Chapter 4

Verilog Simulation

A HARDWARE DESCRIPTION LANGUAGE (HDL) is a programming language designed specifically to describe digital hardware. Typical HDLs look somewhat like software programming languages in terms of syntax, but have very different semantics for interpreting the language statements. Digital hardware, when examined at a sufficient level of detail, is described as a set of Boolean operators executing concurrently. These Boolean operators may be complex Boolean functions, refined into sets of Boolean gates (NAND, NOR, etc.), or described in terms of the individual transistors that implement the functions, but fundamentally digital systems operate through the combined effect of these Boolean operators executing at the same time. There may be a few hundred gates or transistors, or there may be tens of millions, but because of the concurrency inherent in these systems an HDL used to describe these systems must be able to support this sort of concurrent behavior. Of course, they also support "software-like" sequential behavior for high-level modeling, but they must also support the very concurrent behavior of the fine-grained descriptions.

To enable this sort of behavior, HDLs are typically executed through event-driven simulators. An event-driven simulator uses a notion of simulation time and an event-queue to schedule events in the system being described. Each HDL construct (think of a construct as modeling a single gate, for example, but it could be much more complex than that) has inputs and outputs. The description of the construct includes not only the function of the construct, but how the construct reacts over time. If a signal changes then the event-queue looks up which constructs are affected by that change to know which code to run. That code may produce a new event at the construct's output, and that output event will be sent to the event queue so that the output event will happen sometime in the future. When simulation time advances to the correct value the output event occurs which may cause other activity in the described system. The event-queue is the central controlling structure that enables the HDL program to behave like the hardware that it is describing. So, although you may think of "running a program" written in a software programming language, it's more correct to think of "running a simulation" when executing an HDL program.

Verilog is one of the two most widely used Hardware Description Languages with VHDL being the other main HDL in wide use today. Much of the simulation information described in this chapter will translate reasonably easily to VHDL simulators, but for this text I'll stick with Verilog. The choice of using Verilog is somewhat arbitrary as the two languages are quite similar in their ability to describe digital systems. In very general terms, many designers find Verilog syntax simpler and easier to use, at the expense of VHDL's richer type system, but both HDLs are used on "real" designs.

One reason to choose Verilog is that some of the tools in this CAD flow, place and route in particular, require Verilog as an input specification.

Verilog program execution (program simulation) requires a Verilog simulator that implements the event-driven semantics of the language. You will also need a method of sending inputs to your program, and a means of checking that the outputs of your Verilog program are correct. This is usually accomplished using a second Verilog program known as a *testbench* or *testfixture*. This is similar to how integrated circuits are tested. In that case the chip to be tested is called the Device Under Test (DUT), and the DUT is connected to a testbench that drives the DUT inputs and records and compares the DUT outputs to some expected values. If we use the same terminology for our Verilog simulations, then the Verilog program that you want to run would be the equivalent of the DUT, and you need to write a testbench program (also in Verilog) to drive the program inputs and look at the program outputs.

This general scheme is shown in Figures 4.1 and 4.2. These figures show the test environment that is created by the Composer system, but they are a good general testbench format. There is a top-level module named **test** that is simulated. This module includes one instance of the DUT. In this case the DUT is our twoBitAdd module from Chapter 3 and the instance name is **top**. It also includes testfixture code in a separate file named **testfixture.verilog**. In this file is an **initial** block that has the testfixture code. Example testfixture code will be seen in the following sections of this Chapter. Note that the module **test** defines all the inputs to the DUT as **reg** type, and outputs from the DUT as **wire** type. This is because the testfixture wants to set the value of the DUT inputs and look at the value of the DUT outputs.

This text does not include a tutorial on the Verilog language. There are lots of good Verilog overviews out there, including Appendix A of the class textbook *CMOS VLSI Design: A Circuits and Systems Perspective, 3rd ed* by Weste and Harris [1]. I'll show some examples of Verilog code and testbench code, but for a basic introduction see Appendix A in that book, or any of the good Verilog introductions on the web.

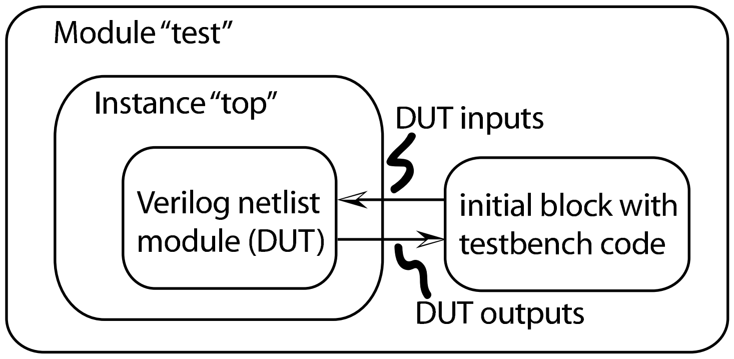

Module "test"
Instance "top"
Verilog netlist module (DUT)
DUT inputs
initial block with testbench code
DUT outputs

Figure 4.1: The simulation environment for a Verilog program (DUT) and testbench

```
‘timescale 1ns / 100ps
module test;

wire  Cout;
reg  Cin;
wire [1:0]  Sum;
reg [1:0]  A;
reg [1:0]  B;

twoBitAdd top(Cout, Sum, A, B, Cin);

‘include "testfixture.verilog"
endmodule
```

Figure 4.2: Verilog code for a DUT/testbench simulation environment

There are three Verilog simulators of interest to this CAD flow. They are:

Verilog-XL: This is an interpreted simulator from Cadence. Interpreted means that there is a run-time interpreter executing each Verilog instruction and communicating with the event-queue. This is an older simulator and is the reference simulator for the Verilog-1995 standard. Because it is the reference simulator for that standard it has not been updated to use some of the more modern features of Verilog, and because it is interpreted it is not the fastest of the simulators. But, it is well integrated into the Cadence system and is the default Verilog simulator for many tools.

NC_Verilog: This is a compiled simulator from Cadence. This simulator compiles the Verilog code into a custom simulator for that Verilog program. It converts the Verilog code to a C program and compiles that C program to make the simulator. The result is that it takes a little longer to start up (because it needs to translate and compile), but the resulting compiled simulator runs much faster than the interpreted Verilog-XL. It is also compatible with a large subset of the Verilog-2000 standard and is being actively updated by Cadence to include more and more of those advanced features.

VCS: This is a compiled simulator from Synopsys. It is not integrated into the Cadence tools, but is integrated to some extant with the Synopsys tools so it is useful if you spend more time in the Synopsys portion of the design flow before using the back-end tools from Cadence. It is also a very fast simulator like NC_Verilog.

4.1 Verilog Simulation of Composer Schematics

The simulators from Cadence are integrated with the Composer schematic capture tool. This means that if there are Verilog models for the cells you use in your schematic, you can simulate your schematics without leaving the dfII environment. All the cell libraries that we will use have Verilog models so all schematic simulation will be done through a Verilog simulator.

In order to do this you need a Verilog version of your schematic. That is, you need walk the schematic hierarchy and generate Verilog code that captures the module connectivity of the schematic. Whenever a node is encountered whose behavior can be described with a piece of Verilog code, you need to insert that Verilog code. The result is a hierarchical Verilog program that captures the functionality of the schematic. This process is known as *netlisting*, and the result is a structural Verilog description that is also sometimes called a *netlist*. According to Figure 4.1 this netlist could also be known as the DUT. Once you have the DUT code, you need to write testbench code to communicate with the DUT during simulation.

Fortunately the Composer-Verilog integration environment will generate a simulatable netlist for you from the schematic, and also generate a template for a testbench file. The netlisting process walks through your hierarchical schematic and generates a program that describes the hierarchy. If the netlister encounters a **behavioral** cell view, that view contains Verilog code that describes that module's function so the code in the **behavioral** view is added to the netlist. If a schematic uses transistors from the **NCSU_Analog_Parts** or **UofU_Analog_Parts** libraries, those transistors are replaced with Verilog transistor models. The Verilog transistor primitives are built into the Verilog language and simulate the transistors as switches. We use the **cmos_sch** view to signal to the netlister that this is a leaf cell that contains only transistors.

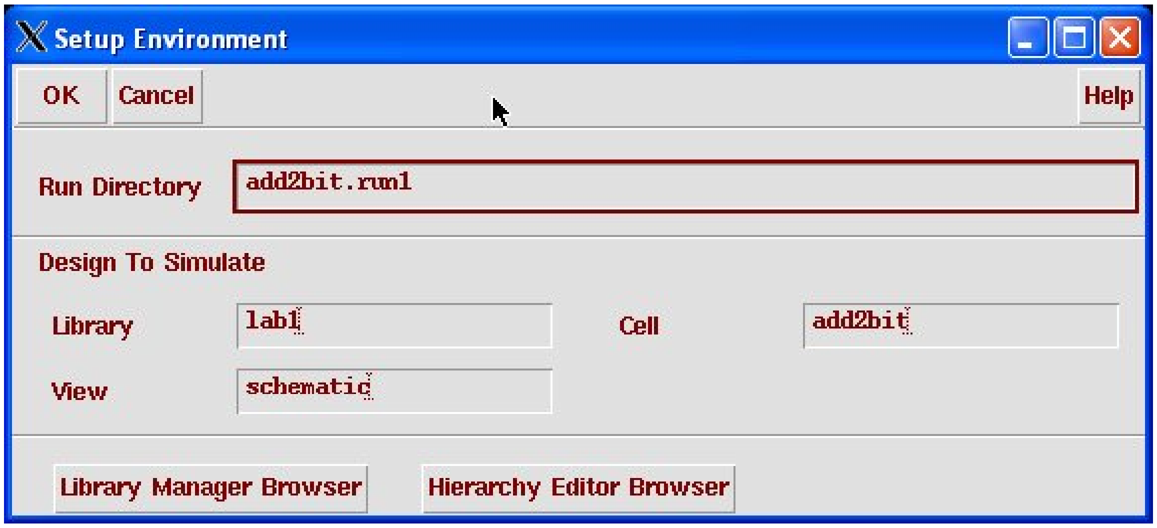
These two different description techniques for leaf cells (**behavioral** and transistor **cmos_sch**) can be mixed in a single schematic, and in fact if the leaf cells have both **behavioral** (Verilog) and **cmos_sch** (transistor) views you can choose which low-level behavior is simulated by manipulating the netlisting procedure in Composer. This is useful because each type of simulation has its own advantages and disadvantages. Behavioral modeling can simulate more quickly, and allows back-annotation of timing from other tools like synthesis tools though a Standard Delay Format (sdf) file (described in more detail in Section 4.4 and in Chapter 8). Switch level modeling can be a more accurate simulation of the low level details of the circuit's operation and can expose problems that are not visible in the more high-level behavioral simulation.

4.1.1 Verilog-XL: Simulating a Schematic

As an example of simulating a schematic using Verilog-XL we'll use the two-bit adder from Chapter 3. To start Verilog-XL simulation you can use the CIW window by going to **Tools** → **Verilog Integration** → **Verilog-XL...**. The **Setup Environment** window appears in which the Run Directory, Library, Cell and View fields need to be filled. Press OK.

Or (the much easier way) open up the Composer schematic of the two-bit adder using the library browser and in the Composer schematic editing window, select **Tools** → **Simulation** → **Verilog-XL**. The **Setup Environment** window appears with all the fields filled. The **Run Directory** can be changed or left as default <***designname***>**.run1**. A dialog box for simulation of the two-bit adder from Chapter 3 is shown in Figure 4.3. Press **OK**.

It is very important to have a separate run directory for each different design, but you can keep the same run directory for different simulations of the same design.



Run Directory: add2bit.run1

Design To Simulate

Library: lab1

Cell: add2bit

View: schematic

Figure 4.3: Dialog Box for Initializing a Simulation Run Directory

This will initialize the Verilog-XL simulator and bring up the Verilog-XL control window as shown in Figure 4.4. This is the window from which the simulation can be invoked. Most simulation activities can be controlled from the menus or from the widget icons on the left side of the panel. Hovering your mouse over those widgets will give you an idea of what they are. This manual won't go over all of them, but some playing around should reveal what most of them do.

Before starting simulation the environment needs to be set up and an input stimulus file for the circuit (a testbench, also called a *test fixture* by Verilog-XL) needs to be created.

Setting up the Simulation Environment

Select **Setup** → **Record Signals...**. In the **Record Signal Options** window which appears you can change the signals that will be recorded during the simulation from **Top Level Primary I/O** to **All Signals** if you like. Saving only the top level I/O signals saves a lot of time and disk space, but only records the signals at the very top level of your circuit. Thus, you can't probe or observe any signals further down in the circuit hierarchy. If you want to be able to see circuit values inside the sub-circuits you should save **All Signals**.

Note: To make changes to the **Record Signal Options** later on, make sure that the interactive simulation is stopped. If it is not stopped then select **Simulation** → **Finish Interactive** or press the widget with the black square

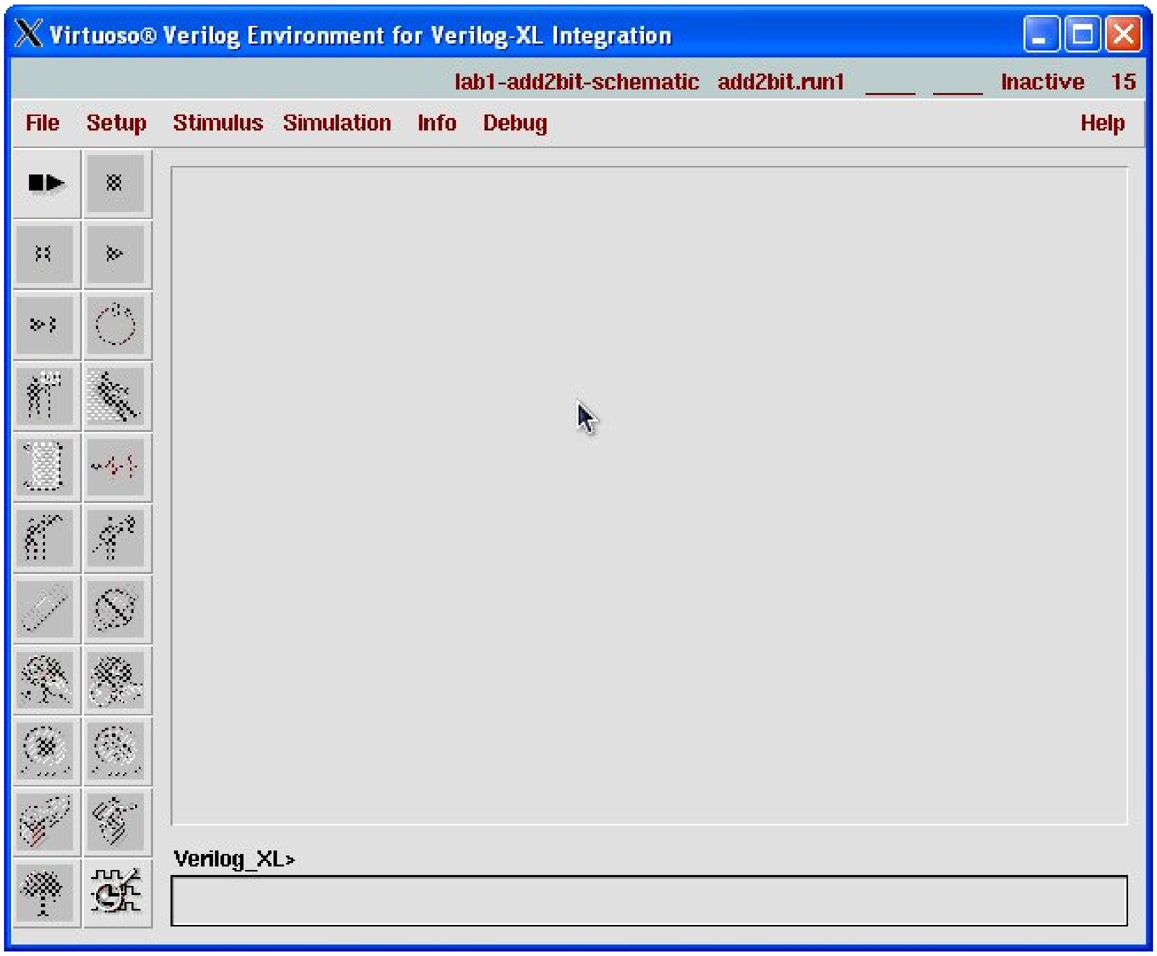

Virtuoso® Verilog Environment for Verilog-XL Integration
lab1-add2bit-schematic add2bit.run1 Inactive 15
File Setup Stimulus Simulation Info Debug Help
Verilog_XL>

Figure 4.4: The Initial Verilog-XL Simulation Control Window

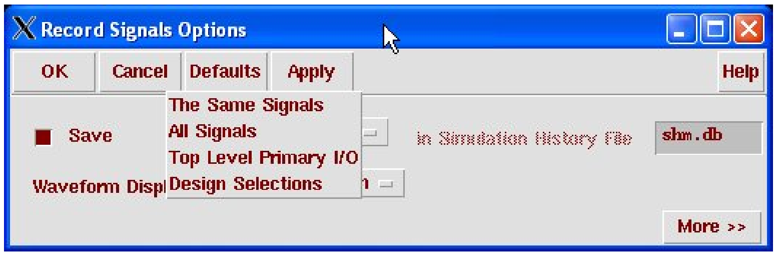
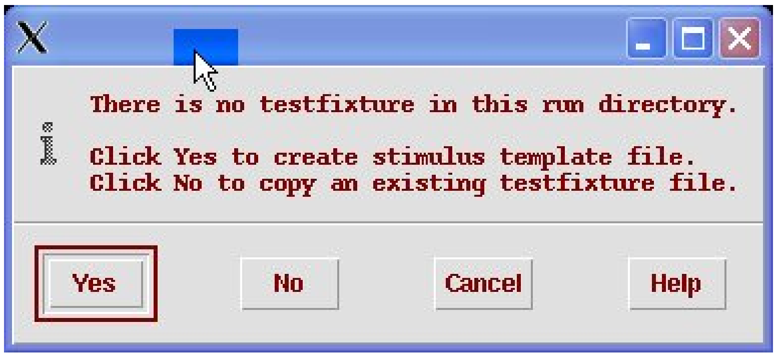


Figure 4.5: The Record Signals Dialog Box


There is no testfixture in this run directory.

Click Yes to create stimulus template file.
Click No to copy an existing testfixture file.

Figure 4.6: Dialog to Create a New Test Fixture Template

which stops the current simulation.

Defining the Test Fixture (testbench)

Select **Stimulus** → **Verilog...** and a prompt window appears asking if you wish to create a template (Figure 4.6). Select **Yes**.

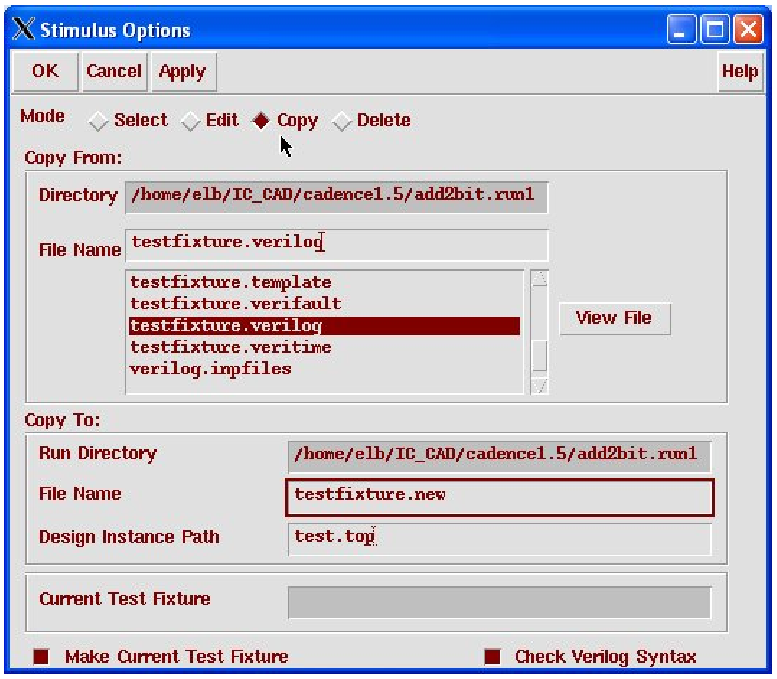
All the steps in setting up the test fixture file must be completed before starting Interactive Simulation. If interactive simulation is already started, select **Simulation** → **Finish Interactive** or press the stop (black square widget) button.

In the Stimulus Options window which appears (See Figure 4.7) select **copy**. In the **Copy From:** frame, select the **File Name** from the list as **testfixture.Verilog**. The **File Name** in the **Copy To:** frame can be changed or left as the default **testfixture.new**. The **Design Instance Path** should not be changed from **test.top**. This is the DUT structure that is created by the Composer netlisting procedure. The uppermost cell is named **test**. This cell contains one instance of your top-level circuit which is given an instance name of **top**. The other component within the **test** module is your testbench code. The template for this testbench will be created when you press **Apply**.

*Verilog lets you access signals in a hierarchy using a "." between levels of the hierarchy. A wire **foo** inside the instance **top** could be accessed using **test.top.foo**, for example.*

Now select **Edit** mode and choose **testfixture.new** (or the file name you have given to the test fixture) from the **Field Name**. Select **Make Current Test Fixture** and **Check Verilog Syntax** and press **Apply** or **OK**.

