B

Troubleshooting and Problem Solving

Troubleshooting and problem-solving guidelines are described in the following sections:

Reducing Simulation/Synthesis Mismatches

The following sections discuss coding styles that might cause synthesis/simulation mismatches:

- Tasks Inside an always_comb Block
- State Conversion: 2-State and 4-State

Tasks Inside an always_comb Block

The LRM does not define the behavior of tasks inside always_comb blocks, but it does define the behavior for functions. To avoid a simulation/synthesis mismatch, use void functions instead of tasks inside always_comb blocks.

To understand why the mismatch can happen, consider what the IEEE Std 1800-2005 says:

“always_comb is sensitive to changes within the contents of a function, whereas always @ is only sensitive to changes to the arguments of a function.”*

Although the LRM does not say it is illegal to use tasks inside always_comb blocks, it does not specify how always_comb blocks should behave with tasks inside the sensitivity list. This could cause a simulation and synthesis mismatch.

To illustrate this mismatch, consider the following:

- The code in Example B-1, which uses a task inside the always_comb block

- The accompanying testbench (Example B-2), GETCH netlist (Example B-3), and simulation log (Example B-4)

Example B-1 RTL With Task In an always_comb Block

```
module comb1(input logic a, b ,c, output logic [1:0] y);

   always_comb orf1(a);

      function void orf1 (a);
         y[0] = a | b | c;
      endfunction

   always_comb ort1 (a);

      task ort1 (a);
         y[1] = a | b | c;
      endtask

endmodule
```

Example B-2 shows the testbench for Example B-1.

Example B-2 Testbench for Design in Example B-1

```
module comb1_tb(output logic a, b, c);
initial
   begin
 a = 0; b = 0; c = 0;
 #10 a = 0; b = 0; c = 1;
 #10 a = 0; b = 1; c = 0;
 #10 a = 0; b = 1; c = 1;
 #10 a = 1; b = 0; c = 0;
 #10 a = 1; b = 0; c = 1;
 #10 a = 1; b = 1; c = 0;
 #10 a = 1; b = 1; c = 1;
   end
endmodule

module top;
wire a_w, b_w, c_w;
wire y1_w, y0_w ;

comb1 u1(a_w, b_w, c_w, {y1_w, y0_w});
comb1_tb u2(a_w, b_w, c_w);

initial
   begin
 $display("\t\tTime A B C Y1 Y0\n");
 $monitor($time,,,,a_w,,,,b_w,,,,c_w,,,,y1_w,,,,y0_w);
    end
endmodule
```

Example B-3 shows the GTECH netlist for Example B-1.

Example B-3 GTECH Netlist

```
module comb1 ( a, b, c, y );
output [1:0] y;
input  a, b, c;
    wire N0, N1;
    GTECH_OR2 C6 ( .A(N0), .B(c), .Z(y[0]) );
    GTECH_OR2 C7 ( .A(a), .B(b), .Z(N0) );
    GTECH_OR2 C8 ( .A(N1), .B(c), .Z(y[1]) );
    GTECH_OR2 C9 ( .A(a), .B(b), .Z(N1) );
endmodule
```

Example B-4 shows the simulation log for Example B-1.

Example B-4 Simulation Log

```
SIMULATION RESULTS USING SUPERLOG:

453 lamb > sv comb1.sv comb1_tb.sv
Info:   Start of SYSTEMSIM version 20021030.00 v2.2.1
Info:   Incorporating SUPERLOG(tm) and CBlend(tm) technology.
Info:   Copyright (C) 1998-2002 Synopsys, Inc. All rights reserved.
Info:   Unpublished -- rights reserved under the copyright laws of the United
States.
Info:   Oct 14, 2003 16:43:54 Reading the design...
Info:   opening file comb1.sv
Info:   opening file comb1_tb.sv
Info:   Checking identifiers...
Info:   Highest level modules:
Info:       top
Info:   Checking types...
Info:   Oct 14, 2003 16:43:54 Elaborating the hierarchy...
Info:   Oct 14, 2003 16:43:54 Starting simulation...
                 Time  A   B   C   Y1 Y0

                   0   0   0   0   0   0
                  10   0   0   1   0   1
                  20   0   1   0   0   1
                  30   0   1   1   0   1
                  40   1   0   0   1   1
                  50   1   0   1   1   1
                  60   1   1   0   1   1
                  70   1   1   1   1   1
Info:   Oct 14, 2003 16:43:54 End of simulation at 70 (event queue empty).
Info:   Pre-Simulation Time   User: 0.04 seconds  System: 0.08 seconds
Info:       Simulation Time   User: 0.00 seconds  System: 0.00 seconds
Info:   End of SYSTEMSIM version 20021030.00 v2.2.1
```

In synthesis, the tool produces four 2-input OR gates, as shown in Example B-3. The outputs y[0] and y[1] are both outputs of 2-input OR gates. The simulation log (Example B-4), shows

- y[0] going to 1 when any of the inputs is 1, which is correct. Note that y[0] is the output from the void function call orf1.

