Digital Design with SystemVerilog

Prof. Stephen A. Edwards

Columbia University

Spring 2014
Synchronous Digital Design

Combinational Logic

Sequential Logic

Summary of Modeling Styles

Example: Bresenham’s Line Algorithm

Testbenches
Why HDLs?

1970s: SPICE transistor-level netlists

An XOR built from four NAND gates

```
.MOSTEL P PMOS
.MMODEL N NMOS

.SUBCKT NAND A B Y Vdd Vss
M1 Y A Vdd Vdd P
M2 Y B Vdd Vdd P
M3 Y A X Vss N
M4 X B Vss Vss N
.ENDS

X1 A B I1 Vdd 0 NAND
X2 A I1 I2 Vdd 0 NAND
X3 B I1 I3 Vdd 0 NAND
X4 I2 I3 Y Vdd 0 NAND
```
Why HDLs?

1980s: Graphical schematic capture programs
Why HDLs?

1990s: HDLs and Logic Synthesis

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ALU is
port(A: in unsigned(1 downto 0);
    B: in unsigned(1 downto 0);
    Sel: in unsigned(1 downto 0);
    Res: out unsigned(1 downto 0));
end ALU;
architecture behv of ALU is begin
    process (A,B,Sel) begin
        case Sel is
            when "00" => Res <= A + B;
            when "01" => Res <= A + (not B) + 1;
            when "10" => Res <= A and B;
            when "11" => Res <= A or B;
            when others => Res <= "XX";
        end case;
    end process;
end behv;
```
Separate but Equal: Verilog and VHDL

Verilog: More succinct, really messy

VHDL: Verbose, overly flexible, fairly messy

Part of languages people actually use identical

Every synthesis system supports both

SystemVerilog a newer version. Supports many more features.
Synchronous Digital Design
The Synchronous Digital Logic Paradigm

- Gates and D flip-flops only
- No level-sensitive latches
- All flip-flops driven by the same clock
- No other clock signals
- Every cyclic path contains at least one flip-flop
- No combinational loops
Timing in Synchronous Circuits

$t_c$: Clock period. E.g., 10 ns for a 100 MHz clock
Timing in Synchronous Circuits

Sufficient Hold Time?

\[ t_{p\text{(min,FF)}} \quad \text{and} \quad t_{p\text{(min,CL)}} \]

CLK

Q

D

Hold time constraint: how soon after the clock edge can D start changing? Min. FF delay + min. logic delay
Timing in Synchronous Circuits

Setup time constraint: when before the clock edge is $D$ guaranteed stable? Max. FF delay + max. logic delay
Combinational Logic
Systems are built from modules

"Continuous assignment" expresses combinational logic

// Full adder
module full_adder(input logic a, b, c,  
output logic sum, carry);

assign sum = a ^ b ^ c;
assign carry = a & b | a & c | b & c;
endmodule
module gates(input logic [3:0] a, b, output logic [3:0] y1, y2, y3, y4, y5);

/* Five groups of two-input logic gates acting on 4-bit busses */
assign y1 = a & b;    // AND
assign y2 = a | b;    // OR
assign y3 = a ^ b;    // XOR
assign y4 = ~(a & b); // NAND
assign y5 = ~(a | b); // NOR
endmodule
module and8(input logic [7:0] a, 
            output logic y);

    assign y = &a;  // Reduction AND

    // Equivalent to

    // Also ~|a   NAND
    // |a       OR
    // ~|a     NOR
    // ^a      XOR
    // ^a      XNOR

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

    // Array of two-input muxes
    assign y = s ? d1 : d0;
endmodule
## Operators in Precedence Order

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!c</td>
<td>NOT, Negate, Reduction</td>
</tr>
<tr>
<td>-c</td>
<td>AND, NAND</td>
</tr>
<tr>
<td>&amp;c</td>
<td>OR, NOR, XOR, XNOR</td>
</tr>
<tr>
<td>~&amp;c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c</td>
</tr>
<tr>
<td>~</td>
<td>c</td>
</tr>
<tr>
<td>^c</td>
<td></td>
</tr>
<tr>
<td>~^c</td>
<td></td>
</tr>
<tr>
<td>a*b</td>
<td>Multiply, Divide, Modulus</td>
</tr>
<tr>
<td>a/b</td>
<td>Add, Subtract</td>
</tr>
<tr>
<td>a%b</td>
<td></td>
</tr>
<tr>
<td>a+b</td>
<td></td>
</tr>
<tr>
<td>a−b</td>
<td></td>
</tr>
<tr>
<td>a&lt;&lt;b</td>
<td>Logical Shift</td>
</tr>
<tr>
<td>a&gt;&gt;b</td>
<td>Arithmetic Shift</td>
</tr>
<tr>
<td>a&lt;&lt;&lt;b</td>
<td></td>
</tr>
<tr>
<td>a&gt;&gt;&gt;b</td>
<td></td>
</tr>
<tr>
<td>a&lt;b</td>
<td>Relational</td>
</tr>
<tr>
<td>a&lt;=b</td>
<td></td>
</tr>
<tr>
<td>a&gt;b</td>
<td></td>
</tr>
<tr>
<td>a&gt;=b</td>
<td></td>
</tr>
<tr>
<td>a==b</td>
<td>Equality</td>
</tr>
<tr>
<td>a!=b</td>
<td></td>
</tr>
<tr>
<td>a&amp;b</td>
<td>AND</td>
</tr>
<tr>
<td>a^&amp;b</td>
<td></td>
</tr>
<tr>
<td>a^b</td>
<td>XOR, XNOR</td>
</tr>
<tr>
<td>a~^b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a?b:c</td>
<td>Conditional</td>
</tr>
<tr>
<td>{a,b,c,d}</td>
<td>Concatenation</td>
</tr>
</tbody>
</table>
An XOR Built Hierarchically

