Tutorial
ModelSim SE

A.   Creating a Project

The goals for this lesson are:
- Create a project

A project is a collection entity for an HDL design under specification or test. Projects ease interaction with the tool and are useful for organizing files and simulation settings. At a minimum, projects have a work library and a session state that is stored in a .mpf file. A project may also consist of:

- o HDL source files or references to source files
- o other files such as READMEs or other project documentation
- o local libraries
- o references to global libraries

For more information about using project files, see the *ModelSim User's Manua*l.

1.   Start ModelSim:

from a Windows shortcut icon, from the Start menu

Upon opening ModelSim for the first time, you will see the **Welcome to ModelSim** dialog. (If this screen is not available, you can display it by selecting **Help > Welcome Menu** from the Main window.)

2.   Select **Create a Project**

from the Welcome dialog, or **File > New > Project** (Main window). In the **Create Project** dialog box, enter "test" as the Project Name and select a directory where the project file will be stored. Leave the Default Library Name set to "work."
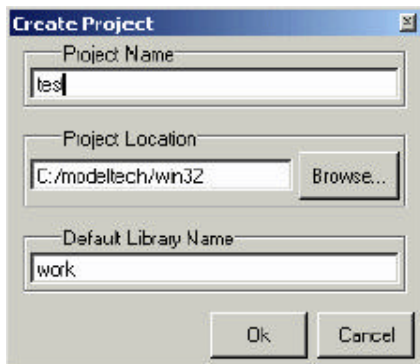
Upon selecting OK, you will see a blank Project tab in the workspace area of the Main window and the **Add Items to the Project** dialog.
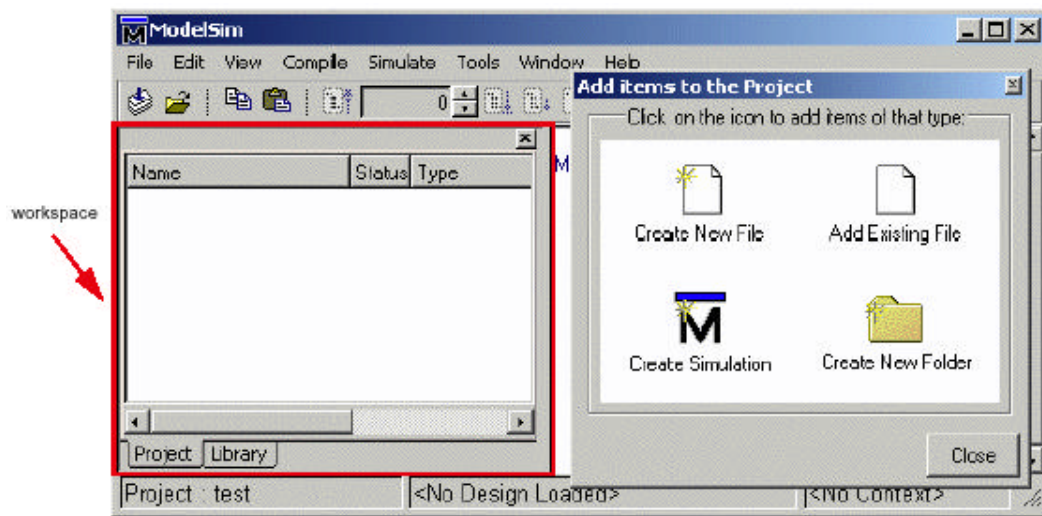
3.   The next step is to add the files that contain your design units.

