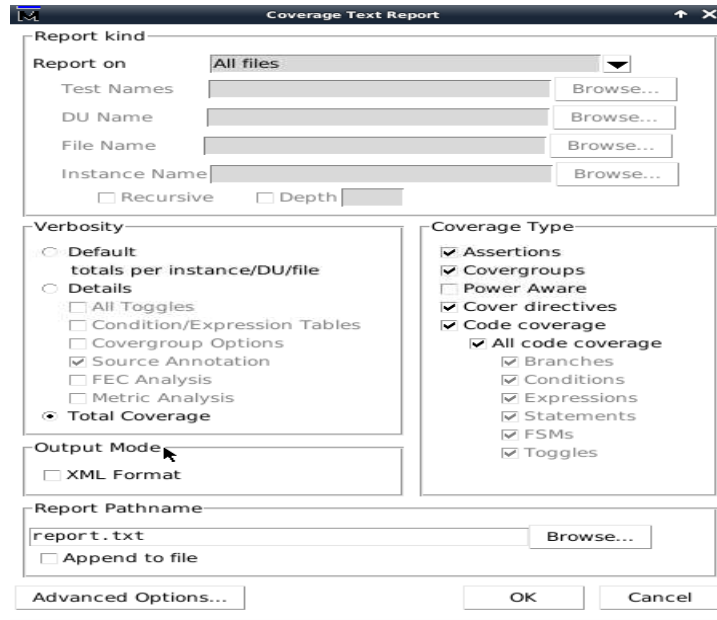
Figure C-12: Coverage Test Report Pane

This will generate a text report of different coverage's as shown in figure C-13.

Figure C-13: Coverage Report

In order to generate coverage report for merge.ucdb file, right click on the merge.ucdb in the *Verification Management Browser* and click on *Invoke CoverageView Mode*.

Now in the *File Browser* select the corresponding file, right click on it and under code coverage select *Code Coverage Reports*. This will generate the *Coverage Test Report Pane* as shown in fig C-12 and the remaining steps are same.