- y[1] going to 1 only when input A goes to 1, and it is not sensitive to the changes of B and C. This is incorrect behavior that occurs because y[1] is the output of task ort1 inside an always_comb block.

This situation results in a simulation/synthesis mismatch, and the synthesis tool issues the following VER-520 warning to indicate that a task is used inside an always_comb block:

```
Compiling source file /remote/cae726/sv/doc/
sim_synth_mismatch/comb1.sv
Warning:  /remote/cae726/sv/doc/sim_synth_mismatch/
comb1.sv:10: Task enable in always_comb block. (VER-520)
```

To avoid this simulation/synthesis mismatch, use void functions inside always_comb blocks, as shown in Example B-5, and do not use tasks.

Example B-5   Recommended Coding Style

```
module comb1(input logic a, b ,c, output logic [1:0] y);

   always_comb orf1(a);

      function void orf1 (a);
         y[0] = a | b | c;
         y[1] = a | b | c;
      endfunction

endmodule
```

State Conversion: 2-State and 4-State

The tool treats 2-state variables the same as 4-state variables. (See Appendix D, “Unsupported Constructs.”) Therefore a simulation/synthesis mismatch can occur when you convert from 4-state to 2-state or from 2-state to 4-state. The simulation tool considers an “x” value as a “don’t know” value, whereas the synthesis tool considers an “x” value as a “don’t care” value.

In Example B-6, “a” is an input of logic type (4-state) making a continuous assignment to “b” which is bit type (2-state). The testbench drives “a” through the variable “a_driver” which is also logic type but is uninitialized at time 0. Therefore for simulation, this is a “don’t know” situation, as shown in the simulation log in Example B-7.

Because we have the statement, assign b = a, and b is bit type, its default value is 0 (if uninitialized), and therefore the simulation log has A = x and B = 0 at time 0.

The tool does not issue warnings for these situations. To avoid this simulation/synthesis mismatch, use only 2-state or only 4-state variables and avoid such conversions.

Example B-6 RTL

```
module logic_bit_test(input logic a, output bit b);
   assign b = a;
endmodule

module logic_bit_testbench(output logic a_driver);
   initial begin // no intial value
      #10 a_driver = ’1;
      #10 a_driver = ’0;
      #10 $finish;
   end
endmodule

module top;
   wire a_con, b_con;

   logic_bit_test u1(a_con, b_con);
   logic_bit_testbench u2(a_con);

   initial
     begin
      $display("\t\tTime A  B\n");
      $monitor($time,,,,a_con,,,,b_con);
    end
endmodule
```

Example B-7 Simulation Log

```
SIMULATION RESULTS USING SUPERLOG ---


654 lamb >  sv  logic_bit_test.sv
Info:   Start of SYSTEMSIM version 20021030.00 v2.2.1
Info:   Incorporating SUPERLOG(tm) and CBlend(tm) technology.
Info:   Copyright (C) 1998-2002 Synopsys, Inc. All rights reserved.
Info:   Unpublished -- rights reserved under the copyright laws of the United
States.
Info:   Oct 27, 2003 09:25:45 Reading the design...
Info:   opening file logic_bit_test.sv
Info:   Checking identifiers...
Info:   Highest level modules:
Info:       top
Info:   Checking types...
Info:   Oct 27, 2003 09:25:45 Elaborating the hierarchy...
Info:   Oct 27, 2003 09:25:45 Starting simulation...
                Time A  B

                   0   x   0
                  10   1   1
                  20   0   0
Info:   logic_bit_test.sv:9 $finish encountered
Info:   Oct 27, 2003 09:25:45 End of simulation at 30 (explicit finish).
Info:   Pre-Simulation Time   User: 0.03 seconds  System: 0.03 seconds
Info:       Simulation Time   User: 0.00 seconds  System: 0.00 seconds
Info:   End of SYSTEMSIM version 20021030.00 v2.2.1
```

Detecting Unintended Hardware and Empty Blocks

When you use the always_ff, always_latch, and always_comb constructs, the tool expects certain hardware. If the expected hardware is not inferred, or if the block might be removed during compile, the tool generates a warning message. For details, see

- “Using always_comb and Inferring a Register” on page 3-3
- “Using always_comb with Empty Blocks” on page 3-5

- “Using always_latch and Not Inferring a Sequential Device” on page 4-4
- “Using always_latch With Empty Blocks” on page 4-5
- “Using always_ff and Not Inferring a Sequential Device” on page 4-7
- “Using always_ff With Empty Blocks” on page 4-8

Port Connection (.*) Reading Restriction

When using the .* port connection style, you must analyze all the lower-level modules before elaborating the top-level module; otherwise the tool gives an ELAB-397 error message.