Click **Add Existing File** in the **Add Items to Project** dialog. For this exercise, we'll add two Verilog files. Click the **Browse** button in the Add file to Project dialog box and open the examples directory in your ModelSim installation. Select *tcounter.v* and *counter.v*. Select **Reference from current location** and then click OK.
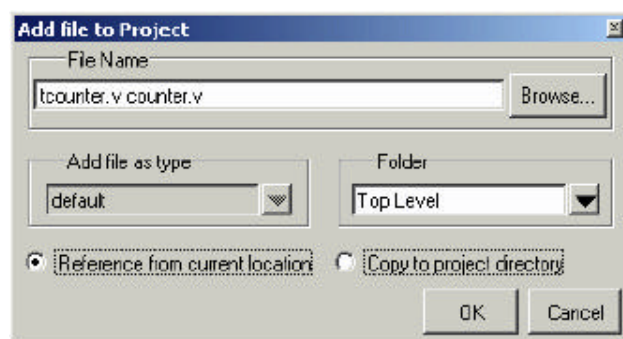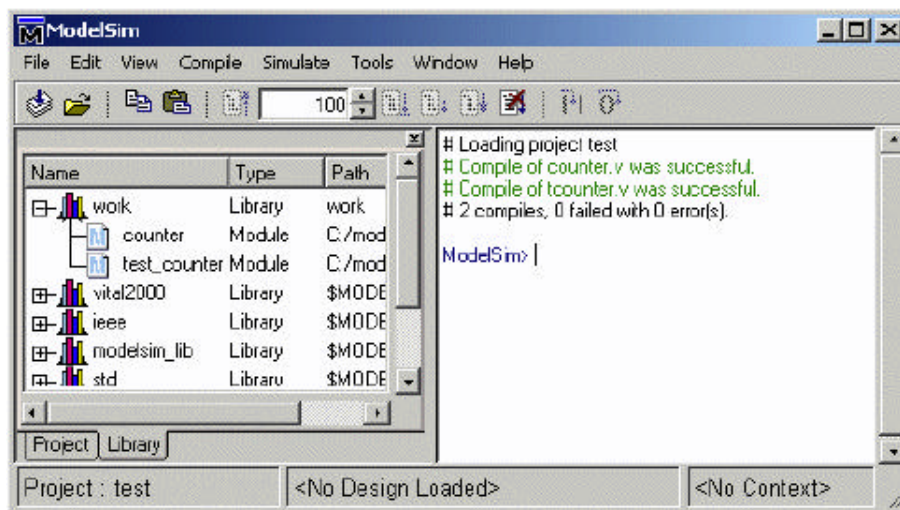

4.   Click your right mouse button
in the Project page and select **Compile > Compile Al**l.
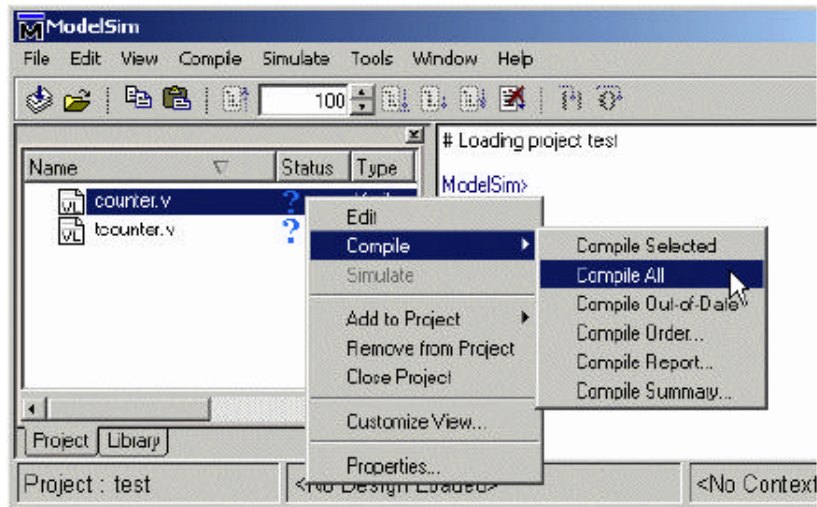
5.   The two files are compiled.
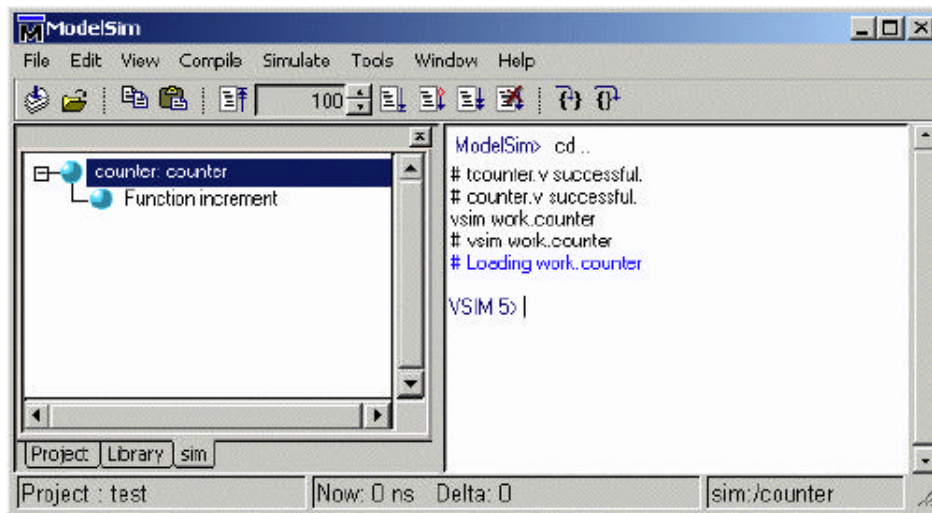
Click on the Library tab and expand the *work* library by clicking the "+" icon. You'll see the compiled design units listed.