The default editor (most likely emacs) will open up. Use this editor to type in the Verilog code that you want to use as your test fixture. Then save the test fixture and close the editor. The original test fixture template should look something like that in Figure 4.8. The interface signals are found on the symbol and repeated here in the test fixture template. Your test code



Mode: Select, Edit, [✓ Copy], Delete

Copy From:

Directory	/home/elb/IC_CAD/cadence1.5/add2bit.run1
File Name	testfixture.verilog

```
testfixture.template
testfixture.verifault
testfixture.verilog
testfixture.veritime
verilog.inpfiles
```

Copy To:

Run Directory	/home/elb/IC_CAD/cadence1.5/add2bit.run1
File Name	testfixture.new
Design Instance Path	test.top
Current Test Fixture	

[✓ Make Current Test Fixture] [✓ Check Verilog Syntax]

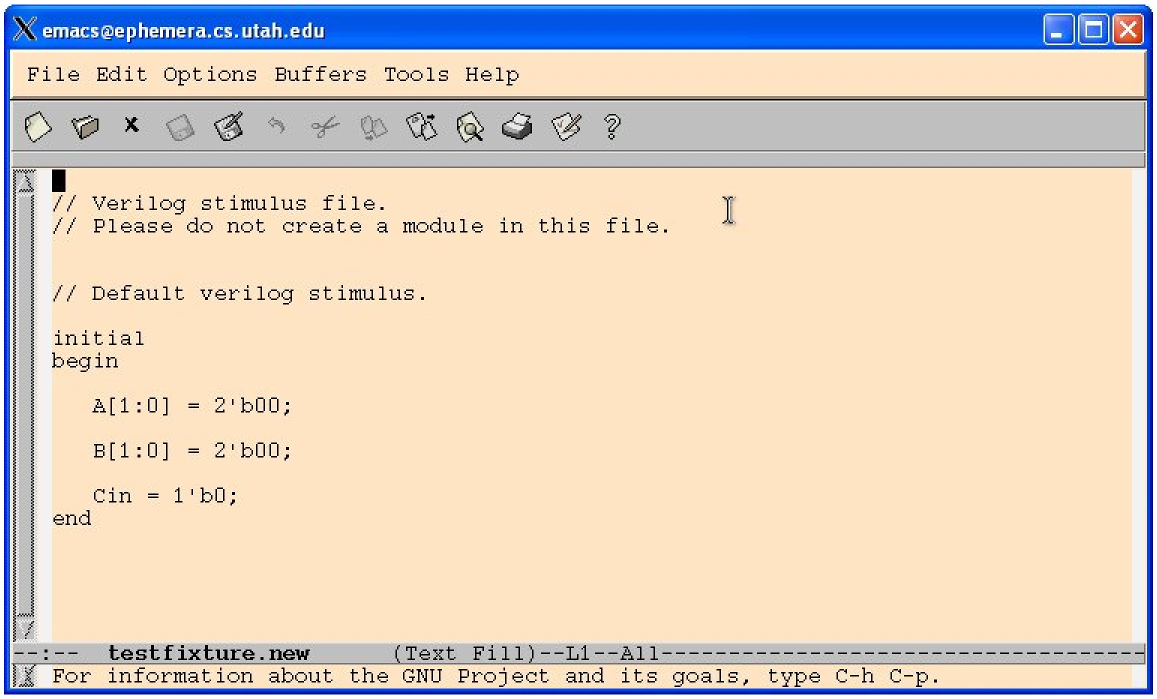
Figure 4.7: The Verilog-XL Stimulus Options Form

should go after the interface signal definitions and before the **end** statement. This piece of Verilog code is an **initial** block which is executed when you simulate the netlist assembled by Composer.

An example of a complete simple test fixture for the two bit adder is shown in Figure 4.9.

Some important things to notice about the test fixture in Figure 4.9 are:

- Verilog comments are either C-style with \**** and *****\\ bracketing the comment, or a pair of backslashes \\\\ denoting a comment to the end of the line. Please use comments in your test fixture!
- A **$display** statement is the equivalent of a C printf statement in a Verilog program. These can be very helpful so that you can see how things are progressing during your simulation.
- A good Verilog testbench *always* checks for the correct value of the outputs in the testbench and prints something if the values are not correct. You can see this in the **if** statements in the testbench code. These statements check for a correct value and print an error message if the value is not correct. *All your testbenches should be self-checking like this!* Waveforms are a great way to debug some things and for the


emacs@ephemera.cs.utah.edu
File Edit Options Buffers Tools Help

```
// Verilog stimulus file.
// Please do not create a module in this file.


// Default verilog stimulus.

initial
begin

   A[1:0] = 2'b00;

   B[1:0] = 2'b00;

   Cin = 1'b0;
end
```

```
--:--  testfixture.new      (Text Fill)--L1--All-----------------------------------
For information about the GNU Project and its goals, type C-h C-p.
```

Figure 4.8: Test Fixture Template for the Two Bit Adder

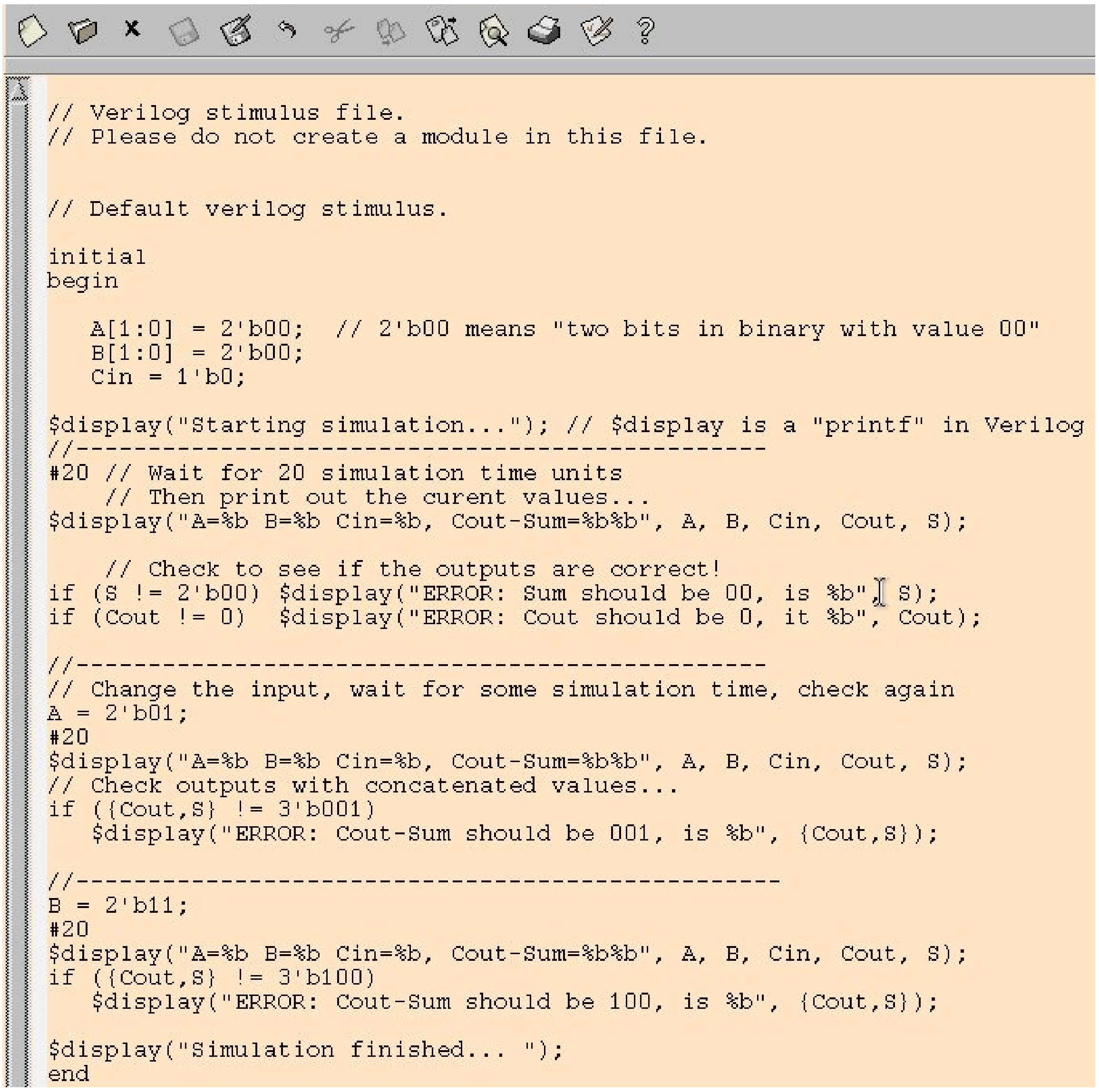
designer of the circuit to see what's going on, but they are really bad for checking whether a complete circuit is doing the right thing, or whether a change in the circuit has caused a change in behavior. Once you have a self-checking testbench you can make improvements and re-run the simulation to see if the behavior has changed.

The syntax check is checking for Verilog-1995 syntax

When you save the testbench code and exit the editor, the system will check the Verilog for correct syntax. If you don't pass the syntax check you'll need to reopen the testbench file and fix the errors. When you successfully dismiss the testfixture dialog box by selecting the new test fixture you are ready for simulation.

An example of a different type of testbench is shown in Figure 4.10. In this testbench loops are used to test the two bit adder exhaustively, and the checks are computed using Verilog to compute the answer instead of the testbench author writing down each result separately. Note that the integer variables required for the **for** loops are defined outside the **initial** block. Also note that there are a variety of syntax choices and shorthands available for referring to the input vectors.

Still another style of writing a testbench is shown in Figure 4.11. In this testbench the values of the inputs, and the values that should be checked



```
// Verilog stimulus file.
// Please do not create a module in this file.


// Default verilog stimulus.

initial
begin

   A[1:0] = 2'b00;  // 2'b00 means "two bits in binary with value 00"
   B[1:0] = 2'b00;
   Cin = 1'b0;

$display("Starting simulation..."); // $display is a "printf" in Verilog
//--------------------------------------------------
#20 // Wait for 20 simulation time units
    // Then print out the curent values...
$display("A=%b B=%b Cin=%b, Cout-Sum=%b%b", A, B, Cin, Cout, S);

    // Check to see if the outputs are correct!
if (S != 2'b00) $display("ERROR: Sum should be 00, is %b", S);
if (Cout != 0)  $display("ERROR: Cout should be 0, it %b", Cout);

//--------------------------------------------------
// Change the input, wait for some simulation time, check again
A = 2'b01;
#20
$display("A=%b B=%b Cin=%b, Cout-Sum=%b%b", A, B, Cin, Cout, S);
// Check outputs with concatenated values...
if ({Cout,S} != 3'b001)
   $display("ERROR: Cout-Sum should be 001, is %b", {Cout,S});

//---------------------------------------------------
B = 2'b11;
#20
$display("A=%b B=%b Cin=%b, Cout-Sum=%b%b", A, B, Cin, Cout, S);
if ({Cout,S} != 3'b100)
   $display("ERROR: Cout-Sum should be 100, is %b", {Cout,S});

$display("Simulation finished... ");
end
```

Figure 4.9: An Example Test Fixture for the Two Bit Adder

```
// Default Verilog stimulus.
integer i,j,k;
initial
begin

   A[1:0] = 2’b00;
   B[1:0] = 2’b00;
   Cin = 1’b0;

$display("Starting simulation...");

for(i=0;i<=3;i=i+1)
 begin
  for(j=0;j<=3;j=j+1)
   begin
    for(k=0;k<=1;k=k+1)
     begin
      #20
      $display("A=%b B=%b Cin=%b, Cout-Sum=%b%b", A, B, Cin, Cout, S);
      if ({Cout,S} != A + B + Cin)
          $display("ERROR: Cout-Sum should equal %b, is %b",
                    (A + B + Cin), {Cin,S});
      Cin=˜Cin; // invert Cin
     end
    B[1:0] = B[1:0] + 2’b01; // add the bits
   end
  A = A+1; // shorthand notation for adding
 end

$display("Simulation finished... ");
end
```

Figure 4.10: Another Test Fixture for the Two Bit Adder

for on the outputs, are held in external text files one value per line. These values might, for example, have been generated by some other application like Matlab or through some C, Java, python, or other program that you write to generate values to check for.

In this test fixture arrays are defined to hold the inputs for **A** and **B**, and for the results of the two bit addition. These Verilog arrays are initialized using the **$readmemb** command to read memory values in binary format (actually it's ASCII format with 1 and 0 as the digits, as opposed to **$readmemh** where the values in the test file are hex digits). The values in the data files are formatted with one array row of data on each line of the data file.

Once the data values are loaded into the arrays the simulation walks through all the values in the test arrays and check the answer against the value in the corresponding location in the **resultsarray**.

Of course, experienced Verilog programmers may have lots of additional ideas about how to write great test fixtures. These are just some ideas for how to think about your testbenches. Remember that all testbenches should be check for the correct answer in the testbench code!

Running the Simulation

Once you have a testbench that passes the syntax check, and you have set up the signals that you want to record, you can run the simulation. Start the Verilog simulation by selecting **Simulation** → **Start Interactive** or pressing the widget button with the square and right-facing **play** button (upper left of the widgets). This netlists your design, check the netlist for errors, and prepares the netlist and test fixture for simulation.

After you have done this once in the current *run directory* you will get a dialog box like that in Figure 4.13 asking if you want to re-netlist the design or use the old netlist. Usually you want to re-netlist at this point so that any changes you've made since that last time you simulated are updated in the simulation netlist.

The results for the netlisting in my example looks like that in Figure 4.14. Note that for each of the basic gates I used in my schematic the netlister chose **behavioral** views of those schematics. Later we'll see how to change the netlisting order so that the netlisting process will get the transistor switch-level views.

Note that once you've successfully netlisted the Verilog and initialized the simulator all the rest of the widgets that used to be grayed out become active. The Verilog window now looks like that in Figure 4.15.

Now that you're in "interactive" mode, you can run the simulation us-

```
// Default Verilog stimulus.
reg [1:0] ainarray [0:4]; // define memory arrays
reg [1:0] binarray [0:4]; // to hold input and result
reg [2:0] resultsarray [0:4];
integer i;

initial
begin

/* A simple Verilog test fixture for testing a 2-bit adder */

   $readmemb("ain.txt", ainarray);  // read values into
   $readmemb("bin.txt", binarray);  // arrays from files
   $readmemb("results.txt", resultsarray);

   A[1:0] = 2'b00; // initialize inputs
   B[1:0] = 2'b00;
   Cin = 1'b0;

   $display("Starting...");
   #10
   $display("A = %b, B = %b, Cin = %b, Sum = %b, Cout = %b",
            A, B, Cin, Sum, Cout);

   for(i=0; i<=4; i=i+1) // loop through all values in arrays
   begin
      A = ainarray[i]; // set the inputs
      B = binarray[i]; // from the memory arrays
      #10
      $display("A = %b, B = %b, Cin = %b, Sum = %b, Cout = %b",
               A, B, Cin, Sum, Cout);
      // check against results array
      if ({Cout,Sum} != resultsarray[i])
         $display("Error: Sum should be %b, is %b instead",
                  resultsarray[i],Sum);
   end
   $display("...Done");
   $finish;
end
```

Figure 4.11: A Test Fixture Using Values From External Files

```
01
10
11
00
01
```

(a) ain.txt

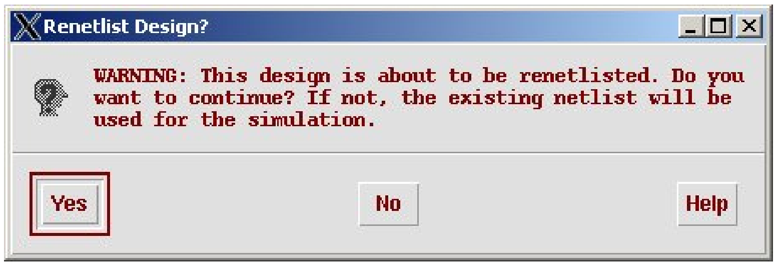
```
01
10
11
11
11
```

(b) bin.txt

```
010
100
110
011
100
```

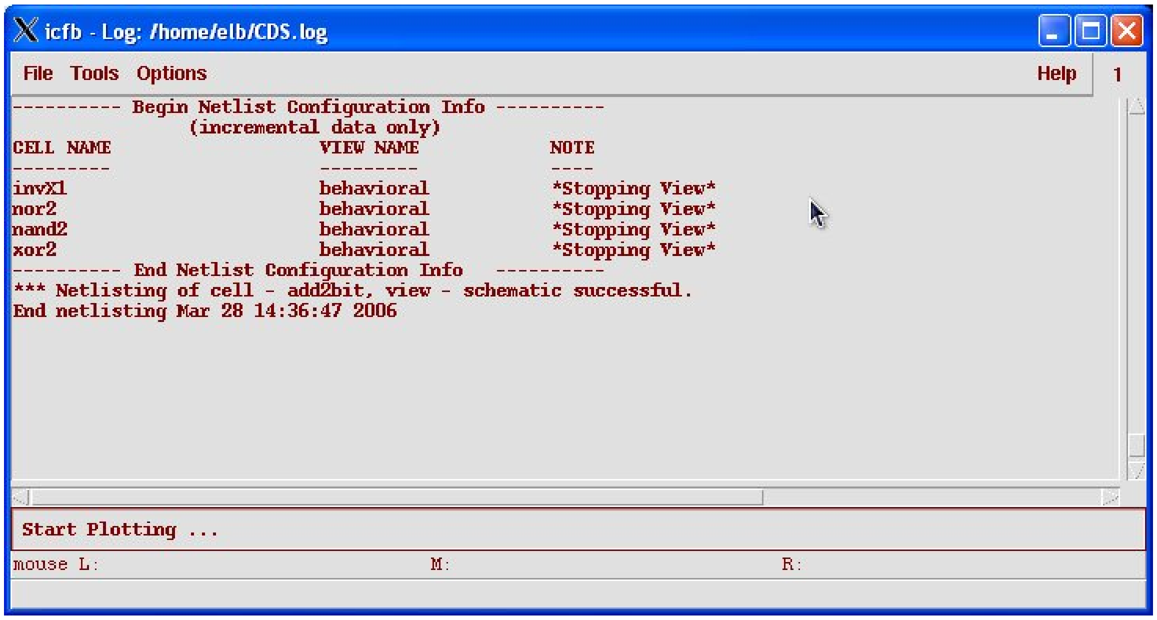
(c) results.txt

Figure 4.12: Data files used in Figure 4.11



WARNING: This design is about to be renetlisted. Do you want to continue? If not, the existing netlist will be used for the simulation.

Figure 4.13: Dialog box for re-netlisting a previously netlisted design



```
---------- Begin Netlist Configuration Info ----------
        (incremental data only)
CELL NAME                   VIEW NAME           NOTE
---------                   ---------           ----
invX1                       behavioral          *Stopping View*
nor2                        behavioral          *Stopping View*
nand2                       behavioral          *Stopping View*
xor2                        behavioral          *Stopping View*
---------- End Netlist Configuration Info  ----------
*** Netlisting of cell - add2bit, view - schematic successful.
End netlisting Mar 28 14:36:47 2006
```

```
Start Plotting ...
mouse L:                              M:                           R:
```

Figure 4.14: The Netlisting Log for the Two Bit Adder

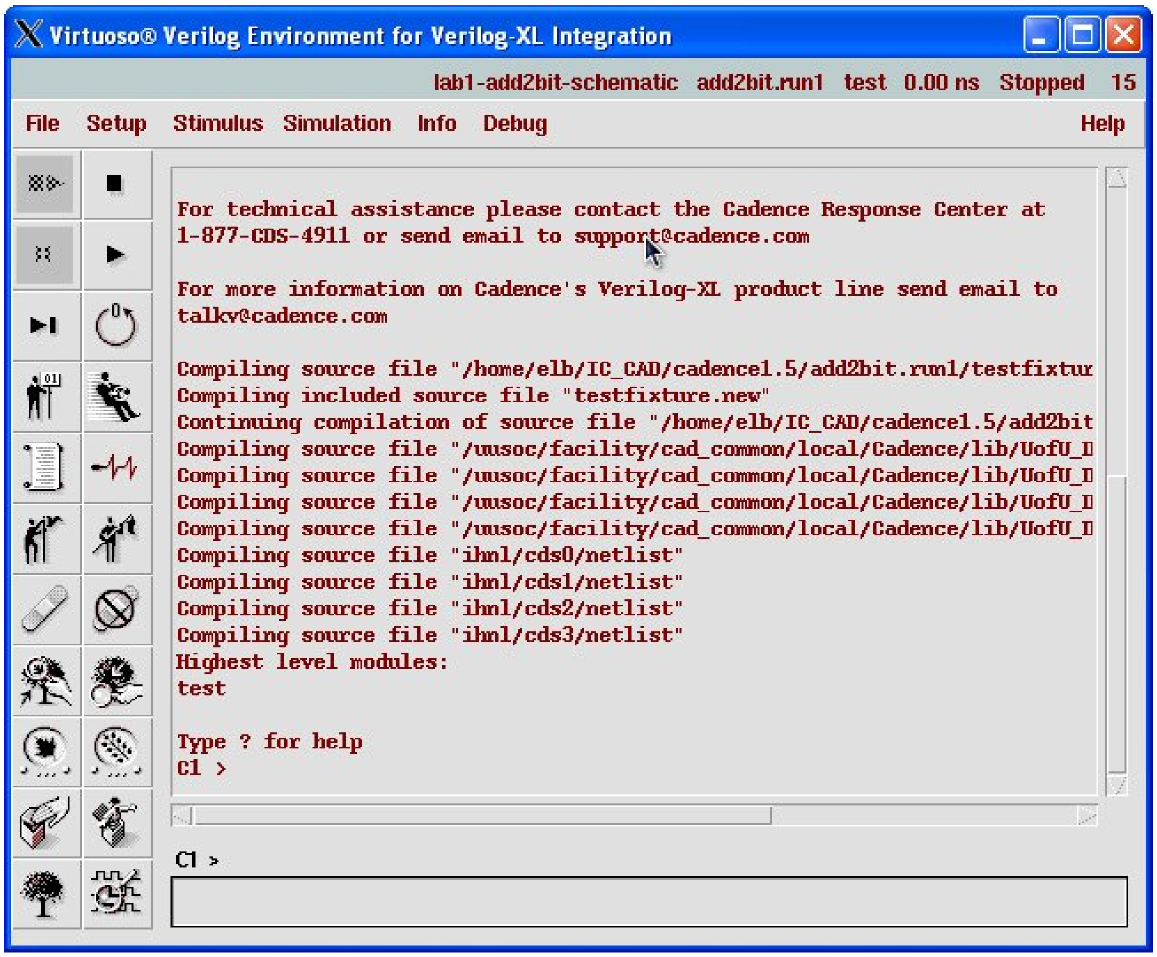
ing the testbench you designed by selecting **Simulation** → **Continue** or by pressing the **play** widget (the right-facing triangle). This runs your testbench on the Composer-generated netlist. The result of the simulation on the testbench from Figure 4.9 is shown in Figure 4.16. You can see that the **$display** statements have printed the simulation values, and none of the **ERROR** statements have printed, which means that the circuit passed this simulation with correct results.

Printing Verilog-XL Output

The output in the Verilog-XL window is available through the **Edit** → **View Log File** → **Simulation** menu choice. This will bring up the contents of the Verilog-XL in a window where you can **Save-As** any file you like. This is just a text file that has the results of the **$display** statements in your testbench.

