Verilog Review

Lecture 03

Josh Brake
Harvey Mudd College

Outline

- Verilog tips
- Design guidelines and basic idioms
- Bad Verilog Examples/Bug finding

Learning Objectives

By the end of this lecture you should be able to…

- Recall basic Verilog idioms for common digital structures
- Analyze Verilog code to find errors

Verilog Tips

- Think about hardware you want, write code that implies it. Use the idioms and never think about this as coding.
- Watch for warnings in tool. Take them seriously – often a sign of a big problem.
- Block diagrams: draw box with inputs, outputs. Divide into simpler boxes until you can understand it.

Design Guidelines and Idioms for Common Structures

- Simple combinational logic: use assign statements
- For a mux: assign ? :
- Truth tables: always_comb and case statement (default case of x)
- Finite state machines. Three portions: state register, next state logic, output logic.
- Use always_ff, always_comb, not plain always blocks.
- Use logic datatype everywhere except on tristates, and don’t use tristates

System Verilog Operators

Listed in order of descending precedence.

<table>
<tr><th>Verilog Operator</th><th>Name</th><th>Functional Group</th></tr>
<tr><td>[ ]</td><td>bit-select or part-select</td><td></td></tr>
<tr><td>( )</td><td>parenthesis</td><td></td></tr>
<tr><td>!
~
&
|
~&
~|
^
~^ or ^~</td><td>logical negation
negation
reduction AND
reduction OR
reduction NAND
reduction NOR
reduction XOR
reduction XNOR</td><td>logical
bit-wise
reduction
reduction
reduction
reduction
reduction
reduction</td></tr>
<tr><td>+
-</td><td>unary (sign) plus
unary (sign) minus</td><td>arithmetic
arithmetic</td></tr>
<tr><td>{ }</td><td>concatenation</td><td>concatenation</td></tr>
<tr><td>{{ }}</td><td>replication</td><td>replication</td></tr>
<tr><td>*
/
%</td><td>multiply
divide
modulus</td><td>arithmetic
arithmetic
arithmetic</td></tr>
<tr><td>+
-</td><td>binary plus
binary minus</td><td>arithmetic
arithmetic</td></tr>
<tr><td><<
>></td><td>shift left
shift right</td><td>shift
shift</td></tr>
<tr><td>>
>=
<
<=</td><td>greater than
greater than or equal to
less than
less than or equal to</td><td>relational
relational
relational
relational</td></tr>
<tr><td>==
!=</td><td>logical equality
logical inequality</td><td>equality
equality</td></tr>
<tr><td>===
!==</td><td>case equality
case inequality</td><td>equality
equality</td></tr>
<tr><td>&</td><td>bit-wise AND</td><td>bit-wise</td></tr>
<tr><td>^
^~ or ~^</td><td>bit-wise XOR
bit-wise XNOR</td><td>bit-wise
bit-wise</td></tr>
<tr><td>|</td><td>bit-wise OR</td><td>bit-wise</td></tr>
<tr><td>&&</td><td>logical AND</td><td>logical</td></tr>
<tr><td>||</td><td>logical OR</td><td>logical</td></tr>
<tr><td>?:</td><td>conditional</td><td>conditional</td></tr>
</table>

Harris and Harris, *Digital Design and Computer Architecture*, ARM Ed., p. 185

Bad Verilog: Learning by Counter Example

Bad Verilog Example #1

```
module mux(input  logic [3:0] d0, d1,
           input  logic       s,
           output logic [3:0] y);
   always_comb @(posedge s)
      if (s) y <= d1;
      else   y <= d0;
endmodule
```

Problem: `always_comb` cannot have a sensitivity list.

Bad Verilog Example #2

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output       [3:0] y);

   tristate t0(d0, s, y);
   tristate t1(d1, s, y);
endmodule
```

Problem: No type defined for the output `y`. First tristate should be activated for `~s`.

Bad Verilog Example #3

```
module mux3(input  logic [3:0] d0, d1, d2,
            input  logic [1:0] s,
            output logic [3:0] y);

  always_comb
    if (s == 2'b00) y <= d0;
    else if (s == 2'b01) y <= d1;
    else if (s == 2'b10) y <= d2;
endmodule
```

Problem: No default case. This will imply a latch and thus throw an error (won't compile). What would happen if you used always instead of `always_comb`?

Bad Verilog Example #4

```
module mux8(input  logic [3:0] d0, d1, d2, d3, d4, d5, d6, d7,
            input  logic [2:0] s,
            output logic [3:0] y);

  always_comb
    case (s)
      1'd0: y <= d0;
      1'd1: y <= d1;
      1'd2: y <= d2;
      1'd3: y <= d3;
      1'd4: y <= d4;
      1'd5: y <= d5;
      1'd6: y <= d6;
      1'd7: y <= d7;
      default: y <= 4'bxxxx;
    endcase
endmodule
```

Problem: Inputs for case statement should be 3 bits instead of 1.

Bad Verilog Example #5

```
module and3(input  logic a, b, c,
            output logic y);

   logic tmp;

   always @(a, b, c)
   begin
      tmp <= a & b;
      y <= tmp & c;
   end
endmodule
```

Problem: Missing `tmp` from sensitivity list. Will synthesize properly, but simulate incorrectly.

Bad Verilog Example #6

```
module counter(input  logic        clk,
               output logic [31:0] q);

  always_ff @(posedge clk)
    q <= q+1;
endmodule
```

Problem: No reset, starts at random value.

Bad Verilog Example #7