6.   Load one of the design units

The last step in this exercise is to load one of the design units. Double-click *counter* on the Library page. You'll see a new page appear in the Workspace that displays the structure of the *counter* design unit.

At this point, you would generally run the simulation and analyze or debug your design. We'll do just that in the upcoming lessons. For now, let's wrap up by ending the simulation and closing the project. Select **Simulate > End Simulation** and confirm that you want to quit simulating. Next, select **File > Close > Projec**t, confirm that you want to close the project, and select **Yes** to update your project file with the changes you made during this session.

Note that a *test.mpf* file has been created in your working directory. This file contains information about the project *test* that you just created. ModelSim will open this project automatically the next time you invoke the tool.

B.   Basic Verilog simulation

The goals for this lesson are:
- Compile a Verilog design
- List signals in the design
- Examine the hierarchy of the design
- Simulate the design
- Change the default run length
- Set a breakpoint

The project feature covered in A executes several actions automatically such as creating and mapping work libraries. In this part we will go through the entire process so you get a feel for how ModelSim really works.

1.   Compiling the design

a)   Create and change to a new directory to make it the current directory.
You can make the directory current by invoking ModelSim from the new directory or by using the **File > Change Directory** command from the ModelSim Main window.

b)   Copy the Verilog files (files with ".v" extension)
from the \<*install_dir*>*modeltech**examples* directory into the current directory.
Before you can compile a Verilog design, you need to create a design library in the new directory. Since ModelSim is a compiled Verilog simulator, it requires a target design library for the compilation. ModelSim can compile both VHDL and Verilog code into the same library if desired.

c)   Invoke ModelSim:
from a Windows shortcut icon, from the Start menu

Click **Close** if the Welcome dialog appears.

d)   Create library
Before you compile any HDL code, you'll need a design library to hold the compilation results. To create a new design library, make this menu selection in the Main window: **File > New > Librar**y.
Make sure **Create: a new library and a logical mapping to it** is selected.
Type "work" in the Library Name field and then select **O**K. This creates a subdirectory named *work* - your design library - within the current directory. ModelSim saves a special file named *_info* in the subdirectory.
(PROMPT: vlib work vmap work work)
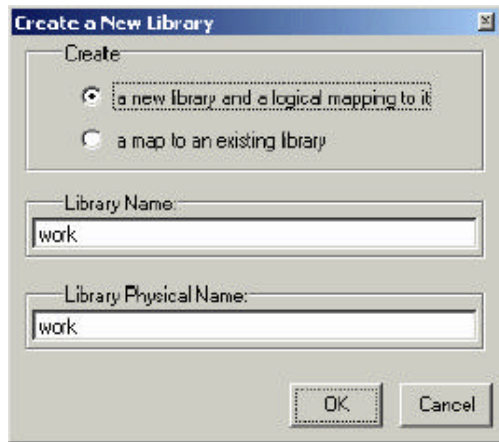
In the next step you'll compile the Verilog design. The example design consists of two Verilog source files, each containing a unique module. The file *counter.v* contains a module called **counter**, which implements a simple 8-bit binary up-counter. The other file, *tcounter.*v, is a testbench module (**test_counter**) used to verify **counter**.

Under simulation you will see that these two files are configured hierarchically with a single instance (instance name **du**t) of module **counter** instantiated by the testbench.

You'll get a chance to look at the structure of this code later. For now, you need to compile both files into the **work** design library.

Note: Do not create a Library directory using Windows commands, because the *_info* file will not be created. Always use the File menu or the **vlib** command from either the ModelSim or UNIX/DOS prompt.)

       e)   *Compile*

Compile the *counter.*v, and *tcounter.*v files into the **work** library by selecting **Compile > Compile** from the menu.