SimVision Waveform Viewer

Now that you have a successful simulation, you can, if you wish, look at the waveforms. Waveforms are a good way to get certain types of information from the simulation, but hopefully you've checked for enough values


Virtuoso® Verilog Environment for Verilog-XL Integration
lab1-add2bit-schematic add2bit.run1 test 0.00 ns Stopped 15
File Setup Stimulus Simulation Info Debug Help
C1 >

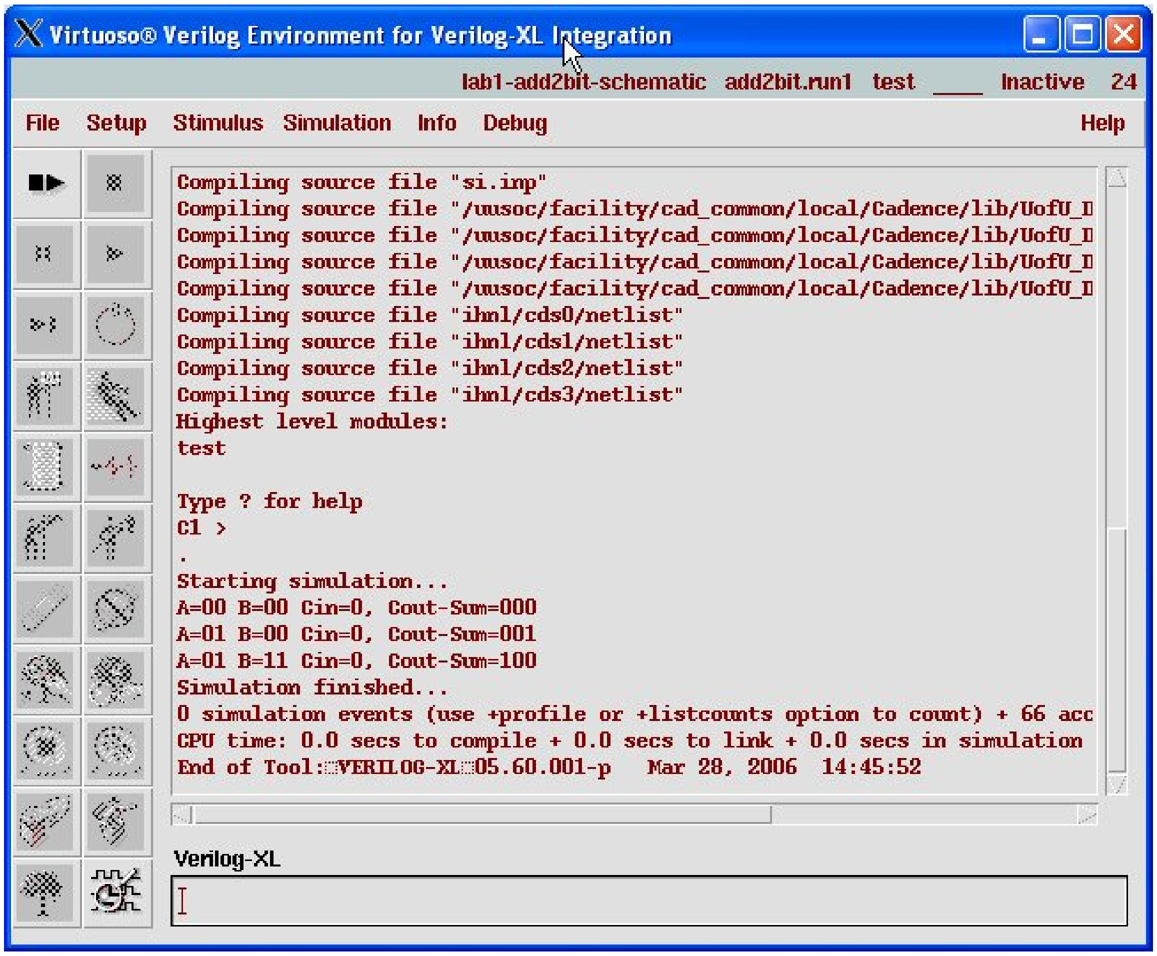
```
For technical assistance please contact the Cadence Response Center at
1-877-CDS-4911 or send email to support@cadence.com

For more information on Cadence's Verilog-XL product line send email to
talkv@cadence.com

Compiling source file "/home/elb/IC_CAD/cadence1.5/add2bit.run1/testfixtur
Compiling included source file "testfixture.new"
Continuing compilation of source file "/home/elb/IC_CAD/cadence1.5/add2bit
Compiling source file "/uusoc/facility/cad_common/local/Cadence/lib/UofU_D
Compiling source file "/uusoc/facility/cad_common/local/Cadence/lib/UofU_D
Compiling source file "/uusoc/facility/cad_common/local/Cadence/lib/UofU_D
Compiling source file "/uusoc/facility/cad_common/local/Cadence/lib/UofU_D
Compiling source file "ihnl/cds0/netlist"
Compiling source file "ihnl/cds1/netlist"
Compiling source file "ihnl/cds2/netlist"
Compiling source file "ihnl/cds3/netlist"
Highest level modules:
test

Type ? for help
C1 >
```

Figure 4.15: The Verilog-XL Window After Netlisting


Virtuoso® Verilog Environment for Verilog-XL Integration
lab1-add2bit-schematic add2bit.run1 test Inactive 24
File Setup Stimulus Simulation Info Debug Help
Verilog-XL

```
Compiling source file "si.inp"
Compiling source file "/uusoc/facility/cad_common/local/Cadence/lib/UofU_D
Compiling source file "/uusoc/facility/cad_common/local/Cadence/lib/UofU_D
Compiling source file "/uusoc/facility/cad_common/local/Cadence/lib/UofU_D
Compiling source file "/uusoc/facility/cad_common/local/Cadence/lib/UofU_D
Compiling source file "ihnl/cds0/netlist"
Compiling source file "ihnl/cds1/netlist"
Compiling source file "ihnl/cds2/netlist"
Compiling source file "ihnl/cds3/netlist"
Highest level modules:
test

Type ? for help
C1 >
.
Starting simulation...
A=00 B=00 Cin=0, Cout-Sum=000
A=01 B=00 Cin=0, Cout-Sum=001
A=01 B=11 Cin=0, Cout-Sum=100
Simulation finished...
0 simulation events (use +profile or +listcounts option to count) + 66 acc
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.0 secs in simulation
End of Tool: VERILOG-XL 05.60.001-p   Mar 28, 2006  14:45:52
```

Figure 4.16: Result of Running with the Testbench from Figure 4.9

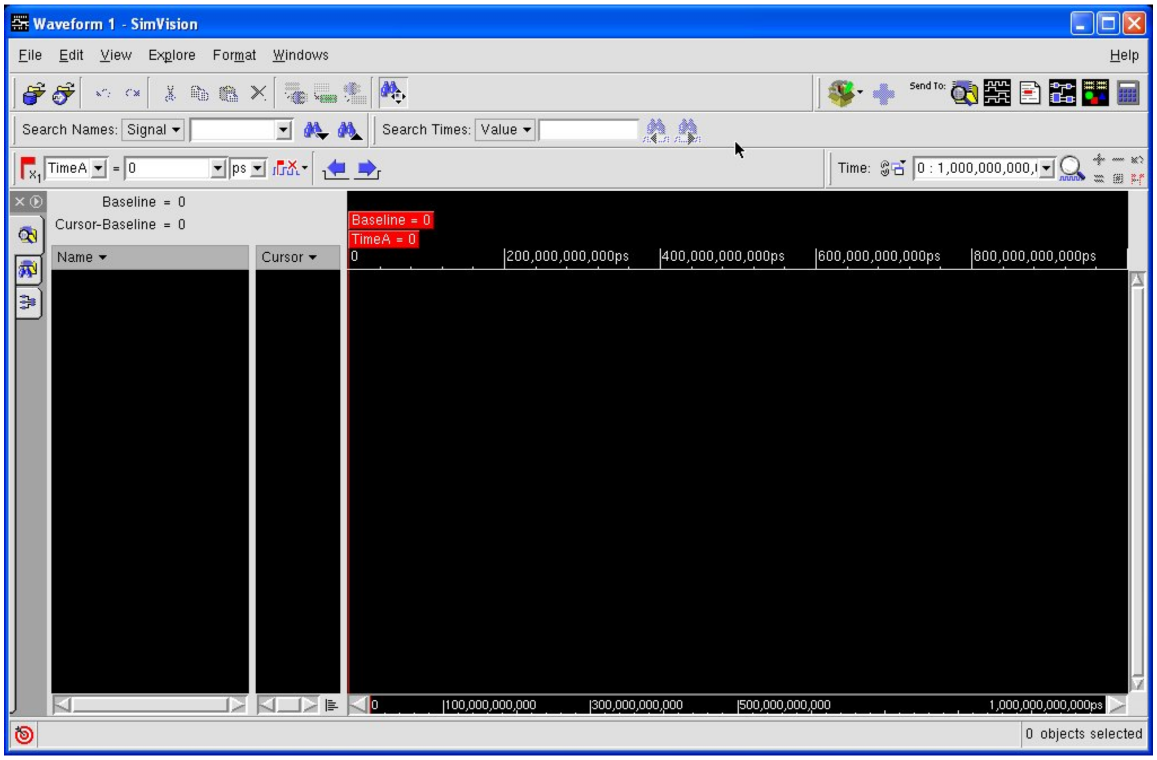


Figure 4.17: Waveform Window Without any Signals

in your testbench that you already know if your circuit is working or not. The waveform viewer is a good place for debugging if your circuit isn't completely correct though. Especially if you've selected the **Record All Signals** option, you can use the waveform viewer and navigation system to look at signals deep inside your circuit to see where things have started to fail. Standard debugging techniques apply here: starting with the incorrect output and working backwards in the circuit to figure out what caused that output to be incorrect is a great way to start. The waveform viewer attached to Verilog-XL by default is SimVision.

To start the SimVision waveform viewer after running a simulation, select **Debug** → **Utilities** → **View Waveform...** or pressing the waveform viewer widget (bottom right in the widget array). The **Waveform 1 - SimVision** window appears as shown in Figure 4.17.

No waveforms appear yet. You need to select which signals you want to appear in the waveform window. You do this through the **Design Browser**. Select **Windows** → **New** → **Design Browser** or press the **Design Browser** button (it looks like a folder with a magnifying glass in front of it), and the **Design Browser 1 SimVision** window appears as shown in Figure 4.18.

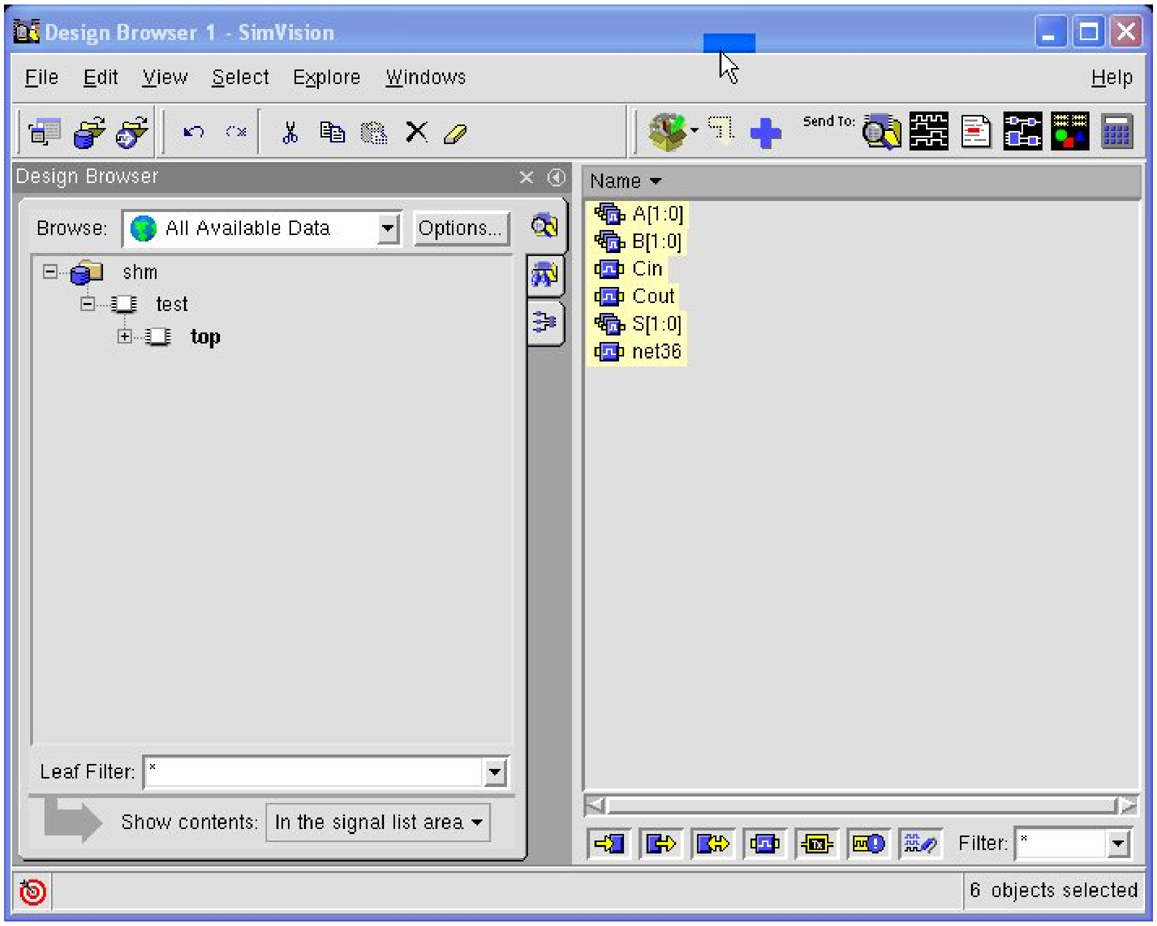
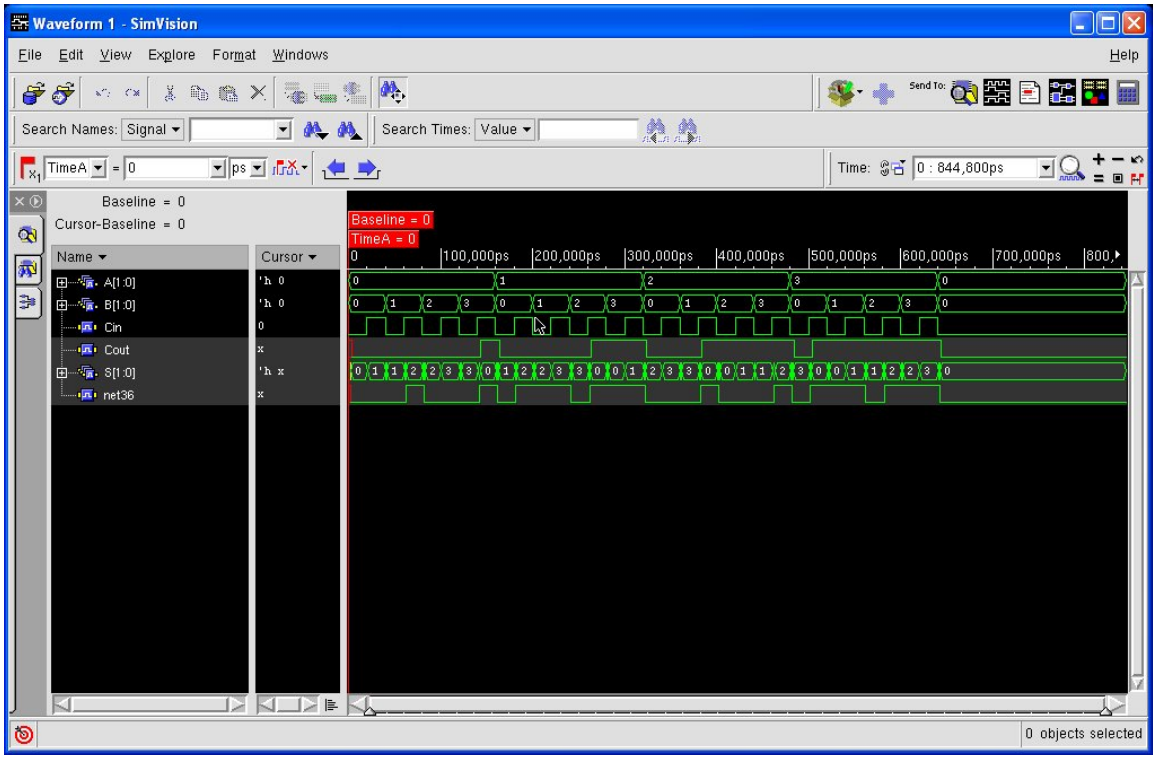

Design Browser 1 - SimVision
File Edit View Select Explore Windows Help
Send To:
Design Browser
Browse: All Available Data
Options...
shm
test
top
Name
A[1:0]
B[1:0]
Cin
Cout
S[1:0]
net36
Leaf Filter: *
Show contents: In the signal list area
Filter: *
6 objects selected

Figure 4.18: Design Browser with Signals Selected

Using this window you can navigate the hierarchy of your circuit to select the specific wires that you'd like to see in the waveform window. Once selected, you send the values to the waveform window using the widget that looks like waveforms in a black square. You can also right-click after selecting the signals you want to get a menu with the choice to **send to Waveform Window**. The **Design Browser** with all the top level I/O signals in the two bit adder selected is shown in Figure 4.18.

Once you send waveforms to the **Waveform** window, you'll see them as in Figure 4.19. This set of waveforms was generated using the exhaustive testbench from Figure 4.10. I've also zoomed out to see the entire test using the controls on the right neat the magnifying glass icon. The widget with the equal sign (=) will zoom out to fit the entire simulation on the X-axis. The + and - buttons will zoom in and out. You can also set cursors and measure time between cursors using these controls.



Search Names: Signal
Search Times: Value
TimeA = 0 ps
Time: 0 : 844,800ps
Baseline = 0
Cursor-Baseline = 0
Baseline = 0
TimeA = 0

Name	Cursor
A[1:0]	'h 0
B[1:0]	'h 0
Cin	0
Cout	x
S[1:0]	'h x
net36	x

Figure 4.19: Waveform Window Showing Exhaustive Test Results

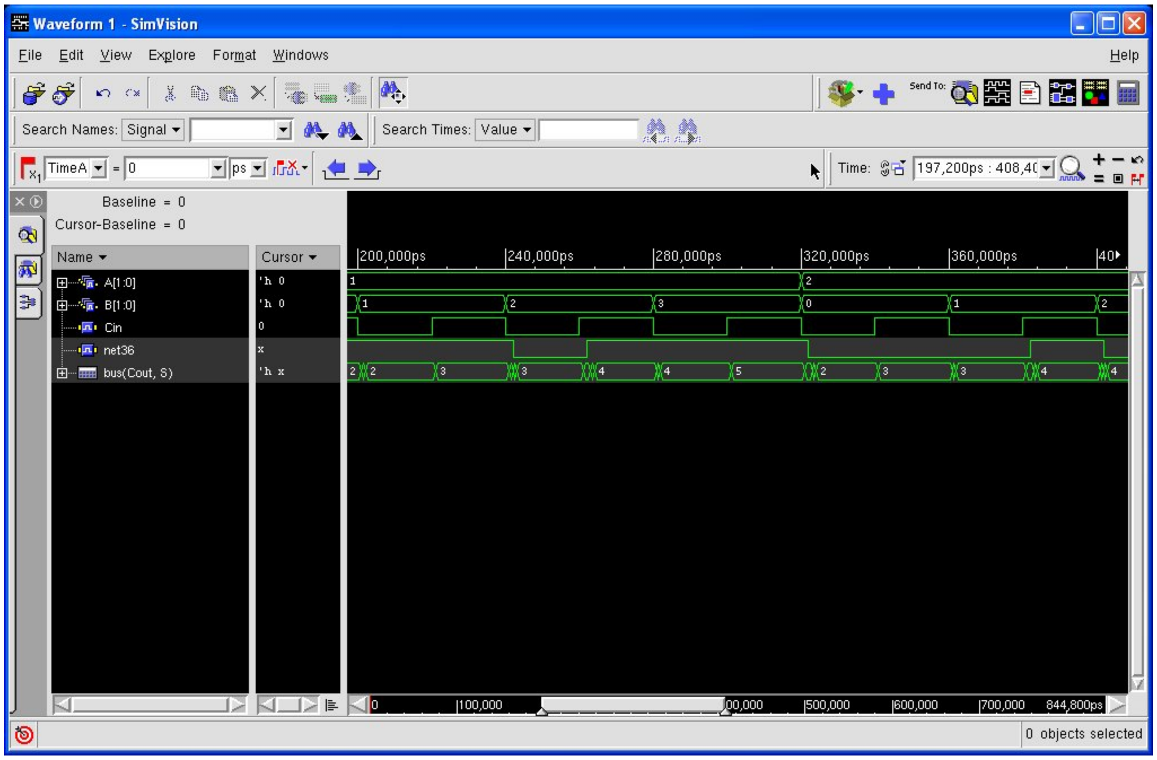


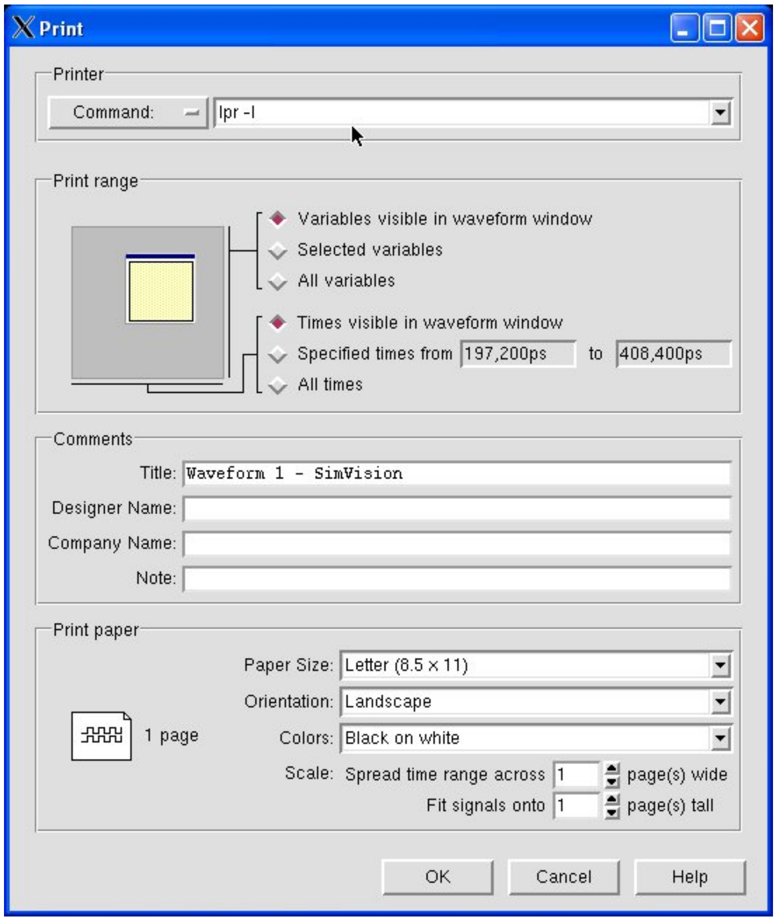
Figure 4.20: Waveform Window Showing Outputs as a Bus

However, the outputs are a little hard to read in this window because the carry-out (**Cout**) and sum (**Sum**) outputs are on different waveforms. It would be easier if these were combined into a single bus so that you could read the output value as a three-bit number. You can do this by selecting the **Cout** and **Sum** traces in the **Waveform** window and collecting them into a **bus** using **Edit** → **Create** → **Bus** or using the **Create Bus** option in the right-click menu once the traces are selected. The result, zoomed in to a closer view, is shown in Figure 4.20. In this waveform the output is collected into a bus and reads as a three-bit output value.