To understand this, consider the following two files:

```
// top.sv has the following code:

module top(input logic in, output logic out);
bottom b1(.*);
endmodule

// bottom.sv has the following code:

module bottom(input logic in, output logic out);
assign out = in;
endmodule
```

If you analyze the top-level module and elaborate the top design without analyzing the bottom, as in the following script:

```
analyze -f sverilog top.sv
elaborate top
compile
write -f verilog -h -o gates.sverilog.v
quit
```

you will see the following error:

```
Error:  ./top.sv:2: The module or interface ‘bottom’ needs to be analyzed.
(ELAB-397)
```

To prevent this error, you can do these workarounds:

1. Modify the script to analyze the bottom module before you elaborate the top.

```
analyze -f sverilog {bottom.sv top.sv}
elaborate top
compile
write -f verilog -h -o gates.sverilog.v
quit
```

2. Provide a placeholder bottom module in the same file as top.sv, such as

```
module bottom(input logic in, output logic out);
// this is a placeholder module that contains port names but no content
endmodule
```

Using do...while Loops

A do....while loop is synthesizable if the exit condition is deterministic. The tool does not handle unknown initial values in loops when the number of iterations can still be bounded. However, the simulator tool, VCS, does not have this restriction. For example, Example B-8 is synthesizable; Example B-9 is not. In Example B-9, the exit condition "x" is an unknown initial value; therefore the tool reports an ELAB-900 error.

Example B-8

```
module do_while_test2(input logic [3:0] count1, output logic [3:0] z);

      reg [3:0] x, count;

           always_comb
                 begin
                      x = 4’d2;
                     count = count1;

                     do
                     begin
                          count++;
                          x++;
                     end
                     while(x < 4’d15);

                       z = count;

             end

endmodule
```

Example B-9

```
module do_while_test2(input logic [3:0] count1, count, x, output logic[3:0] z);
       always_comb
            begin
            count = count1;
            do
               begin
                    count++;
                    x++;
               end
              while(x < 4’d15);

              z = count;

            end
endmodule
```

Using the typedef Construct

You need to define typedef before using it, as shown in Example B-10.

Example B-10

```
For example, allowed.sv is allowed; not_allowed.sv is not allowed.

   allowed.sv
   ==========

   typedef logic mytype;

   module allowed(input logic clock, input mytype in, output mytype out);
        always_ff@(posedge clock)
             out = in;
 endmodule


 not_allowed.sv
 ==============

 module not_allowed(input logic clock, input mytype in, output mytype out);

        always_ff@(posedge clock)
             out <= in;
 endmodule

 typedef logic mytype;
```

Reading Assertions in Synthesis

The following SystemVerilog keywords are parsed and ignored: assert, assume (VCS, the simulation tool, may not support this keyword at this time), before, bind, bins, binsof, class, clocking, constraint, cover, coverpoint, covergroup, cross, endclass, endclocking, endgroup, endpackage, endprogram, endproperty, endsequence, extends, final, first_match, intersect, ignore_bins, illegal_bins, local, package, program, property, protected, sequence, super, this, var, throughout, within. If an assertion-related keyword is not parsed and ignored, it is considered to be unsupported. For these unsupported keywords, see “Unsupported Constructs” in Appendix D.

Example B-11 shows how the synthesis tool parses and ignores the “assert” keyword. The example correctly infers a flip-flop, as shown in the inference report in Example B-12.

Example B-11

```
module dff_with_imm_assert(input DATA, CLK, RESET, output logic Q);
  //synopsys sync_set_reset "RESET"
   always_ff @(posedge CLK)
      if (~RESET)
           begin
            Q <= 1’b0;
                assert (Q == 1’b0)
                $display("%m PASS:Flip Flop got reset");
                else
                $display("%m FAIL:Flip Flop got reset");
           end
      else
            Q <= DATA;

endmodule
```

Example B-12 Inference Report

```
=============================================================================
|    Register Name    |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=============================================================================
|       Q_reg         | Flip-flop |   1    |  N  | N  | N  | N  | Y  | N  | N  |
=============================================================================
Presto compilation completed successfully.
```

Other Troubleshooting Guidelines

- For unsupported SystemVerilog constructs, see “Unsupported Constructs” in Appendix D.
- If you are having problems with $unit, make sure you are following the coding guidelines in Chapter 2, “Global Name Space ($unit).
- If you are having problems with interfaces, make sure you are following the coding guidelines in Chapter 6, “Interfaces.
- Regarding casting, the use of nonvoid function calls as statements is supported but generates a warning.
- In some cases, the tool does not correctly rename the output. See “Renaming Example 3” on page 6-36.
- If your design contains interfaces, you cannot use the elaborate command to instantiate a parameterized design. See “Reading SystemVerilog Files” on page 1-7.
- If your design uses checker libraries, see “Reading Designs With Assertion Checker Libraries” on page 1-10.
- Generally, all the restrictions for HDL Compiler (Presto Verilog) apply to the SystemVerilog tool. For details, see the *HDL Compiler (Presto Verilog) Reference Manual*.