(PROMPT: vlog counter.v tcounter.v)

This opens the Compile HDL Source Files dialog box.
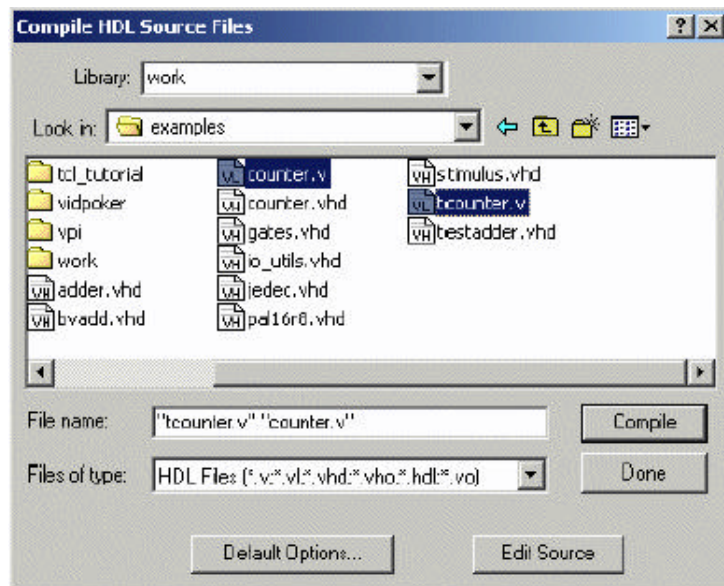
Select *counter.v* and *tcounter.v* (use Ctrl + click) and then choose **Compile** and then **Don**e.

Note: The order in which you compile the two Verilog modules is not important (other than the source-code depende ncies created by compiler directives). So it doesn't matter here if you choose to compile *counter.v* before or after *tcounter.* v.

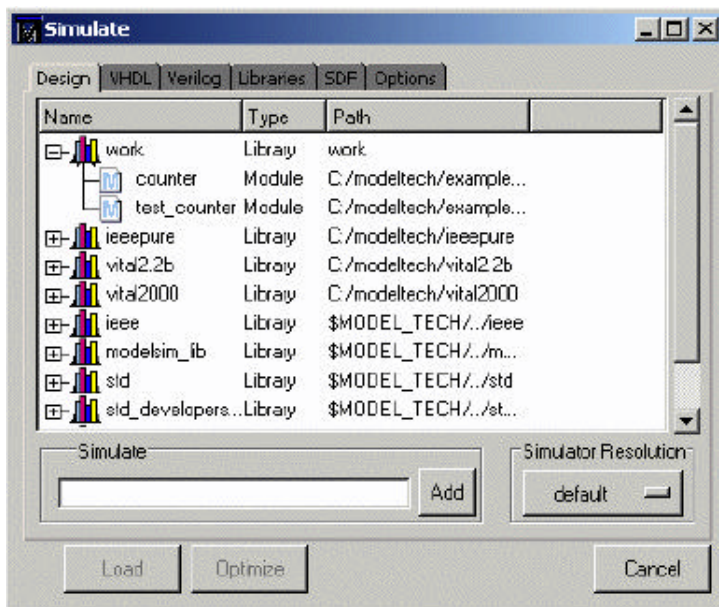## 2.   Loading the design

      a)   Load the design
 by selecting **Simulate > Simulate** from the menu:

(PROMPT: vsim test_counter)

The Simulate dialog appears. Click the "+" sign next to 'work' to see the **counter** and **test_counter** design units. (You won't see this dialog box if you invoke **vsim** with *test_counter* from the command line.)


The Simulate dialog allows you to select a design unit to load from the specified library. You can also select the resolution limit for the simulation. The default resolution is 1 ns.
Select **test_counter** and click **Load** to accept these settings.

      b)   Bring up the Signals, Source, and Wave windows

by entering the following command at the VSIM prompt within the Main window : view signals source wave
(Main MENU: View > <window name>)