Printing Waveform Outputs

Output from the SimVision waveform viewer can be printed using the **File** → **Print Window...** menu choice. This will bring up yet another different print dialog box, as shown in Figure 4.21. You can fill in your name and other information to have it printed along with the waveforms. At the top of the dialog box you can select a Unix/Linux print command (the default **lpr -l** works fine), or you can select **Print to file:** and give a file name. The

With no -P argument, lpr uses the printer defined in your $PRINTER environment variable



Printer — Command: lpr -l

Print range
- [✓ Variables visible in waveform window]
- Selected variables
- All variables
- [✓ Times visible in waveform window]
- Specified times from 197,200ps to 408,400ps
- All times

Comments
- Title: Waveform 1 - SimVision
- Designer Name:
- Company Name:
- Note:

Print paper
- 1 page
- Paper Size: Letter (8.5 x 11)
- Orientation: Landscape
- Colors: Black on white
- Scale: Spread time range across 1 page(s) wide
- Fit signals onto 1 page(s) tall

Figure 4.21: Printing Dialog Box for SimVision

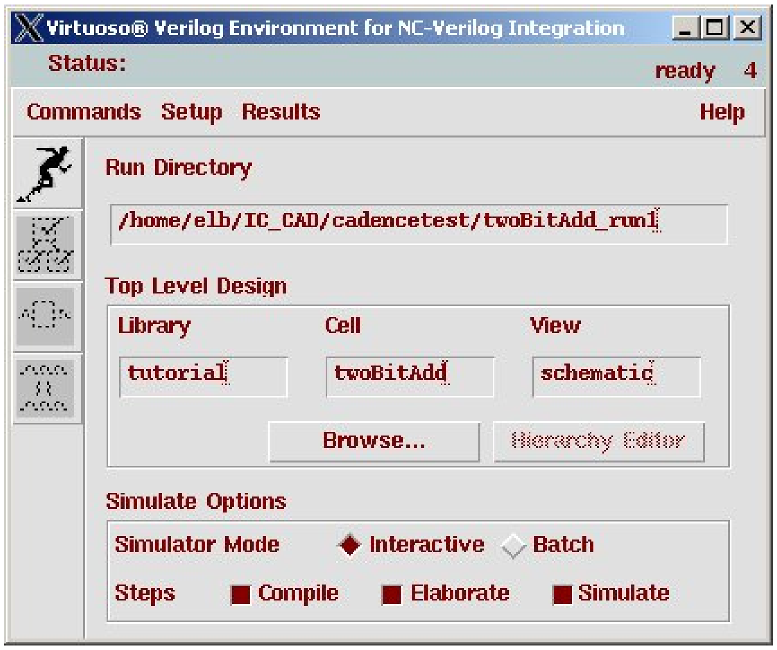
result is a postscript file. I haven't found hand-modification required for the postscript produced by SimVision.

4.1.2 NC_Verilog: Simulating a Schematic

As an example of simulating a schematic we'll again use the two-bit adder from Chapter 3. To start NC_Verilog simulation you can use the CIW window by going to **Tools** → **Verilog Integration** → **NC_Verilog...**. The **Verilog Environment for NC_Verilog Integration** window appears in which the Run Directory, Library, Cell and View fields need to be filled in.

Or (the much easier way) open up the Composer schematic of the two-bit adder using the library browser and in the Composer schematic editing window, select **Tools** → **Simulation** → **NC_Verilog**. The **Verilog Environ-**

It is very important to have a separate run directory for each different design, but you can keep the same run directory for different simulations of the same design.


Virtuoso® Verilog Environment for NC-Verilog Integration
Commands Setup Results Help
Browse... Hierarchy Editor

Status: ready 4

Run Directory: /home/elb/IC_CAD/cadencetest/twoBitAdd_run1

Top Level Design

Library	Cell	View
tutorial	twoBitAdd	schematic

Simulate Options

Simulator Mode: [✓ Interactive] Batch

Steps: [✓ Compile] [✓ Elaborate] [✓ Simulate]

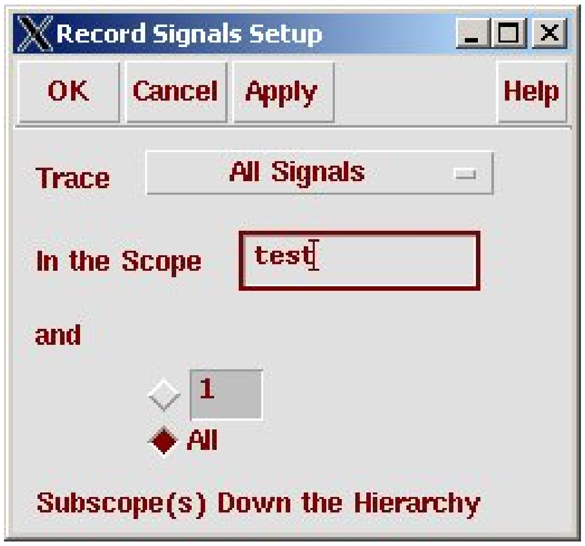
Figure 4.22: Dialog Box for Initializing a Simulation Run Directory for NC_Verilog

ment for NC_Verilog Integration window appears with all the fields filled in. If the fields are not filled in, you can use the **Browse** button to open a small **library manager** window to select the library, cell, and schematic view.

The **Run Directory** can be changed or left as default **<designname>_run1**. Note that the default run directory name for NC_Verilog has an underscore rather than the dot used in the Verilog-XL environment. It's not a terribly important difference, but it can help you keep track of which run directory is which if you're using both simulators. A dialog box for simulation of the two-bit adder from Chapter 3 is shown in Figure 4.22.

Once the **Verilog Environment for NC_Verilog Integration** dialog has been filled in, initialize the design by selecting **Commands** → **Initialize Design** or by selecting the **Initialize Design** widget that looks like a sprinter starting a race.

This will initialize the NC_Verilog simulator and make some other widgets in the **Verilog Environment for NC_Verilog Integration** dialog active, including the **Generate Netlist** widget. Selecting that widget (or issuing that command using the **Commands** menu) will generate a simulation netlist for the schematic.



Trace: All Signals

In the Scope: test

and

1

[✓ All]

Subscope(s) Down the Hierarchy

Figure 4.23: The Record Signals Dialog Box

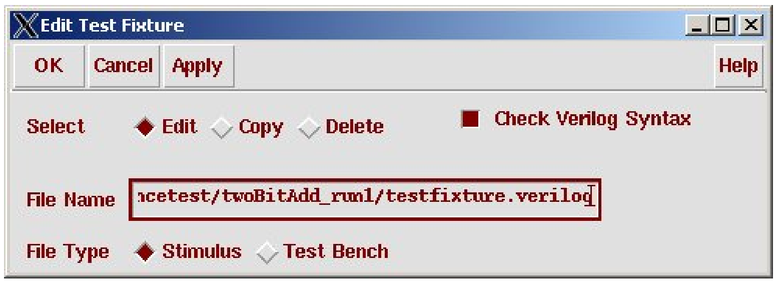
Before starting simulation the environment needs to be set up and an input stimulus file for the circuit (a testbench, also called a *test fixture* by NC_Verilog) needs to be created.

Setting up the Simulation Environment

Select **Setup** → **Record Signals...**. In the **Record Signal Setup** window which appears you can change how many levels of hierarchy will have their internal signals saved. If you leave the default setting of 1 you will only get the top-level I/O signals from your schematic. The **test** scope in the netlist is the wrapper that the NC_Verilog integration system builds with one instance of your circuit (with label **top**) and the test fixture. If you would like to be able to see the values of signals further down in your circuit hierarchy, change the number or select **All** to have all hierarchical signals saved. See Figure 4.23 for details.

Generating the Simulation Netlist

Next generate the simulation netlist by selecting the **Generate Netlist** widget (the widget that looks like a tree of boxes with check marks in them), or by using the **Commands** → **Generate Netlist** menu choice. The result is that a netlist is generated with a top-level simulation macro named **test** that contains one instance of your circuit (the DUT) labeled **top** and a template for the test fixture code that you will fill in to drive your simulation.



Select [✓ Edit] Copy Delete

[✓ Check Verilog Syntax]

File Name: ıcetest/twoBitAdd_run1/testfixture.verilog

File Type [✓ Stimulus] Test Bench

Figure 4.24: Dialog to Create a New Test Fixture Template

If you're re-simulating a design in the same run directory, you'll see a re-netlisting dialog box like that in Figure 4.13. As in the Verilog-XL case, it's almost always the case that you should re-netlist to make sure you're simulating the most recent changes to your circuit. The result of the netlisting command should be very similar to that shown in Figure 4.14 because the same netlister is used for both Verilog-XL and NC_Verilog. This step also enables more of the widgets and commands in the **Verilog Environment for NC_Verilog Integration** window.

Defining the Test Fixture (testbench)

Select **Commands** → **Edit Test Fixture**. This brings up the **Edit Test Fixture** window with the default test fixture file name of **testfixture.Verilog** already selected as shown in Figure 4.24. The **File Type** should be **Stimulus** and it's a good idea to select the **Check Verilog Syntax** box. The **Test Bench File Type** is the top-level module named **test** that is created with the instance of your circuit instantiated as the DUT named **top**. You can view this file if you're curious. Your stimulus code is inserted with a Verilog **'include** statement in the **Test Bench**

You can edit the stimulus portion of the test bench by selecting **edit** in the **Edit Test Fixture** dialog and either selecting **Apply** or **OK**. The default editor (most likely emacs) will open up. Use this editor to type in the Verilog code that you want to use as your test fixture. Then save the test fixture and close the editor. The original test fixture template should look something like that in Figure 4.8 from Section 4.1.1. The interface signals are found on the symbol and repeated here in the test fixture template. Your test code should go after the interface signal definitions and before the **end** statement. This piece of Verilog code is an **initial** block which is executed when you simulate the netlist assembled by Composer.

For examples of test fixtures for the two bit adder, see Figures 4.9, 4.10, and 4.11 in Section 4.1.1. Once you have entered your test fixture code and saved the result without Verilog errors you are ready to simulate.

Note that even though NC_Verilog includes many Verilog-2000 features, the syntax checker for the test fixture appears to be checking for the Verilog-1995 syntax that Verilog-XL prefers. I belive, but have not tested completely, that if you want to use Verilog-2000 features in your test fixture you can remove the **Check Verilog Syntax** option when editing. However, this will open the door for Verilog syntax errors of all types to creep into your test fixture code so beware!

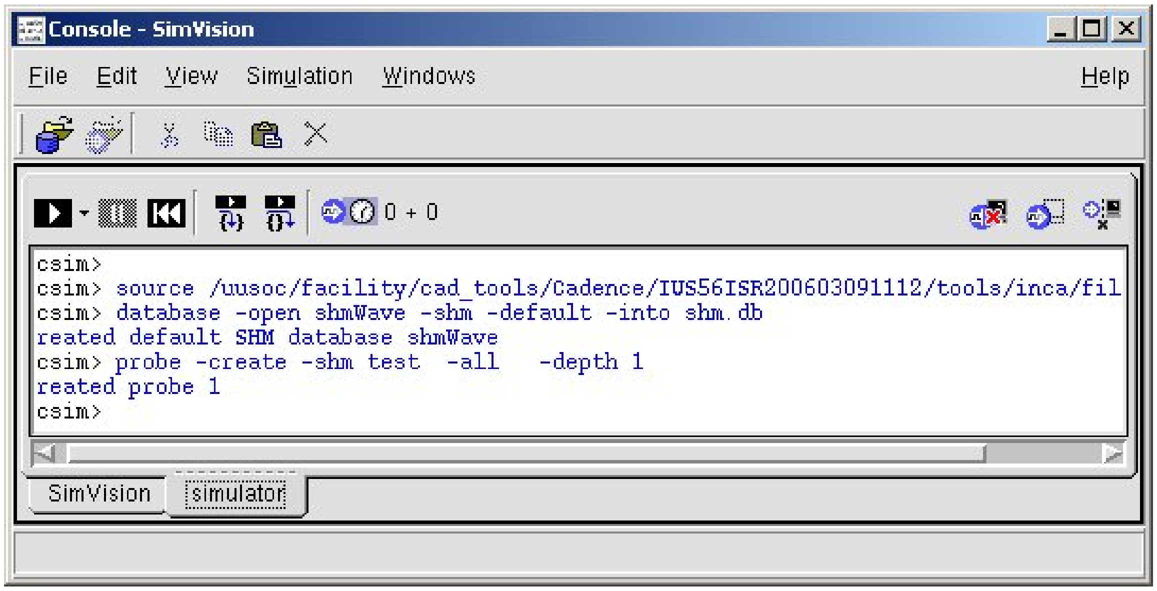
Running the Simulation

Once you have a testbench that passes the syntax check, and you have set up the signals that you want to record, you can run the simulation. Start the Verilog simulation by selecting **Commands** → **Simulate** menu choice or pressing the **Simulate** widget button with the DUT rectangle with square waves on right and left denoting inputs and outputs.

This will analyze, elaborate, and compile the NC_Verilog simulator for your design. The result is that two new windows pop up as shown in Figures 4.25 and 4.26: the **Console - SimVision** and the **Design Browser 1 - SimVision**. The **Design Browser** is the same as seen in the Verilog-XL example and is shown after the **test** and **top** nodes have been expanded. The **Console** is a control console for the NC_Verilog simulation. If you look carefully you'll see the command line that initializes the simulation which includes the information we set previously to include (**probe**) signals at all levels of the circuit hierarchy.

Once these windows are open you can select the signals that you would like to see using the **Design Browser**. Click the **Send to Waveform** widget in the **Design Browser** window to open a **Waveform** window. Then click on the **Run Simulation** button in any of the windows. This button is looks like a white right-pointing triangle, or a standard "play" button from applications like audio players. The simulation runs and the waveforms appear in the **Waveform** window. You can zoom in and out, add or delete waveforms, and group or expand buses as described in Section 4.1.1. Printing a waveform from SimVision is also described in that Section. The result of simulating using the exhaustive test fixture from Figure 4.11 is shown in Figure 4.27.

The SimVision application is a very complex front end to the NC_Verilog simulator. It's not just a waveform viewer as it is with Verilog-XL. From SimVision you can set break points, single step simulations, explore the circuit being simulated using a **Signal Flow Browser**, reference the simulation to the Verilog source code, and many other features. Please explore these features as you are simulating your Verilog code with NC_Verilog!



```
csim>
csim> source /uusoc/facility/cad_tools/Cadence/IUS56ISR200603091112/tools/inca/fil
csim> database -open shmWave -shm -default -into shm.db
reated default SHM database shmWave
csim> probe -create -shm test  -all   -depth 1
reated probe 1
csim>
```

Figure 4.25: SimVision Console Window

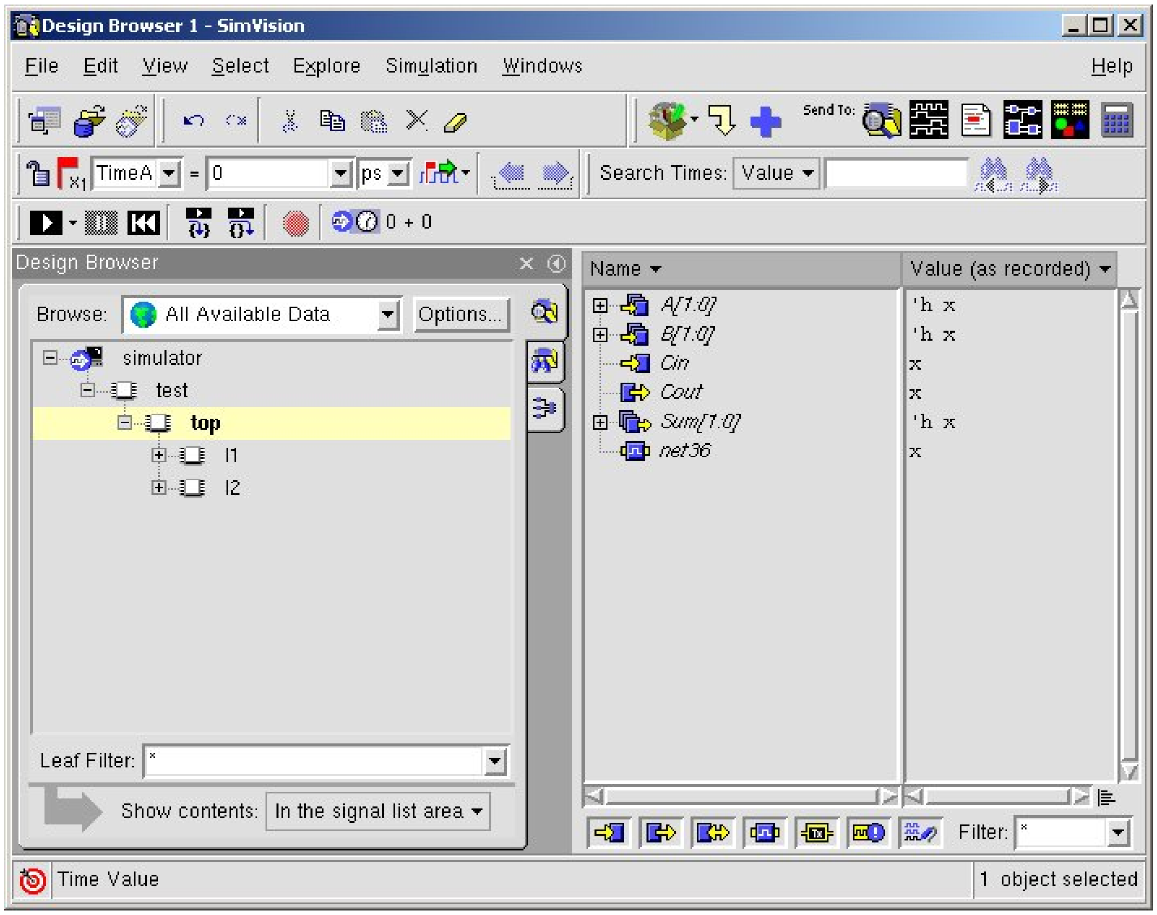
Printing NC_Verilog Output

Printing waveforms from SimVision is the same whether you get to SimVision from Verilog-XL or from NC_Verilog. The dialog for printing waveforms is shown in Figure 4.21 in Section 4.1.1.

The output of the **$display** statements in your testbench code are stored in the log file of the simulation. There may be a way to access the log file from the SimVision menus, but I haven't found it. To see the output of the **$display, $monitor** and other output statements in your testbench look at the logfile in the simulation directory. Recall that the default simulation directory name is <**designname**>**_run1**, and the default log file name is **simout.tmp**. This is simply a text file that includes the log information from the NC_Verilog simulation including all the outputs from **$display** and from other output commands in your test fixture.

4.2 Behavioral Verilog Code in Composer

One way to describe the functionality of your circuit is to use collections of Boolean gates from a standard cell library, as is shown in the previous sections. Another way is to describe the functionality of the circuit as Verilog code. The Verilog code might be purely structural meaning that it consists only of instantiations of gates from that same standard cell library. In that


Design Browser 1 - SimVision
File Edit View Select Explore Simulation Windows Help
Send To:
TimeA = 0 ps
Search Times: Value
0 + 0
Design Browser
Browse: All Available Data Options...
simulator
test
top
I1
I2
Leaf Filter: *
Show contents: In the signal list area
Name
Value (as recorded)
Filter: *
Time Value
1 object selected

Name	Value (as recorded)
A[1:0]	'h x
B[1:0]	'h x
Cin	x
Cout	x
Sum[1:0]	'h x
net36	x

Figure 4.26: SimVision Design Browser Window

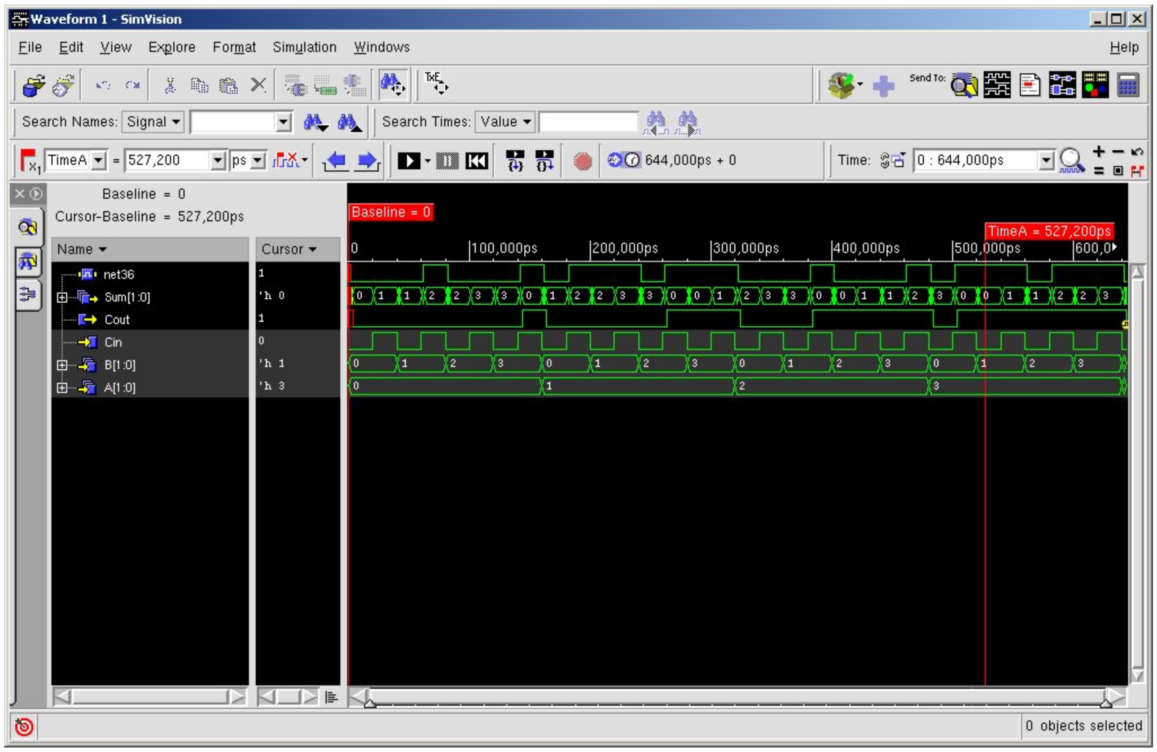


Figure 4.27: Waveform Window Showing the Output of Exhaustive Simulation

case the Verilog is simply a textual version of the schematic. Verilog can also describe a circuit as a behavior meaning that it uses statements in the Verilog language to describe the desired behavior. A behavioral Verilog description will look much more like a software program than like a circuit description, and will need to be converted (or *synthesized*) into a structural description at some point if you want to actually build the circuit, but as an initial description, a behavioral model can be much more compact, can use higher-level descriptions, and can simulate much more quickly.

Synthesis of behavioral Verilog into structural (gate-level) Verilog is covered in Chapter 8.