```
module counter2(input logic clk,
                output logic [31:0] q);
  initial q <= 32'b0;

  always_ff @(posedge clk) q <= q+1;
endmodule
```

Problem: initial causes this to work fine in simulation, but starting value is unpredictable in hardware. Can fix with reset.

Bad Verilog Example #8

```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);

   always @(a)
      begin
         y1 <= a & b; // AND
         y2 <= a | b; // OR
         y3 <= a ^ b; // XOR
         y4 <= ~(a & b); // NAND
         y5 <= ~(a | b); // NOR
      end
endmodule
```

Problem: Missing b from sensitivity list. Should have used an always_comb statement.

Bad Verilog Example #9

```
module priority_always(input  logic [3:0] a,
                       output logic [3:0] y);

   // a 4-input priority encoder
   always_comb
      if      (a[3]) y <= 4'b1000;
      else if (a[2]) y <= 4'b0100;
      else if (a[1]) y <= 4'b0010;
      else if (a[0]) y <= 4'b0001;
endmodule
```

Problem: Missing default statement; will infer a latch.

Bad Verilog Example #10

```
module seven_seg_display_decoder(input  logic [3:0] data,
                                 output logic [6:0] segments);
   always_comb
      case (data)
         0: segments <= 7'b000_0000; // ZERO
         1: segments <= 7'b111_1110; // ONE
         2: segments <= 7'b011_0000; // TWO
         3: segments <= 7'b110_1101; // THREE
         4: segments <= 7'b011_0011; // FOUR
         5: segments <= 7'b101_1011; // FIVE
         6: segments <= 7'b101_1111; // SIX
         7: segments <= 7'b111_0000; // SEVEN
         8: segments <= 7'b111_1111; // EIGHT
         9: segments <= 7'b111_1011; // NINE
      endcase
endmodule
```

Problem: No default statement; will infer a latch.

Bad Verilog Example #11

```
module latch(input  logic       clk,
             input  logic [3:0] d,
             output logic [3:0] q);

   always_latch @(clk)
      if (clk) q <= d;
endmodule
```

Problem: Missing d from sensitivity list.

Bad Verilog Example #12

```
module floprsen(input  logic       clk,
                input  logic       reset,
                input  logic       set,
                input  logic [3:0] d,
                output logic [3:0] q);

   always_ff @(posedge clk, posedge reset)
      if (reset) q <= 0;
      else       q <= d;

   always @(set)
      if (set) q <= 1;
endmodule
```

Problem: Set is in the wrong block. Should put it in the same block as the other signals.

Bad Verilog Example #13

```
module twobitflop(input  logic       clk,
                  input  logic [1:0] d,
                  output logic [1:0] q);

   always_ff @(posedge clk)
      q[1] = d[1];
      q[0] = d[0];
endmodule
```

Missing begin and end statement.

Bad Verilog Example #14

```
module FSMbad(input  logic clk,
              input  logic a,
              output logic out1, out2);

   logic  state;

   always_ff @(posedge clk)
      if (state == 0) begin
         if (a) state <= 1;
      end else begin
         if (~a) state <= 0;
      end

   always_comb
      if (state == 0) out1 <= 1;
      else            out2 <= 1;
endmodule
```

Output logic is missing logic to set the other outputs to 0 in each state.

Bad Verilog Example #15

```
module divideby3counter(input  logic       clk, reset,
                        output logic [1:0] q);

  always_ff @(posedge clk or posedge reset)
    if (reset) q = 0;
    else begin
      q = q+1;
      if (q == 2) q = 0;
    end
endmodule
```

Problem: Divides by two because it uses blocking assignments. This would work if you used non-blocking assignments.

Bad Verilog Example #16

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic out);

   logic  [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;

   // State Register
   always_ff @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;
// Next State Logic
   always_comb
      case (state)
         S0: nextstate <= S1;
         S1: nextstate <= S2;
         S2: nextstate <= S0;
      endcase

   // Output Logic
   assign out = (state == S2);
endmodule
```

Problem: Missing default from nextstate logic; infers a latch.

Bad Verilog Example #17

```
module divideby3FSM2(input  logic clk,
                     output logic out);

   logic  [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;

   initial state = 2'b00;

   // State Register
   always_ff @(posedge clk)
      if      (state == 2'b00) state <= 2'b01;
      else if (state == 2'b01) state <= 2'b10;
      else if (state == 2'b10) state <= 2'b00;

   // Output Logic
   assign out = (state == S2);
endmodule
```

Works in sim, could get stuck in state 2’b11 in hardware. Can fix with a default or a reset.

Bad Verilog Example #18

```
module adventuregameFSM(input              clk, reset, N, S, E, W,
                        output logic [3:0] room,
                        output logic       win, die);

  always_ff @(posedge clk or posedge reset)
    if (reset) begin
      room <= 4'b0001;
      die <= 0;
      win <= 0;
    end

    else case (room)
      4'b0001: if (E) room <= 4'b0010;
            else die <= 1;
      4'b0010: if (S) room <= 4'b0100;
            if (W) room <= 4'b0001;
            else die <= 1;
      4'b0100: if (E) room <= 4'b1000;
            if (N) room <= 4'b0010;
            else die <= 1;
    endcase
always_comb
    if (room == 4'b1000) win <= 1;
endmodule
```

Win is assigned in two different always blocks. Missing elseifs in two blocks. Missing default cases.

Wrap Up

- Think about hardware you want. Then write the HDL to imply the proper logic.
- Check the Netlist Analyzer to ensure that the tool is inferring the logic you are intending.
- Make sure to not infer latches.

Announcements and Reminders