      c)   Add signals
Now let's add signals to the Wave window with ModelSim's drag and drop feature. In the Signals window, select **Edit > Select All** to select the three signals. Drag the signals to either the pathname or the values pane of the Wave window.
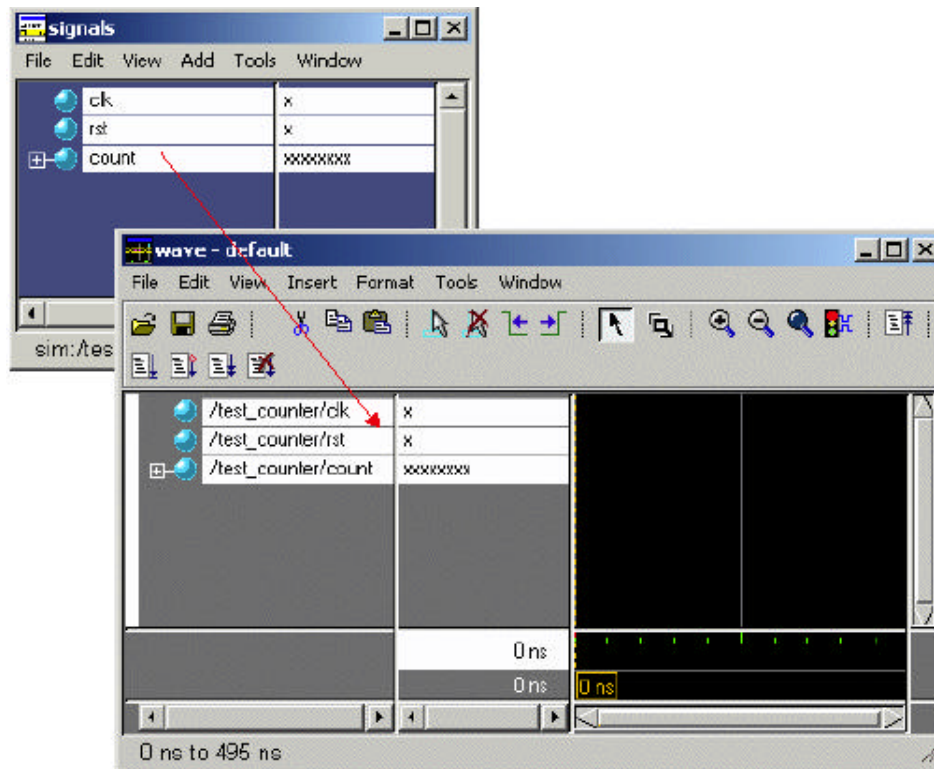
HDL items can also be copied from one window to another (or within the Wave and List windows) with the **Edit > Copy** and **Edit > Paste** menu selections.

d)   *Structure pane*

You may have noticed when you loaded the design in Step 1 that a new tab appeared in the workspace area of the Main window.
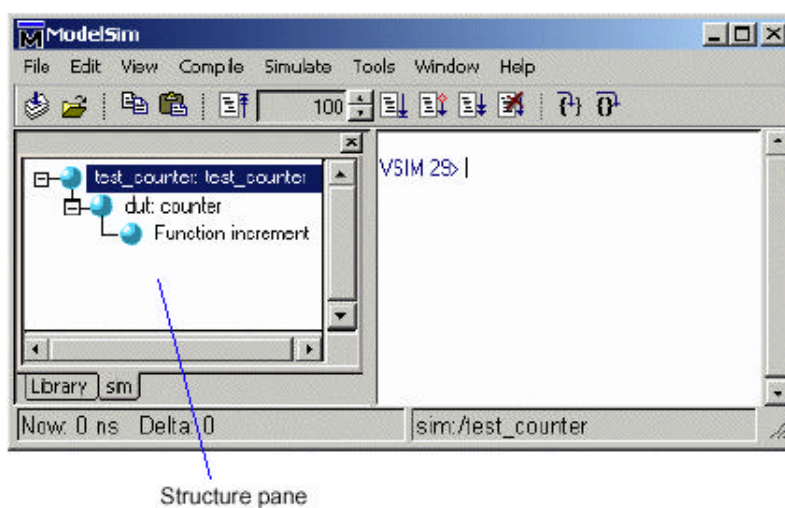
Structure pane

The Structure tab shows the hierarchical structure of the design. By default, only the top level of the hierarchy is expanded. You can navigate within the hierarchy by clicking on any line with a "+"

(expand) or "-" (contract) symbol. The same navigation technique works anywhere you find these symbols within ModelSim.
By clicking the "+" next to **dut: counter** you can see all three hierarchical levels: **test_counter**, **counter** and a function called **increment**. (If **test_counter** is not displayed you simulated **counter** instead of **test_counter**.)

Click on **Function increment** and notice how other ModelSim windows are automatically updated as appropriate. Specifically, the Source window displays the Verilog code at the hierarchical level you selected in the Structure window, and the Signals window displays the appropriate signals. Using the Structure tab in this way is analogous to scoping commands in interpreted Verilogs.
For now, make sure the **test_counter** module is showing in the Source window by clicking on the top line in the Structure pane.