A behavioral Verilog model can be developed as a Verilog program using a text editor, and simulated using a Verilog simulator without ever using the Composer schematic tool as will be seen in Section 4.3. However, symbols in Composer schematics can be an interface to behavioral Verilog code just as easily as they can contain gate schematics. These symbols which encapsulate Verilog behavioral code may be connected in schematics using wires just like other symbols, and can peacefully coexist with other symbols that encapsulate gate views. The cell view used for general schematics is **schematic** and the cell view we use for behavioral Verilog views is **behavioral**. In fact, a single cell may have both of these views and you can tell the netlister later which view to use in a given simulation. You can actually use any view name you like for the Verilog behavioral view. It might actually make sense in some situations to use a different name, like **verilog** for your Verilog view so you can keep track of which pieces of Verilog are your own behavioral code and which pieces of Verilog are the library cell descriptions. For now we won't make a distinction.

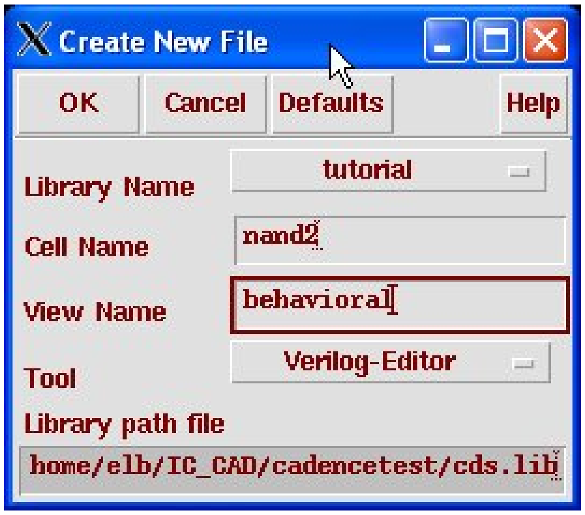
*Note that cell views that consist of transistors or that correspond to a single standard cell may use **cmos_sch** as their schematic view*

This allows great freedom in terms of how you want to describe your circuits. You can start with a Verilog **behavioral** view for initial functional simulation, and then incrementally refine pieces of the circuit into gate level **schematic** views, all while using the same testbench code for simulation.

4.2.1 Generating a Behavioral View

Making a behavioral view is very similar to making any other cell view. You use the Library Manager to make a new cell view, except that you choose **behavioral** as the view name instead of **schematic**.

Start in the Library Manager and select the library in which you want to make your new cell view. Of course, if you're starting a whole new library, you need to make the new library first. If you're adding a behavioral view to a cell that already exists, select the cell too. That will fill in both the library and the cell in the **new cell view** dialog box as shown in Figure 4.28. In this case I'm using the nand2 that was designed as a transistor schematic in Chapter 3 Section 3.3 and will add a behavioral view of that cell. Clicking OK to the dialog box of Figure 4.28 will open up a **Verilog Editor**. Actually,



Library Name: tutorial
Cell Name: nand2
View Name: behavioral
Tool: Verilog-Editor
Library path file: home/elb/IC_CAD/cadencetest/cds.lib

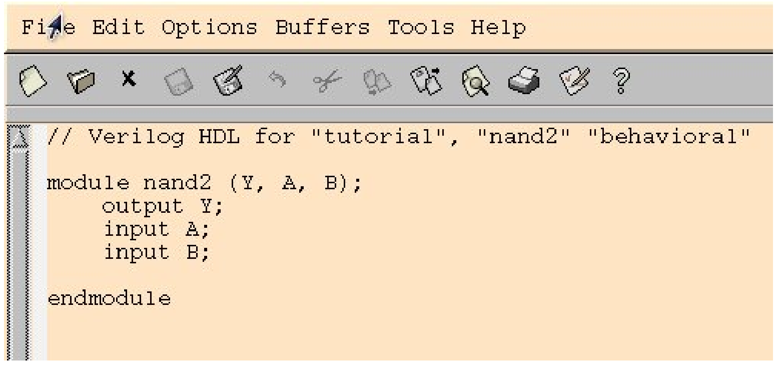
Figure 4.28: Dialog Box for Creating a Behavioral View

this is just a text editor, in this case emacs.

In the editor you will get a very simple template for your new Verilog component. If you have already made a symbol for your cell, then the template will already have the inputs and outputs of the Verilog module defined from the pins on the symbol. If you haven't yet made a symbol, there will just be a module name. In this case because we already defined a symbol, the Verilog template looks like Figure 4.29. You now need to fill in this template with Verilog code that describes the behavior of your cell.

Verilog built-in gate primitives include and, or, not, nand, nor, xor, and xnor.

One (particularly simple) example of a Verilog behavioral description is shown in Figure 4.30. This description uses the Verilog built-in nand gate

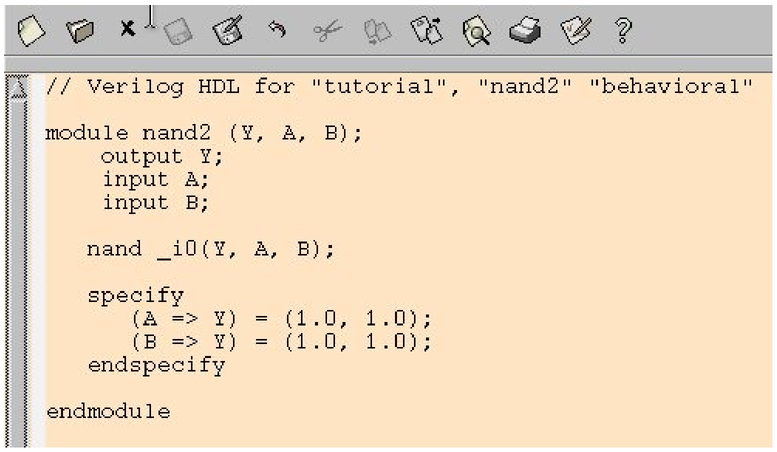


```
// Verilog HDL for "tutorial", "nand2" "behavioral"

module nand2 (Y, A, B);
    output Y;
    input A;
    input B;

endmodule
```

Figure 4.29: Behavioral View Template Based on the nand2 Symbol



```
// Verilog HDL for "tutorial", "nand2" "behavioral"

module nand2 (Y, A, B);
    output Y;
    input A;
    input B;

   nand _i0(Y, A, B);

   specify
      (A => Y) = (1.0, 1.0);
      (B => Y) = (1.0, 1.0);
   endspecify

endmodule
```

Figure 4.30: Complete Behavioral Description of a nand2 Cell

model. This isn't a gate from an external library, it's the built-in Verilog primitive. The name of the instance is **_i0**. Note that I've given the instance a name that starts in an underscore so that if I ever navigate to this instance in a simulation I can tell that it came from the behavioral model of the cell.

I've also used **specify** statements to describe the input to output delay of the cell. In this case both the rising and falling delays for both the A to Y and B to Y paths are set to 1.0 time units. These specify blocks are very important later on! They are the mechanism through which extracted timings from the synthesis procedures are annotated to your schematic. The timings in the Standard Delay Format (.sdf) file that comes from the synthesis program will be in terms of the input to output path delay, and the back-annotation procedure will look for specify statements in the cell descriptions to update to the new extracted timing values.

There are many different descriptions you could use in Verilog for a simple NAND gate like this. You could, for example, define the function as a continuous assignment

```
assign Y = ~(A & B);
```

or you could define the function as an assignment inside an **always** block

```
reg Y;
always @(A or B)
  begin
    Y = ~(A & B);
  end
```

or, if you had a standard cell library available with the appropriate gates, you could define it the NAND as a structural composition of gates from that library.

```
wire w;
AND2 _u1(w, A, B);
INV  _u2(Y,w);
```

Each of these descriptions would work. The point is that any legal Verilog behavioral/structural syntax that will simulate in the Verilog simulator will work for the contents of a **behavioral** cell view.

The syntax checker appears to be checking for Verilog 1995 (i.e. Verilog-XL) syntax in this check.

Once you have entered your Verilog code into the template in the editor, save the file and quit the editor. This will cause the Composer Verilog integration system to check the syntax of your code and make sure that it is compatible with the Verilog editor. If it reports errors you need to re-edit and fix your Verilog code.

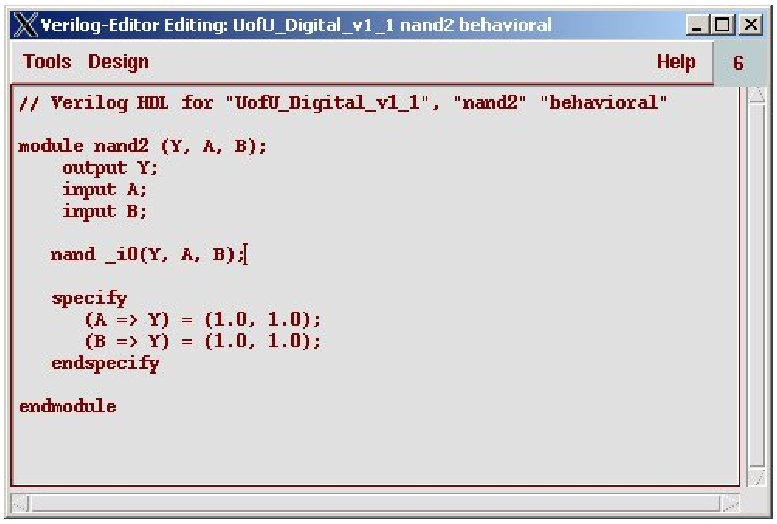
It will also check to make sure that the pin descriptions in your schematic view, the pins in your symbol, and the interface in your behavioral view are all consistent. This is critical if all the different views can be used in various combinations in later simulations.

Once you have a behavioral view of a cell, that view can be simulated from Composer using a Cadence Verilog simulator, and it can be included via the symbol in other schematics and those schematics can be simulated using either Verilog-XL or NC_Verilog. All of the cells in the **UofU_Digital_v1_1** library have behavioral views. These views are the default choice when simulating schematics that include instances of these cells. If a transistor switch-level simulation is desired, follow the procedure in Section 4.4.1.

4.2.2 Simulating a Behavioral View

The simplest way to simulate a behavioral view is to make a new schematic and include an instance of the symbol that encapsulates the behavioral view. Then you can use the procedure described in Section 4.1.1 to simulate with Verilog-XL, or the procedure described in Section 4.1.2 to simulate with NC_Verilog.

You can also simulate the Verilog code in the behavioral view directly without going through a schematic. If you double-click the **behavioral** view in the **library manager** you'll open an emacs editing window to edit the code. However, if you right-click on the behavioral view you'll get a menu where you can choose **Open (Read Only)**. If you select this choice you'll open a read-only window that looks like that in Figure 4.31. From here you



```
// Verilog HDL for "UofU_Digital_v1_1", "nand2" "behavioral"

module nand2 (Y, A, B);
    output Y;
    input A;
    input B;

   nand _i0(Y, A, B);

   specify
      (A => Y) = (1.0, 1.0);
      (B => Y) = (1.0, 1.0);
   endspecify

endmodule
```

Figure 4.31: Read Only Window for Simulation of Behavioral View

can select **Tools** → **Verilog-XL** to simulate that code using the Verilog-XL simulator as described in Section 4.1.1. There appears to be no option for direct simulation using NC_Verilog.

4.3 Stand-Alone Verilog Simulation

The preceeding sections have all used the Composer integration framework to simulate circuits and behavioral Verilog code through the Composer schematic capture tool. Of course, you may have Verilog code that you've developed outside of the schematic framework that you'd like to simulate. It's entirely reasonable and common to develop a system as a Verilog program using a text editor and then using that Verilog code as the starting point for a CAD flow. The CAD flow in this case goes from that Verilog code (through synthesis) directly to the back end place and route without ever using a schematic. Of course, the Verilog simulators can run (simulate) that Verilog code directly.

In order to simulate your Verilog code you will need both the code that describes your system, and testbench code to set the inputs of your system and check that the outputs are correct. As shown in Figure 4.1 this corresponds to DUT code (the code describing your system) and testbench or testfixture code. These should both be part of a single top-level module that is simulated. An example is shown in Figure 4.2 where the testfixture code is contained in a separate file named **testfixture.verilog**.

As another example, consider the Verilog code describing a simple state machine shown in Figure 4.32. This state machine looks at bits on the input **insig** and raises the output signal **saw4** whenever the last four bits have all been **1**. Of course, there are simpler ways of building a circuit with this functionality, but it makes a perfectly good example of a small finite state machine to simulate with each of the Verilog simulators.

In order to simulate this piece of Verilog code a testbench is required. Based on the technique described in Figures 4.1 and 4.2, a top-level Verilog file is required that includes an instance of the **see4** module and some test-bench code. This top-level file is shown in Figure 4.33, and an example of a possible testbench in the **testfixture.v** included file is shown in Figure 4.34. These files will be used to demonstrate each of the Verilog simulators in stand-alone mode.

4.3.1 Verilog-XL

Verilog-XL is an interpreted Verilog simulator from Cadence that is the reference simulator for the Verilog-1995 standard. This means that Verilog constructs from later versions of the standard will not run with this simulator. As seen in Section 4.1.1, it is well-integrated with the Composer schematic capture system, but it is easily used on its own too. The inputs to the simulator are, at the simplest, just a list of the Verilog files to simulate, but can also include a dizzying array of additional arguments and switches to control the behavior of the simulator. If you're already set your path to include the standard CAD scripts (see Section 2.2), then you can invoke the Verilog-XL simulator using the script

```
sim-xl <verilogfilename>
```

I find it useful to put the files that you want to simulate in a separate file, and then invoke the simulator with the -f switch to point to that file. In the case of our example from Figures 4.32 to 4.34, the **files.txt** file simply lists the see4.v and seetest.v files and looks like:

```
see4.v
seetest.v
```

In this case, I would invoke the Verilog-XL simulator using the command:

```
sim-xl -f files.txt
```

Anything that you put on the **sim-xl** command line will be passed through to the Verilog-XL simulator so you can include any other switches that you like this way. Try `sim-xl -help` to see a list of some of the switches.

```
// Verilog HDL for "Ax", "see4" "behavioral"
// Four in a row detector - written by Allen Tanner
module see4 (clk, clr, insig, saw4);
   input clk, clr, insig;
   output saw4;

   parameter s0 = 3’b000;  // initial state, saw at least 1 zero
   parameter s1 = 3’b001;  // saw 1 one
   parameter s2 = 3’b010;  // saw 2 ones
   parameter s3 = 3’b011;  // saw 3 ones
   parameter s4 = 3’b100;  // saw at least, 4 ones

   reg [2:0] state, next_state;

   always @(posedge clk or posedge clr)  // state register
     begin
        if (clr) state <= s0;
        else state <= next_state;
     end

   always @(insig or state)  // next state logic
     begin
        case (state)
                s0: if (insig) next_state = s1;
                    else next_state = s0;
                s1: if (insig) next_state = s2;
                    else next_state = s0;
                s2: if (insig) next_state = s3;
                    else next_state = s0;
                s3: if (insig) next_state = s4;
                    else next_state = s0;
                s4: if (insig) next_state = s4;
                    else next_state = s0;
           default: next_state = s0;
        endcase
     end

// output logic
assign saw4 = state == s4;
endmodule //see4
```

Figure 4.32: A simple state machine described in Verilog: **see4.v**

```
\\ Top-level test file for the see4 Verilog code
module test;

\\ Remember that DUT outputs are wires, and inputs are reg
wire  saw4;
reg  clk, clr, insig;

\\ Include the testfixture code to drive the DUT inputs and
\\ check the DUT outputs
‘include "testfixture.v"

\\ Instantiate a copy of the see4 function (named top)
see4 top(clk, clr, insig, saw4);

endmodule //test
```

Figure 4.33: Top-level Verilog code for simulating **see4** named **seetest.v**

If you look inside the **sim-xl** script you will see that it starts a new shell, sources the setup script for Cadence, and then calls Verilog-XL with whatever arguments you supplied. The command puts the log information into a file called **xl.log** in the same directory in which **sim-xl** is called.

If I run this command using the **see4** example, I get the output shown in Figure 4.35. The important parts of the output are the results from the **$display** statements in the test bench. They indicate that the state machine is operating as expected because none of the **ERROR** statements has printed.

Of course, if the testbench checking code is not correct, all bets are off!

If there were errors in the original Verilog code, and the testbench was written to correctly check for faulty behavior then you would get **ERROR** statements printed. As an example, if I modify the **see4** code in Figure 4.32 so that it doesn’t operate correctly, a simulation should signal errors. I’ll change the next-state function in state **s4** so that instead of looping in that state on a **1** the state machine goes back to state **s0**. If I simulate that (faulty) state machine, the **xl.log** file of the simulation prints out a series of **ERROR** statements as shown in Figure 4.36.

Stand-Alone Verilog-XL Simulation with simVision

It’s also possible to run the Verilog-XL simulator in stand-alone mode and also invoke the gui, which includes the waveform viewer. To do this, use the `sim-xlg` script instead of `sim-xl`. The only difference is that the `sim-xlg` invokes the gui (simVision), and starts the simulator in *interactive* mode which means that you can select signals to see in the waveform viewer before starting the simulation. The command would be

```
sim-xlg -f files.txt
```

```
// Four ones in a row detector testbench (testfixture.v)
// Main tests are in an initial block
initial
begin
   clk = 1’b0; // initialize the clock low
   clr = 1’b1; // start with clr asserted
   insig = 1’b0; // insig starts low

   #500 clr = 1’b0; // deassert clear and start running

   // use the send_test task to test the state machine
   send_test(32’b0011_1000_1010_1111_0000_0111_1110_0000);
   send_test(32’b0000_0001_0010_0011_0100_0101_0110_0111);
   send_test(32’b1000_1001_1010_1011_1100_1101_1110_1111);
   send_test(32’b1011_1111_1101_1111_1111_1100_1011_1111);

   // Print something so we know we’re done
   $display("\nSaw4 simulation is finished...");
   $display("If there were no ’ERROR’ statements, then everything worked!\n");
   $finish;
end

// Generate the clock signal
always #50 clk = ~clk;

// this task will take the 32 bit input pattern and apply
// those bits one at a time to the state machine input.
// Bits are changed on negedge so that they’ll be set up for
// the next active (posedge) of the clock.
task send_test;
  input [31:0]pat; // input bits in a 32-bit array
  integer i;       // integer for looping in the for statement
  begin
    for(i=0;i<32; i=i+1) // loop through each of the bits in the pat array
      begin
        // apply next input bit before next rising clk edge
        @(negedge clk)insig = pat[i];

          // remember to check your answers!
          // Look at last four bits to see if saw4 should be asserted
          if ((i > 4)
             && ({pat[i-4],pat[i-3],pat[i-2],pat[i-1]} == 4’b1111)
             && (saw4 != 1))
             $display("ERROR - didn’t recognize 1111 at pat %d,", i);
          else if ((i > 4)
                   && ({pat[i-4],pat[i-3],pat[i-2],pat[i-1]} != 4’b1111)
                   && (saw4 == 1))
                   $display("ERROR - signalled saw4 on %b inputs at step %d",
                            {pat[i-3],pat[i-2],pat[i-1],pat[i]}, i);
      end // begin-for
   end    // begin-task
endtask   // send_test
```

Figure 4.34: Testbench code for **see4.v** in a file named **testfixture.v**

```
--->sim-xl -f test.txt
Tool:   VERILOG-XL      05.10.004-s   Jul 29, 2006  20:50:01

Copyright (c) 1995-2003 Cadence Design Systems, Inc.  All Rights Reserved.
Unpublished -- rights reserved under the copyright laws of the United States.

Copyright (c) 1995-2003 UNIX Systems Laboratories, Inc.  Reproduced with Permission.

THIS SOFTWARE AND ON-LINE DOCUMENTATION CONTAIN CONFIDENTIAL INFORMATION
AND TRADE SECRETS OF CADENCE DESIGN SYSTEMS, INC.  USE, DISCLOSURE, OR
REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN PERMISSION OF
CADENCE DESIGN SYSTEMS, INC. RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to
restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in
Technical Data and Computer Software clause at DFARS 252.227-7013 or
subparagraphs (c)(1) and (2) of Commercial Computer Software --
Restricted
Rights at 48 CFR 52.227-19, as applicable.

                Cadence Design Systems, Inc.
                555 River Oaks Parkway
                San Jose, California  95134

For technical assistance please contact the Cadence Response Center at
1-877-CDS-4911 or send email to support@cadence.com

For more information on Cadence's Verilog-XL product line send email to
talkv@cadence.com

Compiling source file ''see4.v''
Compiling source file ''seetest.v''

Warning!  Code following 'include command is ignored
[Verilog-CAICI]
          ''seetest.v'', 6:
Compiling included source file ''testfixture.v''
Continuing compilation of source file ''seetest.v''
Highest level modules:
test

Saw4 simulation is finished...
If there were no 'ERROR' statements, then everything worked!

L17 ''testfixture.v'': $finish at simulation time 13200
1 warning
0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.0 secs in
simulation
End of Tool:    VERILOG-XL      05.10.004-s   Jul 29, 2006  20:50:02
--->
```

Figure 4.35: Output of stand-alone Verilog-XL simulation of **seetest.v**

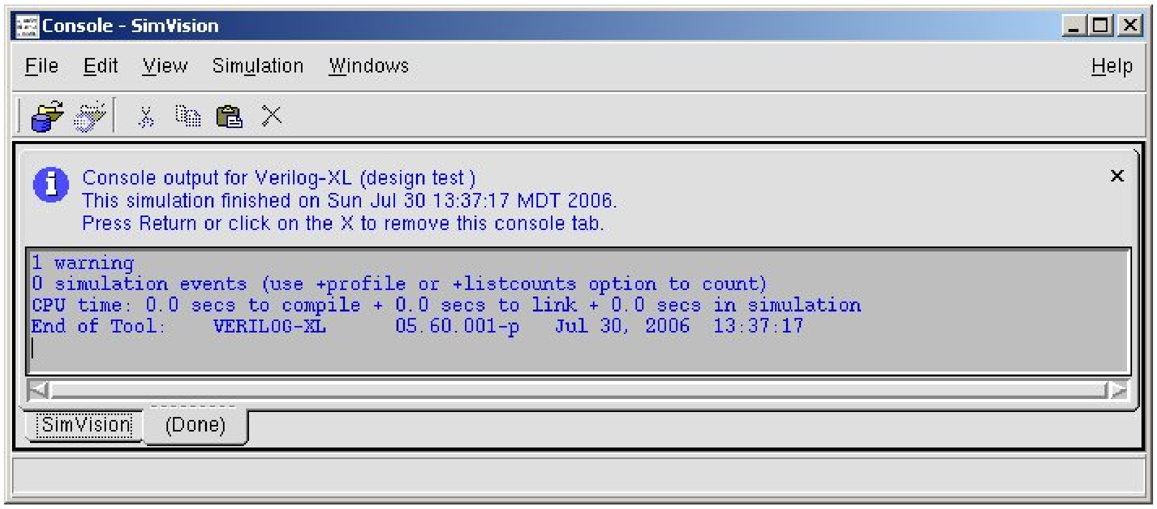
```
<previous text not included...>
Compiling included source file ‘‘testfixture.v’’
Continuing compilation of source file ‘‘seetest.v’’
Highest level modules:
test

ERROR - didn’t recognize 1111 at pat          10,
ERROR - didn’t recognize 1111 at pat          11,
ERROR - didn’t recognize 1111 at pat           5,
ERROR - didn’t recognize 1111 at pat           6,
ERROR - didn’t recognize 1111 at pat          15,
ERROR - didn’t recognize 1111 at pat          16,
ERROR - didn’t recognize 1111 at pat          17,
ERROR - didn’t recognize 1111 at pat          18,
ERROR - didn’t recognize 1111 at pat          20,
ERROR - didn’t recognize 1111 at pat          21,
ERROR - didn’t recognize 1111 at pat          27,
ERROR - didn’t recognize 1111 at pat          28,
ERROR - didn’t recognize 1111 at pat          29,
ERROR - didn’t recognize 1111 at pat          30,

Saw4 simulation is finished...
If there were no ’ERROR’ statements, then everything worked!

L17 ‘‘testfixture.v’’: $finish at simulation time 13200
1 warning
0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.0 secs in
simulation
End of Tool:    VERILOG-XL      05.10.004-s   Jul 29, 2006  21:21:49
```

