Chapter 4

Verilog Simulation

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

!(Copyright © 2005, 2010, Cadence Design Systems, Inc. All rights reserved worldwide. Reprinted with permission.)

Figure 4.3: Dialog box for initializing a simulation run directory
Figure 4.4: The initial Verilog-XL simulation control window
Figure 4.5: The Record Signals dialog box

Figure 4.6: Dialog to create a new testfixture template
Figure 4.7: The Verilog-XL Stimulus Options form
// 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

Figure 4.8: Testfixture template for the two-bit adder
// 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
    // Wait for 20 simulation time units
    // Then print out the current 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 testfixture 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
  end
  B[1:0] = B[1:0] + 2'b01; // add the bits
end
A = A+1; // shorthand notation for adding

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

Figure 4.10: Another test fixture for the two-bit adder
// 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 testfixture 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 testfixture using values from external files

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cout</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>01</td>
<td>010</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>011</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

(a) ain.txt (b) bin.txt (c) results.txt

Figure 4.12: Data files used in Figure 4.11
Figure 4.13: Dialog box for re-netlisting a previously netlisted design

Figure 4.14: The netlisting log for the two-bit adder
Figure 4.15: The Verilog-XL window after netlisting
Figure 4.16: Result of running with the testbench from Figure 4.9
Figure 4.17: Waveform window without any signals
Figure 4.18: Design browser with signals selected
Figure 4.19: Waveform window showing exhaustive test results
Figure 4.20: Waveform window showing outputs as a bus
Figure 4.21: Printing dialog box for SimVision
Figure 4.22: Dialog box for initializing a simulation run directory for NC_Verilog
Figure 4.23: The Record Signals dialog box

Figure 4.24: Dialog to create a new testfixture template
Figure 4.25: SimVision Console window
Figure 4.26: SimVision Design Browser window
Figure 4.27: Waveform window showing the output of exhaustive simulation
Figure 4.28: Dialog box for creating a behavioral view

Figure 4.29: Behavioral view template based on the `nand2` symbol
Figure 4.30: Complete behavioral description of a \texttt{nand2} cell

\begin{verbatim}
// 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);
    endif
endmodule
\end{verbatim}

Figure 4.31: Read only window for simulation of \texttt{behavioral} view
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`
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 see4test.v
// 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
downtask // send_test

Figure 4.34: Testbench code for see4.v in a file named testfixture.v
Figure 4.35: Output of stand-alone Verilog-XL simulation of seetest.v
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,

Saw 4 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
Figure 4.37: Control console for stand-alone Verilog-XL simulation using SimVision
Figure 4.38: Hierarchy browser for the see4 example
Figure 4.39: Waveform viewer after running the see4 example
Figure 4.40: Output of stand-alone NC Verilog simulation of see4.v
Figure 4.41: Control console for NC_Verilog through SimVision
--- sim-vcs -f test.files
--- Chronologic VCS (TM)
--- 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
-1c -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
VCS Simulation Report
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
Figure 4.44: Console window for controlling a VCS simulation through DVE
Figure 4.45: The result of running the see4 simulation in DVE
Figure 4.46: Waveform window for DVE after running the see4 simulation
Figure 4.47: A netlisting log for the two-bit adder that stops at behavioral views of the standard cell gates

Figure 4.48: The Setup Netlist dialog from Verilog-XL
Figure 4.49: The Setup Netlist dialog from NC_Verilog
Figure 4.50: Netlisting result after removing behavioral from the verilogSimViewList module

```verilog
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

```verilog
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 a delay parameter

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
Figure 4.55: Rename Reference Library dialog box from Library Manager

Figure 4.56: Netlist log for the nand2 cell
Figure 4.57: Waveform from simulation of nand2 cell

Figure 4.58: Adding a property to a net
Figure 4.59: A net with a netType property
Figure 4.60: Nand2 cell with r_nmos and trireg modifications

Figure 4.61: Waveform from simulation of nand2 cell with modifications