3.   Running the simulation

Now you will exercise different Run functions from the toolbar.
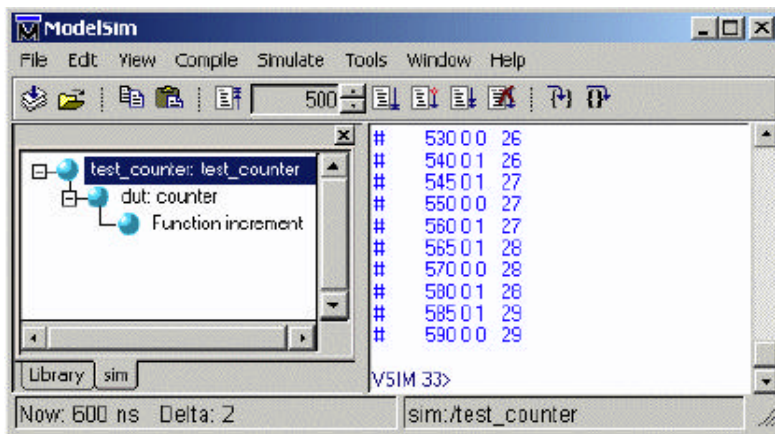
a)   Run
Select the **Run** button on the Main window toolbar. This causes the simulation to run and then stop after 100 ns (the default simulation length).
(PROMPT: run) (MENU: Simulate > Run > Run 100 ns)

b)   Specify run length
Next change the run length to 500 on the **Run Length** selector and select the **Run** button again.

Now the simulation has run for a total of 600ns (the default 100ns plus the 500 you just asked for). The status bar at the bottom of the Main window displays this information.

c)   Run until specified time

The last command you executed (**run 50**0) caused the simulation to advance for 500ns. You can also advance simulation to a specific time. Type: run @ 3000
This advances the simulation to time 3000ns. Note that the simulation actually ran for an additional 2400ns (3000 - 600).

d) Run until breakpoint

Now select the **Run -All** button from the Main window toolbar. This causes the simulator to run until the stop statement in *tcounter.v*.
(PROMPT: run -all) (MENU: Simulate > Run > Run -All)
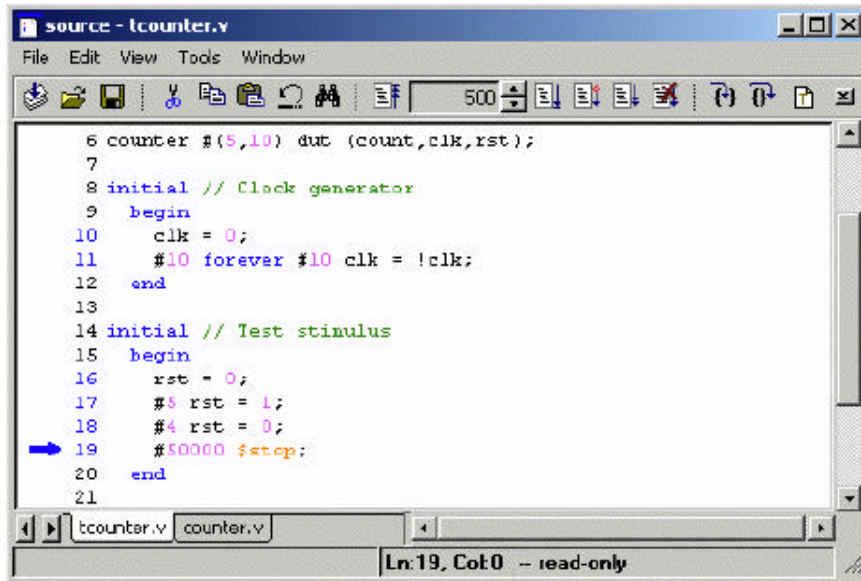
You can also use the **Break** button to interrupt a run.

(MENU: Simulate > Break)

4.   Debugging

Next we'll take a brief look at an interactive debugging feature of the ModelSim environment.