Figure 4.36: Result of executing a faulty version of **see4.v**



Console output for Verilog-XL (design test)
This simulation finished on Sun Jul 30 13:37:17 MDT 2006.
Press Return or click on the X to remove this console tab.

```
1 warning
0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.0 secs in simulation
End of Tool:    VERILOG-XL      05.60.001-p   Jul 30, 2006  13:37:17
```

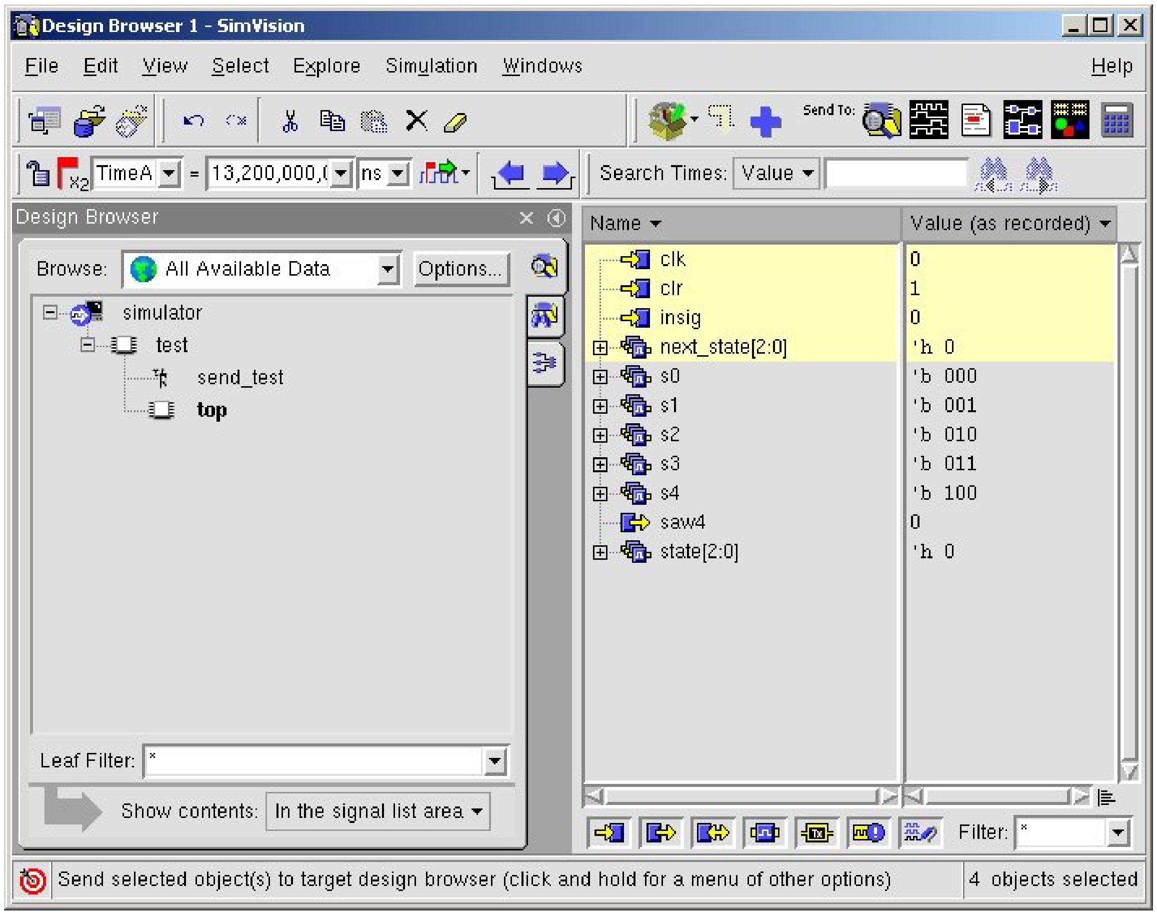
Figure 4.37: Control console for stand-alone Verilog-XL simulation using SimVision

which, if you look inside the script, does exactly the same thing that **sim-xl** does, but adds a couple switches to start up the simulation in the gui in interactive mode. The gui that is used is the same gui environment from Section 4.1.1. Select the signals that you would like to see in the waveform first before starting the simulation with the **play** button (the right-facing triangle). The control console is shown in Figure 4.37, the hierarchy browser in Figure 4.38, and the waveform window (after signals have been selected and the simulation run) in Figure 4.39. See Section 4.1.1 for more details of driving the simVision gui.

4.3.2 NC_Verilog

NC_Verilog is a compiled simulator from Cadence that implements many of the Verilog 2000 features. It compiles the simulator from the Verilog simulation by translating the Verilog to C code and compiling the C code. This results in a much faster simulation time at the expense of extra compilation time at the beginning. This simulator is also well-integrated with the Composer schematic capture tool as seen in Section 4.1.2. At its simplest, the inputs to the NC_Verilog simulator are just a list of files to simulate, but like other Verilog simulators, there are many many switches that can be given at the command line to control the simulation. If you're already set your path to include the standard CAD scripts (see Section 2.2), then you can invoke the NC_Verilog simulator using the script

```
sim-nc <verilogfilename>
```


Design Browser 1 - SimVision
File Edit View Select Explore Simulation Windows Help
Send To:
TimeA = 13,200,000,(ns
Search Times: Value
Design Browser
Browse: All Available Data Options...
simulator
test
send_test
top
Leaf Filter: *
Show contents: In the signal list area
Name
Value (as recorded)
Filter: *

Name	Value (as recorded)
clk	0
clr	1
insig	0
next_state[2:0]	'h 0
s0	'b 000
s1	'b 001
s2	'b 010
s3	'b 011
s4	'b 100
saw4	0
state[2:0]	'h 0

Send selected object(s) to target design browser (click and hold for a menu of other options)   4 objects selected

Figure 4.38: Hierarchy browser for the **see4** example

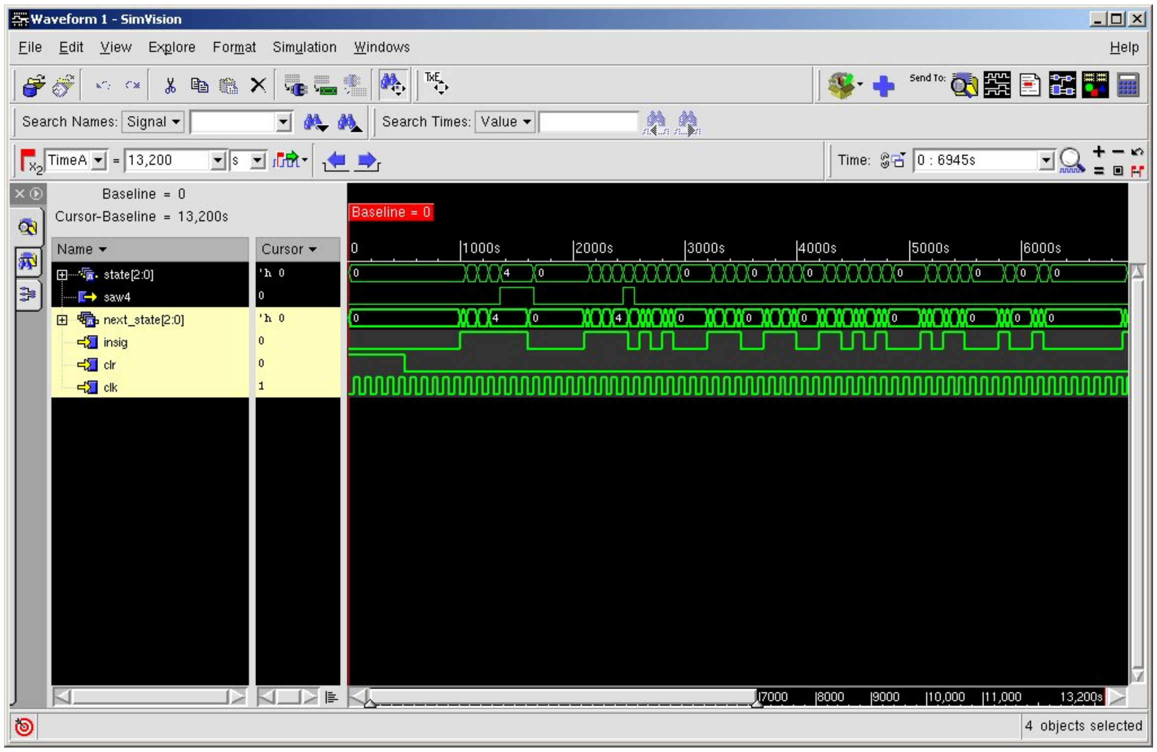


Figure 4.39: Waveform viewer after running the **see4** example

I find it useful to put the files that you want to simulate in a separate file, and then invoke the simulator with the -f switch to point to that file. In the case of our example from Figures 4.32 to 4.34, the **files.txt** file simply lists the see4.v and seetest.v files and looks like:

```
see4.v
seetest.v
```

In this case, I would invoke the NC_Verilog simulator using the command:

`sim-nc -f files.txt`

Anything that you put on the **sim-nc** command line will be passed through to the NC_Verilog simulator so you can include any other switches that you like this way. Try `sim-nc -help` to see a list of some of the switches. If you look inside the **sim-nc** script you will see that it starts a new shell, sources the setup script for Cadence, and then calls NC_Verilog with the command line information that you have supplied. The log file is **nc.log**. If I run this command using the **see4** example, I get the output shown in Figure 4.40. The important parts of the output are the results from the **$display** statements in the test bench. They indicate that the state machine is operating as expected because none of the **ERROR** statements has printed.

If there were errors in the original Verilog code, and the testbench was written to correctly check for faulty behavior then you would get **ERROR** statements printed. These error statements would look similar to those in Figure 4.36.

Of course, if the testbench checking code is not correct, all bets are off!

Stand-Alone NC_Verilog Simulation with simVision

It's also possible to run the NC_Verilog simulator in stand-alone mode and also invoke the gui, which includes the waveform viewer. To do this, use the `sim-ncg` script instead of `sim-nc`. The only difference is that the `sim-ncg` invokes the gui (simVision), and starts the simulator in *interactive* mode which means that you can select signals to see in the waveform viewer before starting the simulation. The command would be

`sim-ncg -f files.txt`

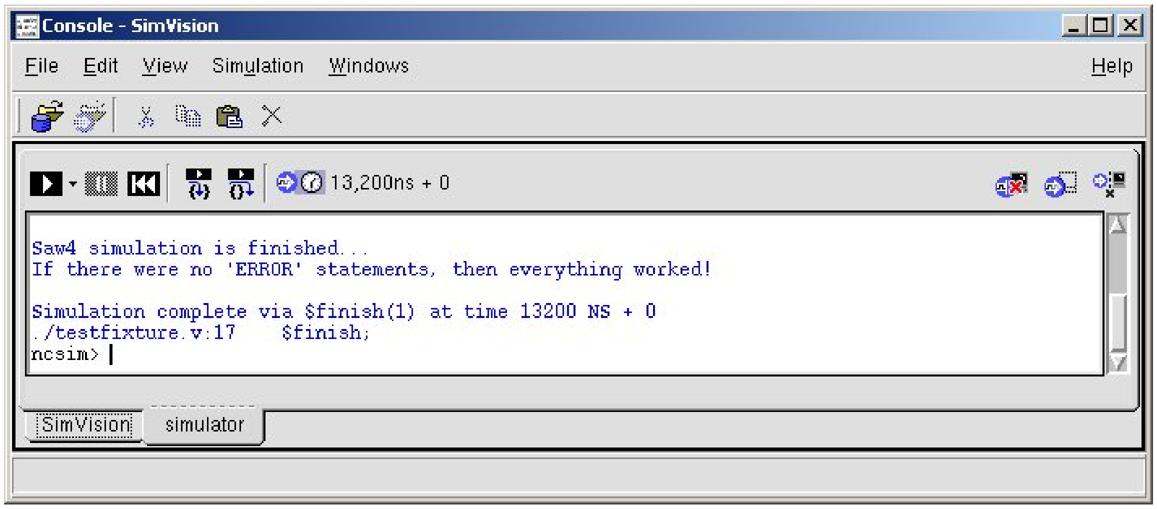
which, if you look inside the script, invokes the same command as does **sim-nc**, but with a switch that starts up the simulation in the SimVision gui. This is the same gui environment from Section 4.1.2. Select the signals that you would like to see in the waveform first before starting the simulation with the **play** button (the right-facing triangle). The control console for the

```
---> sim-nc -f test.files
ncverilog: 05.10-s014: (c) Copyright 1995-2004 Cadence Design Systems,
Inc.
file: see4.v
        module worklib.see4:v
                errors: 0, warnings: 0
file: seetest.v
        module worklib.test:v
                errors: 0, warnings: 0
                Caching library 'worklib' ....... Done
        Elaborating the design hierarchy:
        Building instance overlay tables: .................... Done
        Generating native compiled code:
                worklib.see4:v <0x3d4ece8f>
                        streams:   5, words:  1771
                worklib.test:v <0x37381383>
                        streams:   6, words:  4703
        Loading native compiled code:     .................... Done
        Building instance specific data structures.
        Design hierarchy summary:
                             Instances  Unique
                Modules:             2       2
                Registers:           7       7
                Scalar wires:        4       -
                Always blocks:       3       3
                Initial blocks:      1       1
                Cont. assignments:   1       1
                Pseudo assignments:  3       4
        Writing initial simulation snapshot: worklib.test:v
Loading snapshot worklib.test:v .................... Done
ncsim> source
/uusoc/facility/cad_tools/Cadence/LDV/tools/inca/files/ncsimrc
ncsim> run

Saw4 simulation is finished...
If there were no 'ERROR' statements, then everything worked!

Simulation complete via $finish(1) at time 13200 NS + 0
./testfixture.v:17    $finish;
ncsim> exit
--->
```

Figure 4.40: Output of stand-alone NC_Verilog simulation of **seetest.v**



```
Saw4 simulation is finished...
If there were no 'ERROR' statements, then everything worked!

Simulation complete via $finish(1) at time 13200 NS + 0
./testfixture.v:17    $finish;
ncsim>
```

Figure 4.41: Control console for NC_Verilog through SimVision

NC_Verilog simulation is shown in Figure 4.41. The hierarchy and waveform windows will look the same as the Verilog-XL version in Figures 4.38 and 4.39. See Section 4.1.1 for more details of driving the simVision gui.

4.3.3 vcs

The Verilog simulator from Synopsys is vcs This isn't integrated with the Cadence dfII tool suite, but is a very capable simulator in its own right so it's important to know about if you're using more of the Synopsys tools than Cadence tools. The vcs simulator is a compiled simulator so it runs very fast once the simulator is compiled, and is compatible with Verilog-2000 features. The inputs to the simulator are, at the simplest, just a list of the Verilog files to simulate, but can also include a dizzying array of additional arguments and switches to control the behavior of the simulator. If you're already set your path to include the standard CAD scripts (see Section 2.2), then you can invoke the vcs simulator using the script

More correctly we are using vcs_mx which is the "mixed mode" version that can support both Verilog and VHDL, but I'll just call it vcs because we're mostly interested in the Verilog version.

```
sim-vcs <verilogfilename>
```

I find it useful to put the files that you want to simulate in a separate file, and then invoke the simulator with the -f switch to point to that file. In the case of our example from Figures 4.32 to 4.34, the **files.txt** file simply lists the see4.v and seetest.v files and looks like:

```
see4.v
seetest.v
```

In this case, I would invoke the vcs simulator using the command:

```
sim-vcs -f files.txt
```

Anything that you put on the **sim-vcs** command line will be passed through to the vcs simulator so you can include any other switches that you like this way. Try `sim-vcs -help` to see a list of some of the switches. If you look inside the **sim-vcs** script you will see that it starts a new shell, sources the setup script for Synopsys, and then calls vcs with the command line information you have supplied. The log file is **vcs.log**. If I run this command using the **see4** example, I get the output shown in Figure 4.42.

For the vcs simulator, running the **sim-vcs** script doesn't actually run the simulation. Instead it compiles the Verilog code into an executable called **simv**. After compiling this simulator, you can run it by running `simv` to get the output seen in Figure 4.43, but the **simv** executable needs some setup information about Synopsys before it can run, so rather than run **simv** directly, you will run it through a wrapper called **sim-simv** as follows:

```
sim-simv <executable-name>
```

Because the executable will be named **simv** unless you've overridden that default name with a command-line switch, this will almost always be called as:

```
sim-simv simv
```

The important parts of the output are the results from the **$display** statements in the test bench. They indicate that the state machine is operating as expected because none of the **ERROR** statements has printed.

Of course, if the testbench checking code is not correct, all bets are off!

If there were errors in the original Verilog code, and the testbench was written to correctly check for faulty behavior then you would get **ERROR** statements printed. These error statements would look similar to those in Figure 4.36.

Stand-Alone vcs Simulation with VirSim

It's also possible to run the vcs simulator in stand-alone mode and also invoke the gui, which includes the waveform viewer. To do this, use the `sim-vcsg` script instead of `sim-vcs`. The only difference is that the `sim-vcsg` invokes the gui (VirSim), and starts the simulator in *interactive* mode which means that you can select signals to see in the waveform viewer before starting the simulation. The command would be

```
sim-vcsg -f files.txt
```

The only real difference in the **sim-vcsg** script from the **sim-vcs** script

```
---> sim-vcs -f test.files
                         Chronologic VCS (TM)
          Version X-2005.06-SP2 -- Sat Jul 29 21:49:35 2006
               Copyright (c) 1991-2005 by Synopsys Inc.
                         ALL RIGHTS RESERVED

This program is proprietary and confidential information of Synopsys
Inc.
and may be used and disclosed only as authorized in a license
agreement
controlling such use and disclosure.

Parsing design file 'see4.v'
Parsing design file 'seetest.v'
Parsing included file 'testfixture.v'.
Back to file 'seetest.v'.
Top Level Modules:
       test
No TimeScale specified
Starting vcs inline pass...
1 module and 0 UDP read.
recompiling module test
make: Warning: File `filelist' has modification time 41 s in the
future
if [ -x ../simv ]; then chmod -x ../simv; fi
g++  -o ../simv -melf_i386 -m32  5NrI_d.o 5NrIB_d.o gzYz_1_d.o SIM_l.o
/uusoc/facility/cad_tools/Synopsys/vcs/suse9/lib/libvirsim.a
/uusoc/facility/cad_tools/Synopsys/vcs/suse9/lib/libvcsnew.so
/uusoc/facility/cad_tools/Synopsys/vcs/suse9/lib/ctype-stubs_32.a -ldl
-lc -lm -ldl
/usr/lib64/gcc/x86_64-suse-linux/4.0.2/../../../../x86_64-suse-linux/bin/ld:
warning: libstdc++.so.5, needed by
/uusoc/facility/cad_tools/Synopsys/vcs/suse9/lib/libvcsnew.so, may
conflict with libstdc++.so.6
../simv up to date
make: warning:  Clock skew detected.  Your build may be incomplete.
CPU time: .104 seconds to compile + .384 seconds to link
--->
```

Figure 4.42: Output of running **sim-vcs** on **files.txt**

```
---> sim-simv simv
Chronologic VCS simulator copyright 1991-2005
Contains Synopsys proprietary information.
Compiler version X-2005.06-SP2; Runtime version X-2005.06-SP2;  Jul 29
21:49 2006


Saw4 simulation is finished...
If there were no 'ERROR' statements, then everything worked!

$finish at simulation time                13200
           V C S   S i m u l a t i o n   R e p o r t
Time: 13200
CPU Time:      0.090 seconds;       Data structure size:   0.0Mb
Sat Jul 29 21:49:54 2006
--->
```

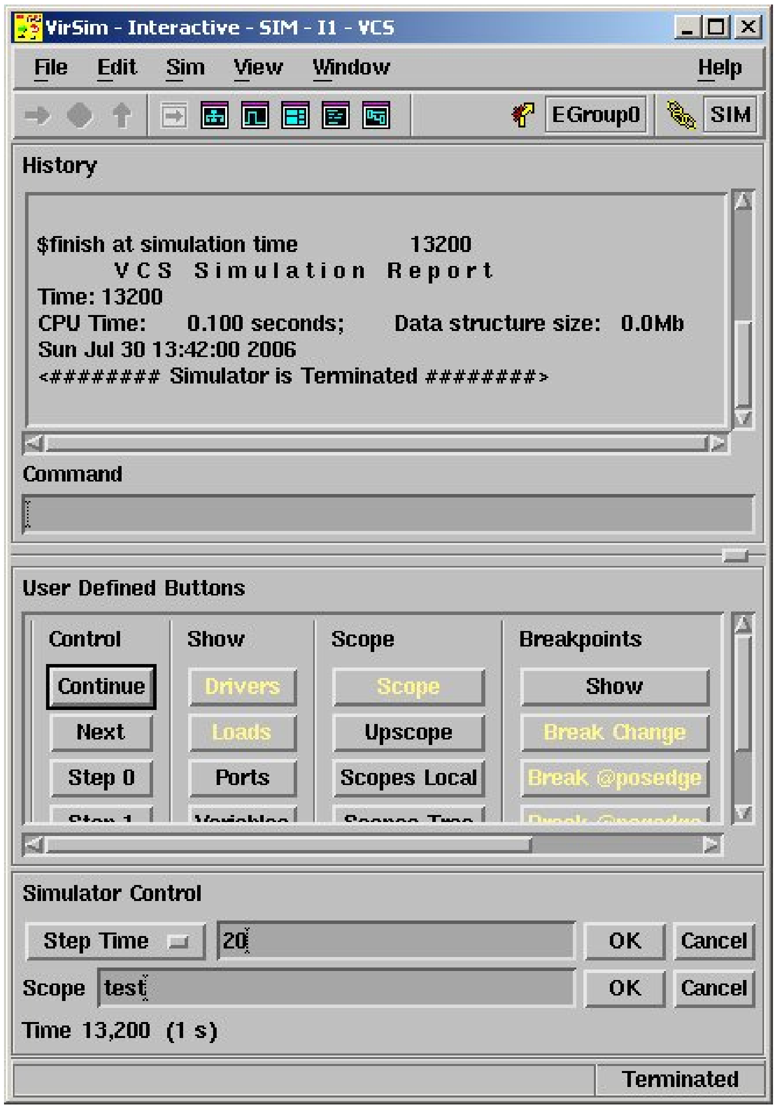
Figure 4.43: Output of stand-alone vcs simulation of **seetest.v** using the compiled **simv** simulator

is that it adds a switch to "Run Interactive" after compilation through the VirSim gui environment. Using the **sim-vcsg** script on the **see4** example you would see the window in Figure 4.44 pop up. This is the control console for the Synopsys interactive simulator VirSim. I would first open up a hierarchy window using the **Window** → **Hierarchy** menu choice, the `ctl-shft-H` hotkey, or the **New Hierarchy Browser** widget. You can use this window (shown in Figure 4.45) to select the signals that you would like to track in the simulation. You can also open a waveform window using the **Window** → **Waveform** menu, the `ctl-shft-W` hotkey, or the **New Waveform Window** widget. Signals selected in the hierarchy window can be added to the waveform window using the **add** button.

Once the signals that you want to track have been added to the waveform window you can run the simulation using the **continue** command (menu or widget). Figure 4.46 shows the waveform window after signals have been added from the hierarchy window to the waveform window and the simulation has been run using the **continue** button.

4.4 Timing in Verilog Simulations

Time is modeled in a Verilog simulation either by explicit delay statements or through implicit constructs that wait for an event or an edge of a signal before progressing. Explicit delays are denoted with the **#** character. You've seen the **#** character used in the testbench code examples in this chapter. The simplest syntax is **#10** which means to delay 10 time units before proceeding



```
$finish at simulation time           13200
       V C S   S i m u l a t i o n   R e p o r t
Time: 13200
CPU Time:     0.100 seconds;       Data structure size:   0.0Mb
Sun Jul 30 13:42:00 2006
<######## Simulator is Terminated ########>
```

Step Time: 20
Scope: test
Time 13,200 (1 s)

Figure 4.44: Console window for controlling a vcs simulation through VirSim

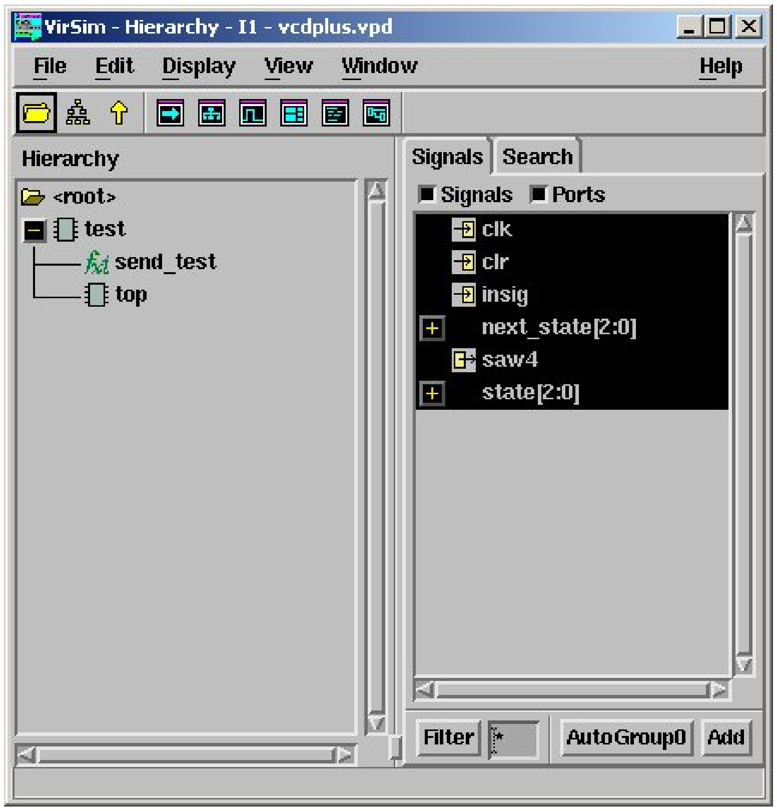
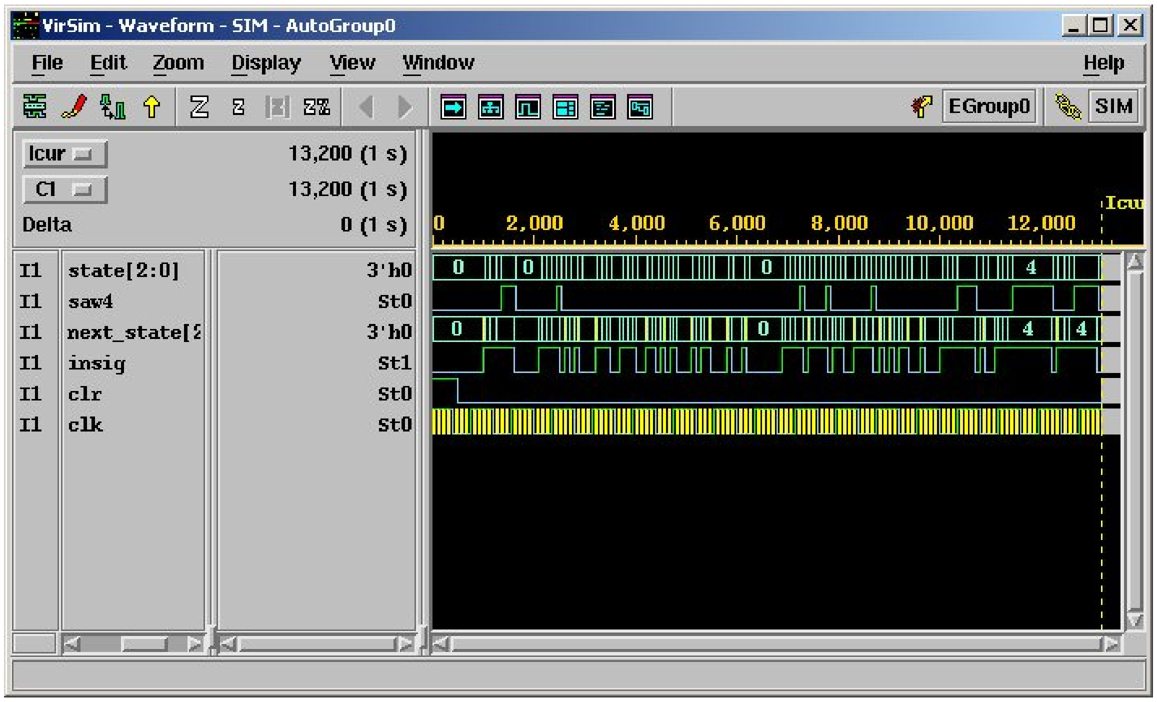


Figure 4.45: Hierarchy browser window from VirSim



```
I1  state[2:0]     3'h0
I1  saw4           St0
I1  next_state[2   3'h0
I1  insig          St1
I1  clr            St0
I1  clk            St0
```

Figure 4.46: Waveform window for VirSim after running the **see4** simulation

with the simulation and executing the following statement. You can also use the **#8:10:12** syntax to indicate minimum, typical, and maximum delays. If delays are specified in this way you can choose which delay to use in a given simulation using command-line switches to the Verilog simulator. Typically (and for the three simulators we use) this is accomplished with a **+mindelays**, **+typdelays** or **+maxdelays** command-line argument.

Of course, if the **#10** construct delays by 10 time units, it is important to know if those time units correspond to any real time units or are simply *unit delay* timings that are not meant to model any specific real time values. Verilog uses the **'timescale** command to specify the connection between timing units and real timing numbers. The **'timescale** directive takes two arguments: a timing unit and a timing precision. For example,

'timescale 1 ns / 100 ps

means that each "unit" corresponds to 1ns, and the precision at which timings are computed is 100ps. But be careful! For many simulations the delay numbers do **not** correspond to actual real times and are simply generic "unit timings" to model the fact that time passes without trying to model the exact amount of time the activity takes. Make sure you understand how timing is used in your simulation before you make assumptions!

Implicit delays in a Verilog simulation are signaled using the **@** character which causes the following statement or procedural block to wait for some event before proceeding. The even might be a signal transition on a wire, an edge (**posedge** or **negedge**) of a signal change, or an abstract *event* through the Verilog **event** construct. These implicit delays are often used, for example, to wait for an active clock edge before proceeding.

4.4.1 Behavioral versus Transistor Switch Simulation

In our CAD flow there are two main types of Verilog simulations:

Behavioral: The Verilog describes the desired behavior of the system in high-level terms. This description does not correspond directly to hardware, but can be synthesized into hardware using a synthesis tool (Chapter 8).

Structural: The Verilog consists of instantiations of primitive gates from a standard cell library. Each of the gates corresponds to a leaf cell from the library that can be placed and routed on the completed chip (Chapter 10). The structural Verilog can be strictly textual (Verilog code), or a hierarchical schematic that uses gate symbols from the standard cell library.

If you're simulating behavior only with high-level behavioral Verilog, then the timing in your simulation depends on the timing that you specify in your code using either explicit (**#10**) or implicit (**@(posedge clk)**) statements in your Verilog code. If you're simulating structural code then you (or, hopefully, the tools) need to generate a simulatable netlist of the structural code where each leaf cell is replaced with the Verilog code that defines the function of that cell.

A schematic may be strictly structural if all the leaf cells of the hierarchical schematic are standard cells, or it can contain symbols that encapsulate behavioral code using the behavioral modeling techniques in Section 4.2.

Given this way of thinking about Verilog descriptions of systems, it is easy to apply this to the standard cells themselves. Each of the standard cells in our standard cell library have two different descriptions which are instantiated in two different cell views in the Cadence library:

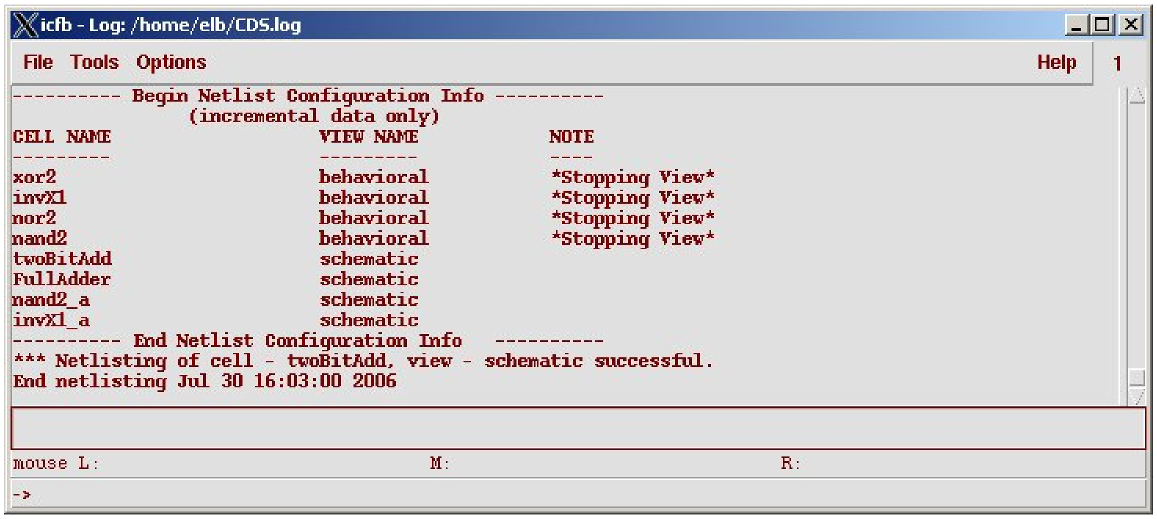
behavioral: In this cell view there is Verilog behavioral code that describes the behavior of the cell. An example of a behavioral view of a standard cell is shown in Figure 4.30. Note that this behavioral description includes unit delays for the **nand2** cell using a **specify** block to specify the input to output delays for that cell as **1.0** units in both the min and max cases.

cmos_sch: In this schematic view the library cells are described in terms of their transistor networks using transistor models from the analog cells libraries. Timing in a simulation of this view would depend on the timing associated with each transistor model.

The CMOS transistors that are the primitives in the **cmos_sch** leaf cell views have their behavior described in a separate cell view:

functional: In this view each transistor is described using a built in Verilog *switch-level* model of a transistor. Switch-level means that the transistor is modeled as a simple switch that is closed (conducting) from source to drain on one value of the gate input, and open (non-conducting) for the other value of the gate input. Using these models results in a *switch level simulation* of the hardware. This models the detailed behavior of the system is simulated with each transistor modeled as an ideal switch. This type of simulation is is more accurate (in some sense) than the behavioral model, but not as accurate (or as time consuming) as a more detailed analog transistor simulation.

If each of the cells in the standard cell library has two different views that can be used to simulate the behavior of the cell, then you should be able



```
---------- Begin Netlist Configuration Info ----------
                (incremental data only)
CELL NAME                   VIEW NAME           NOTE
---------                   ---------           ----
xor2                        behavioral          *Stopping View*
invX1                       behavioral          *Stopping View*
nor2                        behavioral          *Stopping View*
nand2                       behavioral          *Stopping View*
twoBitAdd                   schematic
FullAdder                   schematic
nand2_a                     schematic
invX1_a                     schematic
---------- End Netlist Configuration Info   ----------
*** Netlisting of cell - twoBitAdd, view - schematic successful.
End netlisting Jul 30 16:03:00 2006
```

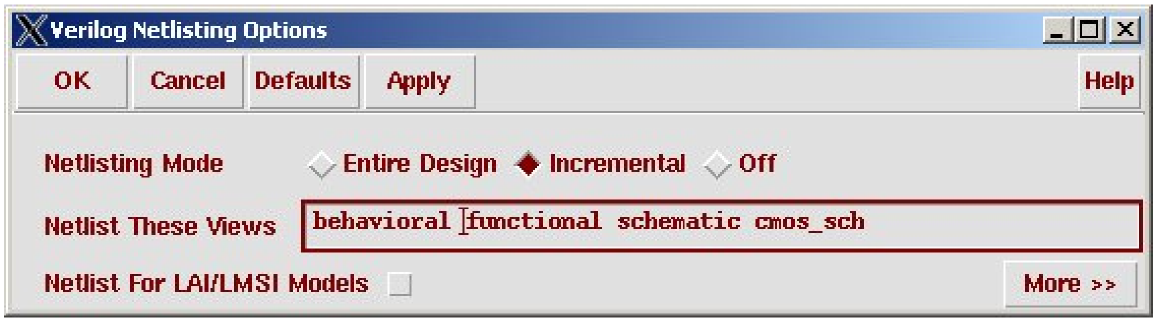
Figure 4.47: A Netlisting Log for the Two Bit Adder that stops at behavioral views of the standard cell gates

to modify how the simulation proceeds by choosing a different view for each cell when you netlist the schematic. If you expand the schematic to the **behavioral** views of the standard cells you will get one flavor of simulation, and if you expand through the **cmos_sch** views all the way to the **functional** views of each transistor you will get a different flavor of simulation of the same circuit. This is indeed possible.

Suppose you were simulating a schematic with either Verilog-XL or NC_Verilog. The netlisting log might look like that in Figure 4.47. This is a repeat of the netlist for the two bit adder from Section 4.1.1 and shows that as the netlister traversed the hierarchy of the circuit it stopped when it found leaf cells with **behavioral** views. If this simulation were to run, it would use the behavioral code in those behavioral views for the simulation of the cells. How did the netlisted decide to stop at the behavioral views? That is controlled by the **verilogSimViewList** and **verilogSimStopList** variables that control the netlister. Those values are set to default values in the **.simrc** file in the class directory. If you looked in that file you would see that they are set to:

```
verilogSimViewList = '("behavioral" "functional" "schematic" "cmos_sch")
verilogSimStopList = '("behavioral" "functional")
```

This means that the netlister will look at all the views in the View list, but stop when it finds a view in the Stop list. You can modify these lists before the netlisting phase of each of the Cadence Verilog simulators used



Netlisting Mode: Entire Design [✓ Incremental] Off

Netlist These Views: behavioral functional schematic cmos_sch

Netlist For LAI/LMSI Models: ☐

Figure 4.48: The **Setup Netlist** dialog from Verilog-XL

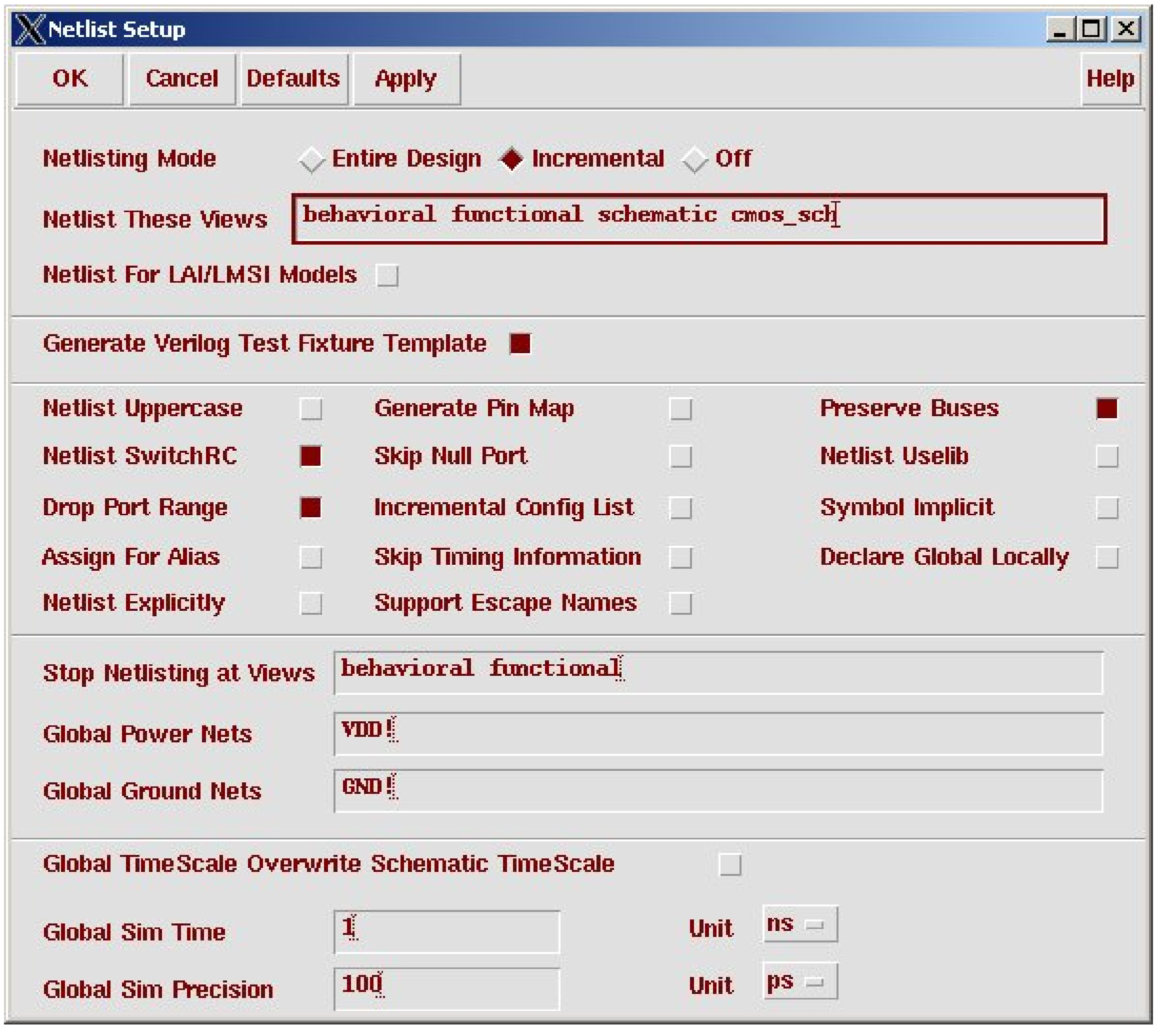
with the Composer schematic capture tool. Using either Verilog-XL or NC_Verilog you can modify the **verilogSimViewList** before generating the netlist. Use the **Setup** → **Netlist** menu choice. In Verilog-XL you'll see the window in Figure 4.48, and in NC_Verilog you'll see the window in Figure 4.49. In either case you can remove **behavioral** from the **Netlist These Views** list. The result of this change and then generating the netlist is seen in Figure 4.50 where the netlister has ignored the **behavioral** views, descended through the **cmos_sch** views, and stopped at the **functional** views of the **nmos** and **pmos** transistors.

4.4.2 Behavioral Gate Timing

If you're stopping the netlisting process at the **behavioral** views then any timing information in those behavioral views will be the timing used for the simulation. There are a number of ways to encode timing in the behavioral descriptions. One way to get timing information into a behavioral simulation is to include timing information explicitly into your behavioral descriptions. For example, using hash notation like **#10** will insert 10 units of delay into your behavioral simulation at that point in the Verilog code. This can enable a rough top-level estimate of system timing in a description that is a long ways from a real hardware implementation. An example of using explicit timing in a behavioral description is shown in Figure 4.51 for a two-input NAND gate. Another style of description with procedural assignment of the NAND function is shown in Figure 4.52.

Be aware, though, that **#10**-type timing is ignored for synthesis. Synthesis takes timing into account using the gate timing of the cells in the target library. It does not try to impose any timing that you specify in the behavioral description. This is covered in more detail in Chapter 8.

You can also put parameters in your descriptions so that you can override



Netlisting Mode: Entire Design [✓ Incremental] Off

Netlist These Views: behavioral functional schematic cmos_sch

Netlist For LAI/LMSI Models ☐

Generate Verilog Test Fixture Template ■

Netlist Uppercase	☐	Generate Pin Map	☐	Preserve Buses	■
Netlist SwitchRC	■	Skip Null Port	☐	Netlist Uselib	☐
Drop Port Range	■	Incremental Config List	☐	Symbol Implicit	☐
Assign For Alias	☐	Skip Timing Information	☐	Declare Global Locally	☐
Netlist Explicitly	☐	Support Escape Names	☐		

Stop Netlisting at Views: behavioral functional

Global Power Nets: VDD!