a) Set a breakpoint

Let's set a breakpoint at line 30 in the *counter.v* file (which contains a call to the Verilog function increment). To do this, select **dut: counter** in the Structure pane of the Workspace. Move the cursor to the Source window and scroll the window to display line 30. Click on or near line number 30 to set a breakpoint. You should see a red dot next to the line number where the breakpoint is set.
The breakpoint can be toggled between enabled and disabled by clicking it. When a breakpoint is disabled, the dot appears open. To delete the breakpoint, click the line number with your right mouse button and select **Remove Breakpoin**t.
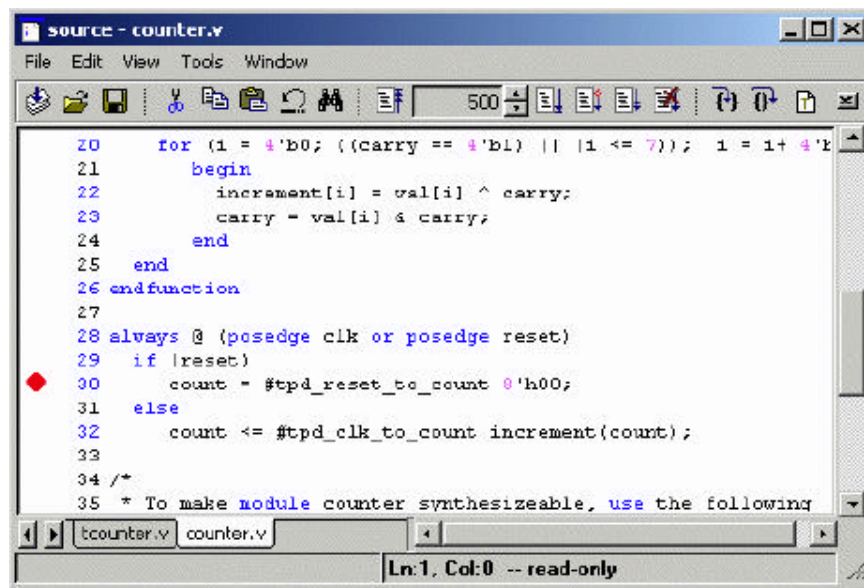
b) Restart

Select the **Restart** button to reload the design elements and reset the simulation time to zero.
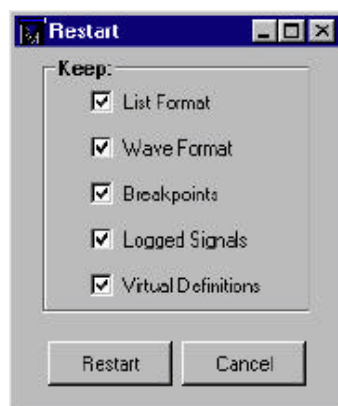

(Main MENU: Simulate > Run > Restart) (PROMPT: restart)

Note: Breakpoints can be set only on executable lines, denoted by blue line numbers.

Make sure all items in the Restart dialog box are selected, then click **Restar**t.

Select the **Run -All** button to re-start the simulation run.

(PROMPT: run -all) (Main MENU: Simulate > Run > Run -All)

When the simulation hits the breakpoint, it stops running, highlights the line with an arrow in the
Source window, and issues a Break message in the Main window.

c)   Reading signal values
When a breakpoint is reached, typically you will want to know one or more signal values. You have
several options for checking values :
- look at the values shown in the Signals window
- hover your mouse pointer over the *count* variable in the Source window and a "balloon" will
  pop up with the value

- select the *count* variable in the Source window, right-click it, and select Examine from the context menu
- use the **examine** command to output the value to the Main window transcript: examine count

d) Step

Let's move through the Verilog source functions with ModelSim's Step command.
Click **Step** on the toolbar.
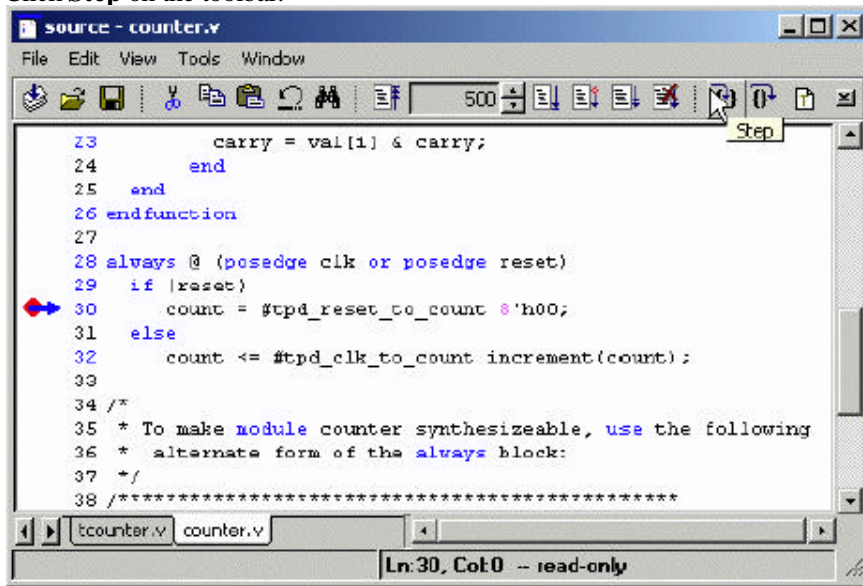
This command single-steps the debugger.