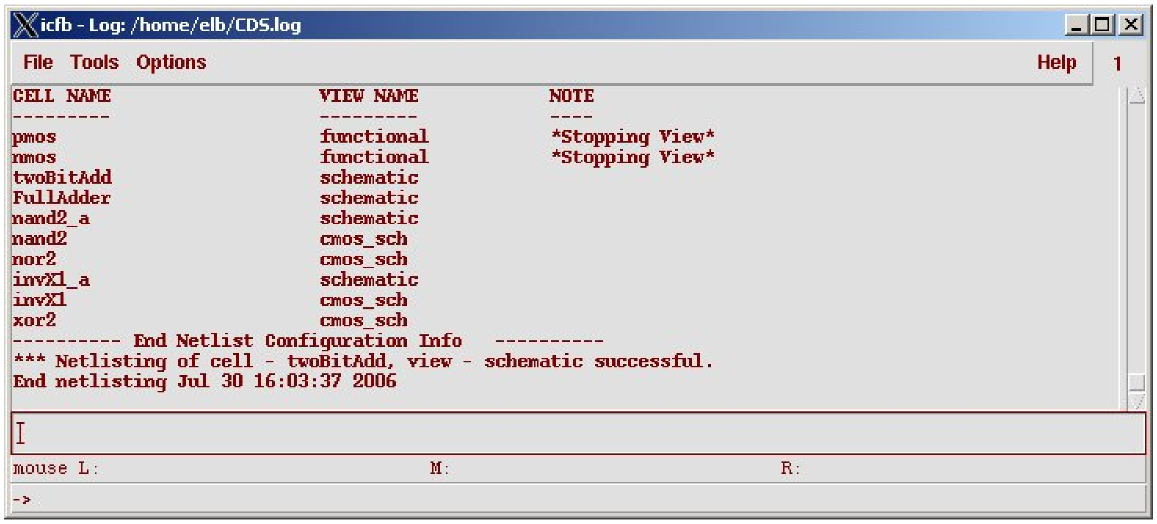
Global Ground Nets: GND!

Global TimeScale Overwrite Schematic TimeScale ☐

Global Sim Time: 1 Unit: ns

Global Sim Precision: 100 Unit: ps

Figure 4.49: The **Setup Netlist** dialog from NC_Verilog



```
CELL NAME                    VIEW NAME            NOTE
---------                    ---------            ----
pmos                         functional           *Stopping View*
nmos                         functional           *Stopping View*
twoBitAdd                    schematic
FullAdder                    schematic
nand2_a                      schematic
nand2                        cmos_sch
nor2                         cmos_sch
invX1_a                      schematic
invX1                        cmos_sch
xor2                         cmos_sch
----------- End Netlist Configuration Info   ----------
*** Netlisting of cell - twoBitAdd, view - schematic successful.
End netlisting Jul 30 16:03:37 2006
```

Figure 4.50: Netlisting result after removing **behavioral** from the **verilogSimViewList**

```
module NAND (out, in1, in2);
    output out;
    input in1, in2;

    assign #10 out = ~(in1 & in2);

endmodule
```

Figure 4.51: Verilog description of a NAND gate with explicit timing

```
module NAND (out, in1, in2);
    output out;
    reg out;
    input in1, in2;

    always @(in1 or in2)
       begin
         #10 out = ~(in1 & in2);
       end
endmodule
```

Figure 4.52: Another description of a NAND gate with explicit timing

```
module NAND (out, in1, in2);
    output out;
    reg out;
    input in1, in2;
    parameter delay = 10;

    always @(in1 or in2)
       begin
         #delay out = ~(in1 & in2);
       end
endmodule
```

Figure 4.53: NAND description with **delay** parameter

the default values when you instantiate the module. For example, the NAND in Figure 4.53 has a default delay of 10 defined as a **delay** parameter. This can be overridden when the NAND is instantiated using the syntax

```
NAND #(5) _io(a,b,c);
```

which would override the default and assign a delay of 5 units to the NAND function.

Behavioral Verilog code can also include timing in a **specify** block instead of in each assignment statement. A **specify** block allows you to define *path delays* from input pins to output pins of a module. Path delays are assigned in Verilog between **specify** and **endspecify** keywords. Statements between these keywords constitute a specify block. A specify block appears at the top level of the module, not within any other initial or always block. Path delays define timing between inputs and outputs of the module without saying anything about the actual circuit implementation of the module. They are simply overall timings applied at the I/O interface of the module. An example of our NAND with a **specify** block is shown in Figure 4.54. In this case the delays are specified with separate rising and falling delays. If only one number is given in a **specify** description is is used for both rising and falling transitions.

4.4.3 Standard Delay Format (SDF) Timing

There are two main reasons to use a specify block to describe timing

1. So that you can describe path timing between inputs and outputs of complex modules without specifying detailed timing of the specific circuits used to implement the module

```
module nand2 (Y, A, B);
   output Y;
   input A;
   input B;

   nand _i0 (Y, A, B);

   specify
      (A => Y) = (1.5, 1.0);
      (B => Y) = (1.7, 1.2);
   endspecify

endmodule
```

Figure 4.54: NAND gate description with **specify** block

2. So that you can back-annotate circuits that use this cell with timing information from a synthesis tool.

Synthesis tools can produce a timing information file as output from the synthesis process in *sdf* format (standard delay format). An sdf file will have extracted timings for all the gates in the synthesized circuit. Synthesis programs from Synopsys and Cadence get these timings from the *.lib* file that describes each cell in the library and from an estimate of wiring delays so they're pretty accurate. They also assume that every cell in the library has a specify block that specifies the delay from each input to each output! It is those **specify** statements that are updated with new data from the **sdf** files.

Details on generating an sdf file are in Chapter 8 on Synthesis, but assuming that you have an sdf file, you can annotate your verilog file with this timing information. This will override all the default values in the specify blocks with the estimated values from the sdf file. For example, a snippet of an sdf file for a circuit that uses a NAND gate from the preceeding figures would look like:

```
(CELL
(CELLTYPE "NAND")
(INSTANCE U21)
(DELAY
(ABSOLUTE
(IOPATH A Y (0.385:0.423:0.423) (0.240:0.251:0.251))
(IOPATH B Y (0.397:0.397:0.397) (0.243:0.243:0.243))
)
)
)
```

Notice the matching of the timing paths in the specify bock to the IOPATH statements in the sdf file, and the the timings are specified in min:typ:max form for both rising and falling transitions. Each instance of a NAND in the circuit would have its own CELL block in the sdf file and thus get its own timing based on the extracted timing done by Synopsys during synthesis. Remember, though, that this is just estimated timing. The timing estimates come from the characterizations of each library cell in the .lib file. You can generate this sdf information after synthesis, or after synthesis and place and route. The sdf information after place and route will include timing information based on the wiring of the circuit and therefore be more accurate than the pre place and route timing.

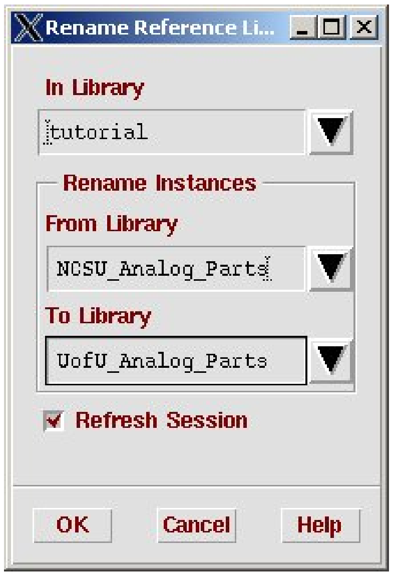
Details on how to integrate the sdf timing in Verilog-XL and NC_Verilog simulations will be postponed until Chapter 8 so that we can have structural Verilog simulations and sdf files to use as examples.

4.4.4 Transistor Timing

When you use individual **nmos** and **pmos** transistors in your schematics (as described in Section 3.3), and you simulate those schematics using Verilog-XL or NC_Verilog, what timing is associated with those primitive switch-level transistor models? That depends on how the transistor models are defined. In our case we have two choices:

1. We can use transistors from the **NCSU_Analog_Parts** library in which case the transistors are modeled as *zero delay* switches. There is no delay associated with switching of the transistors.
2. We can use transistors from the **UofU_Analog_Parts** library in which case there are 0.1 units of delay associated with each transistor. This is supposed to very roughly correspond to 0.1ns of delay for each transistor which is very roughly similar to the delay in transistors in the 0.5μ CMOS process that we can use through MOSIS for free class chip fabrication.

Fortunately (by design) each of these libraries has exactly the same names for each of the transistors (**nmos, pmos, r_nmos, r_pmos, bi_nmos**, and **bi_pmos** are the most commonly used devices for digital circuits). This means that you can control whether you are getting the versions with zero delay or the versions with 0.1 units of delay by changing which library you are using. There is a menu choice in the Library Manager which can be used to change the reference library name for a whole library. That is, using **Edit → Rename Reference Library ...** (see the dialog box in Figure 4.55) you can change all references to gates in library A to references of that same



In Library: tutorial

Rename Instances

From Library: NCSU_Analog_Parts

To Library: UofU_Analog_Parts

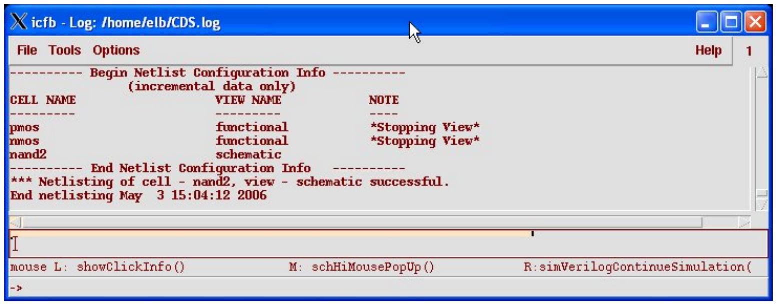
[✓] Refresh Session

Figure 4.55: **Rename Reference Library** dialog box from Library Manager

gate in library B. The figure shows how you could change all transistor references from the **NCSU_Analog_Parts** library to the same transistors in the **UofU_Analog_Parts** library and therefore switch between zero delay switch level simulation and unit (0.1 units) delay switch level simulation.

Consider simulating the NAND gate designed as a schematic in Figure 3.12) using the 0.1ns delays on each transistor using the **UofU_Analog_Parts** library. In this case you will open the nand2 schematic and simulate using Verilog-XL. The netlist result in the CIW (The result is shown in Figure 4.56) shows that in this case the netlisting has proceeded through the nand2 and down to the **functional** views of the **nmos** and **pmos** devices. These **functional** views are the switch models of the transistor devices. A portion of the simulation waveform for this circuit is shown in Figure 4.57.

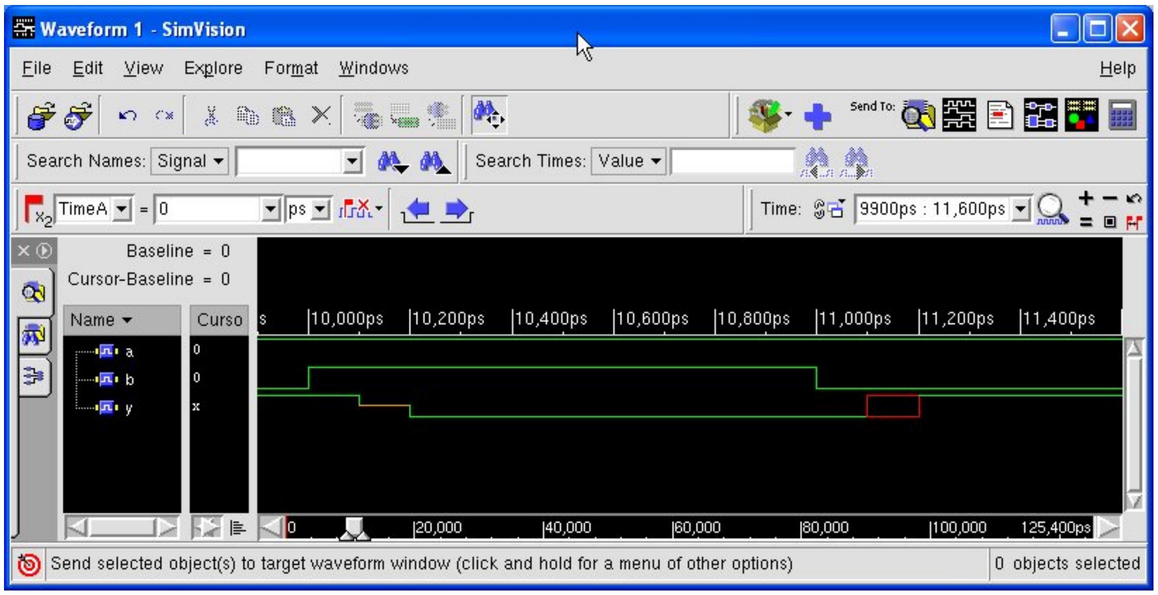
What's going on in this waveform? The **y** output signal should behave in a nice digital way but it looks like on one transition of the **b** input it's going to a high-impedance (**Z**) value (shown by the orange trace in the middle of the high and low ranges) for a while. On another edge of **b** the **y** output is going to an unknown (**X**) value (shown by the red box covering both high and low values). This is happening because of the delays at each transistor switch. If you are using the zero-delay transistors of the **NCSU_Analog_Parts** library you won't see this effect, but you will see a yellow circle warning and a red transition to show that things are glitching in zero time.



```
---------- Begin Netlist Configuration Info ----------
          (incremental data only)
CELL NAME                VIEW NAME          NOTE
---------                ---------          ----
pmos                     functional         *Stopping View*
nmos                     functional         *Stopping View*
nand2                    schematic
---------- End Netlist Configuration Info  ----------
*** Netlisting of cell - nand2, view - schematic successful.
End netlisting May  3 15:04:12 2006
```

```
mouse L: showClickInfo()          M: schHiMousePopUp()          R:simVerilogContinueSimulation(
->
```

Figure 4.56: Netlist Log for the nand2 Cell



Send selected object(s) to target waveform window (click and hold for a menu of other options)   0 objects selected

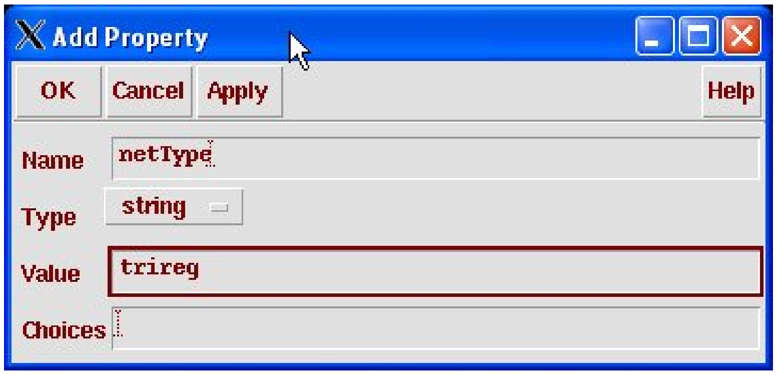
Figure 4.57: Waveform from Simulation of nand2 Cell

The reason that adding a small delay on each transistor causes this behavior is that there are serial pulldown **nmos** transistors pulling down the **y** output. When these transistors switch, it takes time for the result of the switching to be passed to the transistor's output. During the time that the transistor is switching, the output is unknown, or is unconnected. This is undesired behavior because the **Z** and **X** values can propagate to the rest of the circuit and cause all sorts of strange behavior that isn't a true reflection of the circuit's behavior. The issue is that in a real circuit it is true that the output might be unconnected for a brief amount of time during switching, but the parasitic capacitance on the output will hold the output at a fixed value during the switching.

To model this behavior in a switch-level simulation we need some way to tell Verilog to emulate this behavior. That is, we need to make the output node of the nand2 gate "sticky" so that when it is unconnected it sticks to the value it had before it was unconnected. This type of wire in Verilog is a **trireg** wire. It is still a wire, but it's a "sticky" wire whose value sticks when the driver is disconnected from the wire. It acts like a wire from the point of view of the Verilog code (i.e. it can be driven by a continuous assignment), but it acts like a register in terms of the simulation (the value sticks). To inform Composer that it should use a **trireg** wire on the output of the nand2 instead of the plain wire type we need to add an **attribute** to the wire.

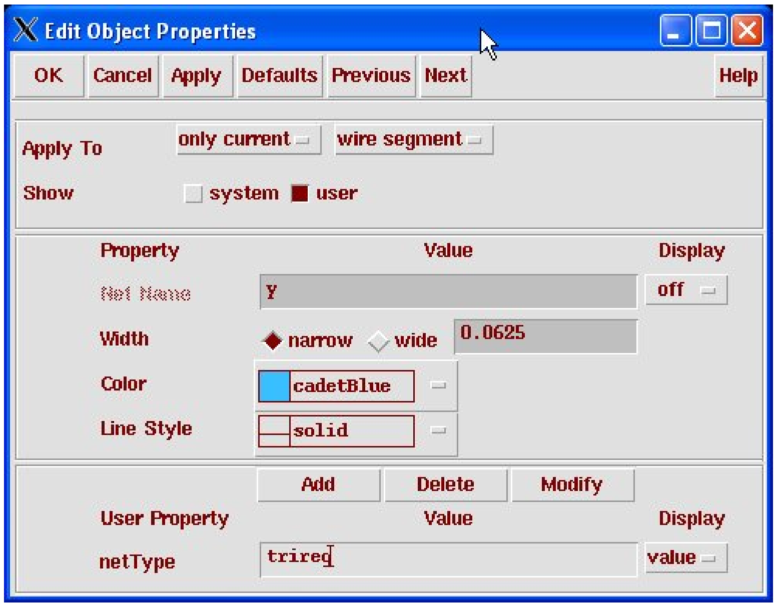
Select the output **y** wire in the schematic and get its properties with the **q** hotkey (or the properties widget). At the bottom of this dialog box you can **Add** a new property to this wire. As shown in Figure 4.58 add a property named **netType** (make sure to get the spelling correct with a capital "T"), type **string**, and value **trireg**. The result is seen in the properties box shown in Figure 4.59. I've also changed the **Display** to **Value** so that the **trireg** value can be seen on the schematic to remind you that this additional property has been set.

In order for the Composer netlister to pay attention to this new property, you also need to set the **Switch RC** property in the netlisting options in Verilog-XL, but this should have been already set for you by the default **.simrc** file. Once you have the **trireg** property on the output node, the output will no longer go to a **Z** value. If the output is ever un-driven it will hold its previous value. Unfortunately, transitions that caused an **X** in the original simulation will still cause an **X** in the nand2 even with the **trireg**. This is because of the delay on the transistors again. There are situations where the pmos devices have turned on but the series nmos stack hasn't turned off yet because of the delay through each device. What you need now is a transistor at the top of the series stack (nearest the output node) that is weak enough to be over-driven by the pmos pullup if they're ever on at the same time.



Name netType
Type string
Value trireg
Choices

Figure 4.58: Adding a Property to a Net



Apply To only current wire segment
Show system [✓ user]

Property	Value	Display
Net Name	y	off
Width	[✓ narrow] wide 0.0625	
Color	cadetBlue	
Line Style	solid	

User Property	Value	Display
netType	trireg	value

Figure 4.59: A Net With a **netType** Property

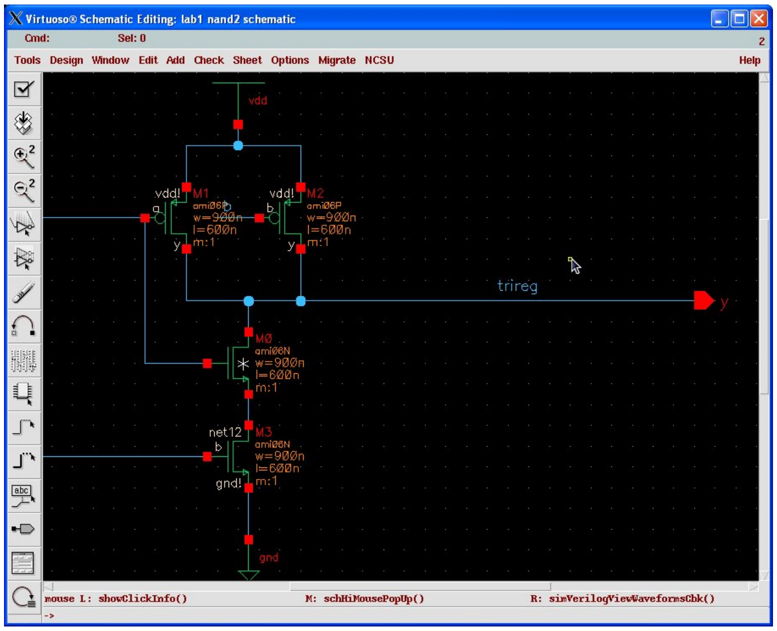

Virtuoso® Schematic Editing: lab1 nand2 schematic
Cmd: Sel: 0
Tools Design Window Edit Add Check Sheet Options Migrate NCSU Help
vdd
vdd! M1 ami06P w=900n l=600n m:1
vdd! M2 ami06P w=900n l=600n m:1
a
b
y
trireg
M0 ami06N w=900n l=600n m:1
net12 M3 ami06N w=900n l=600n m:1
gnd!
gnd
mouse L: showClickInfo() M: schHiMousePopUp() R: simVerilogViewWaveformsCbk()

Figure 4.60: Nand2 Cell with **r_nmos** and **trireg** Modifications

The weak transistors are called **r_nmos** and **r_pmos** (the “r_” means that they are “resistive” switch models). Replacing the topmost nmos device with a weak device doesn’t change the function of the nand2 gate, but results in a cleaner waveform at the output. The schematic is seen in Figure 4.60. Figure 4.61 shows the same portion of the waveform but with both modifications: **trireg** on the output node and a **r_nmos** device at the top of the pulldown stack.

Of course, you may not care about the glitches on the outputs of the nand2 gate. In a synchronous system where the gate outputs are being sent to a storage element like a latch or flip flop, the glitches should die out long

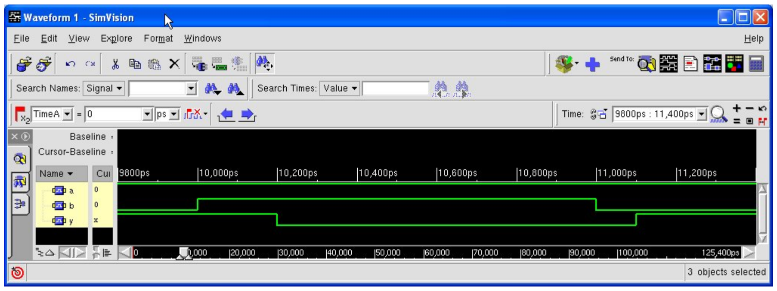


Figure 4.61: Waveform from Simulation of nand2 Cell with Modifications

before they reach the input of the storage element. In that case you may not care if you make these tweaks to your transistor circuits or not. Also, in general, it may not be possible to design a transistor circuit that had no glitches for all possible input changes using the Verilog transistor switch delay model.

I suggest that you at least add **netType trireg** attributes on the outputs of combinational circuits that you design. That's a simple thing to do and helps a lot in terms of confusing outputs.