e) Hands-on

Experiment by yourself for awhile. Set and clear breakpoints and use the Step and Step Over commands until you feel comfortable with their operation. When you're done, quit the simulator by entering the command: quit –force

C. Exhaustive simulation using verilog

1. Test bench for the counter module

As you have seen, along with the actual module is another Verilog file, called a test bench for this module. This module creates the test vector that enables us to check that the program actually performs the right function.
In this example, the test bench is pretty short, since the only input is the clock, but other systems might have more inputs and you might want to simulate all possible realizations of these inputs. (For example, for a 3-to8 decoder, you want to generate all possible 3 bit inputs)
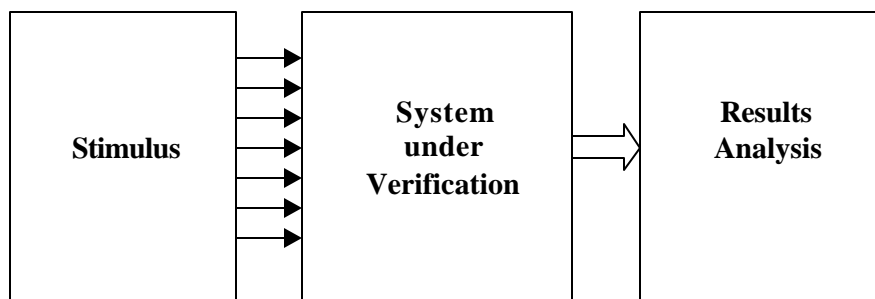
We will now analyse the structure of the program to understand the different commands.

----
module test_counter;

reg clk, rst;
wire [7:0] count;

   Regular module declaration : there is no inputs nor outputs
----

   *In th is code fragment, the stimulus and response capture are going to be coded using
   a couple of initial blocks. An initial block can contain sequential statements that can
   be used to describe the behaviour of signals in a test bench. In the Stimulus initial
   block, we need to generate waveforms on the clock and reset inputs. Thus:*

initial // Clock generator
 begin
  clk = 0;
  #10 forever #10 clk = !clk;
 end

initial // Test stimulus
 begin
  rst = 0;
  #5 rst = 1;
  #4 rst = 0;
  #50000 $stop;
 end

   *Everytime there is a pound sign followed by a number n, it means that the simulator
   advances by n times of simulation and then does whatever assignment is specified.
   The $stop command is a Verilog built-in command that stops the simulation. In this
   case, the simulation will stop after 50009 simulation times.*

   *Concerning the clock, its behavior is easy to understand. At the beginning, its value is
   set to zero ; then, every 10 seconds and for ever, its value is inverted.*
----
counter#(5,10) dut (count,clk,rst);

   The test bench has to instantiate an instance of the module counter.
----
initial

```
$monitor($stime,, rst,, clk,,, count);
```

The Response initial block can be described very easily in Verilog as we can benefit from a built-in Verilog system task. Indeed, $monitor is a system task that is part of the Verilog language. Its mission in life is to print values to the screen. The values it prints are those corresponding to the arguments that you pass to the task when it is executed. The $monitor task is executed whenever any one of its arguments changes, with one or two notable exceptions :

$time is a system function (as opposed to a system task). It returns the current simulation time. In the above example, $time is an argument to $monitor. However, $time changing does not cause $monitor to execute — $monitor is clever enough to know that you wouldn't really want to print to the screen the values of all of the arguments every time the simulation time changed.

The succession of two commas in the argument list ensures that a space is printed to the screen after the value of $time each time $monitor is executed. This is a simple method of formatting the screen output.

Finally we come to the signal arguments themselves. Each time one of these signals changes value, $monitor will execute. When $monitor executes it will print all of the argument values to the screen, including $time. This is the output created by $monitor in our counter test bench:

```
#      0 0 0   x
#      5 1 0   x
#      9 0 0   x
#     15 0 0   0
#     20 0 1   0
#     25 0 1   1
#     30 0 0   1
#     40 0 1   1
#     45 0 1   2
#     50 0 0   2
#     60 0 1   2
#     65 0 1   3
#     70 0 0   3
#     80 0 1   3
#     85 0 1   4
#     90 0 0   4
```

endmodule

A last note on this test bench ; to access and monitor a signal which is in a lower hierarchical level, you just have to call the signal by the module name, followed by a dot, followed by the signal name.