```verbatim
module mynand2(input logic a, b,
             output logic y);
    assign y = ~(a & b);
endmodule

module myxor2(input logic a, b,
             output logic y);
    logic abn, aa, bb;

    mynand2 n1(a, b, abn),
             n2(a, abn, aa),
             n3(abn, b, bb),
             n4(aa, bb, y);
endmodule
```

 Declare internal wires
 n1: A mynand2 connected to a, b, and abn

[Diagram showing hierarchy of modules and connections]

A Decimal-to-Seven-Segment Decoder

always_comb: combinational logic in an imperative style

module dec7seg(input logic [3:0] a, output logic [6:0] y);

always_comb:

case (a)
  4’d0: y = 7’b111_1110;
  4’d1: y = 7’b011_0000;
  4’d2: y = 7’b110_1101;
  4’d3: y = 7’b111_1001;
  4’d4: y = 7’b011_0011;
  4’d5: y = 7’b101_1011;
  4’d6: y = 7’b101_1111;
  4’d7: y = 7’b111_0000;
  4’d8: y = 7’b111_1111;
  4’d9: y = 7’b111_0011;

default: y = 7’b000_0000;
endcase
endmodule

Multiway conditional

4’d5: decimal “5” as a four-bit binary number

seven-bit binary vector (_ is ignored)

Mandatory

“blocking assignment”: use in always_comb
Verilog Numbers

16’h8_0F

Number of Bits
Base: b, o, d, or h

Value:
_’s are ignored
Zero-padded

4’b1010 = 4’o12 = 4’d10 = 4’ha
16’h4840 = 16’b 100_1000_0100_0000
module comb1(
  input logic [3:0] a, b,
  input logic s,
  output logic [3:0] y);

  always_comb
    if (s)
      y = a + b;
    else
      y = a & b;

endmodule

Both a + b and a & b computed, mux selects the result.
Imperative Combinational Logic

```verilog
defmodule comb2(
    input logic [3:0] a, b,
    input logic s, t,
    output logic [3:0] y);

    always_comb
    if (s)
        y = a + b;
    else if (t)
        y = a & b;
    else
        y = a | b;

endmodule
```

All three expressions computed in parallel. Cascaded muxes implement priority (s over t).

<table>
<thead>
<tr>
<th>s</th>
<th>t</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>–</td>
<td>a + b</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>a &amp; b</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>a</td>
</tr>
</tbody>
</table>
module comb3(
  input logic [3:0] a, b,
  input logic s, t,
  output logic [3:0] y, z);

  always_comb begin
    z = 4'b0;
    if (s) begin
      y = a + b;
      z = a - b;
    end else if (t) begin
      y = a & b;
      z = a + b;
    end else
      y = a | b;
  end
endmodule

Separate mux cascades for y and z.
One copy of a + b.
An Address Decoder

```
module adecode(
    input logic [15:0] address,
    output logic RAM, ROM,
    output logic VIDEO, IO);

always_comb begin
    {RAM, ROM, VIDEO, IO} = 4'b 0;
    if (address[15])
        RAM = 1;
    else if (address[14:13] == 2'b 00)
        VIDEO = 1;
    else if (address[14:12] == 3'b 101)
        IO = 1;
    else if (address[14:13] == 2'b 11)
        ROM = 1;
end

endmodule
```

Omitting defaults for RAM, etc. will give "construct does not infer purely combinational logic."
Sequential Logic
A D-Flip-Flop

```verilog
module mydff(input logic clk,
             input logic d,
             output logic q);

always_ff @ (posedge clk)
    q <= d;

endmodule
```

- **always_ff** introduces sequential logic.
- **posedge** clk is triggered by the rising edge of clk.
- Copy d to q.
- Non-blocking assignment: happens "just after" the rising edge.

**Diagram:**
- D-Flip-Flop module with inputs d and clk, and output q.
- The diagram represents the circuit symbol for a D-Flip-Flop with connections to the inputs and output.
module count4(input logic clk, 
    output logic [3:0] count);

always_ff @(posedge clk)
    count <= count + 4'd 1;
endmodule
module dec_counter(input logic clk,
                   input logic reset, hold, load,
                   input logic [3:0] d,
                   output logic [3:0] count);

always_ff @(posedge clk)
  if (reset) count <= 4’d 0;
  else if (load) count <= d;
  else if (~hold)
    if (count == 4’d 9) count <= 4’d 0;
    else count <= count + 4’d 1;

endmodule
Moore and Mealy Finite-State Machines

The Moore Form:

Outputs are a function of only the current state.
Moore and Mealy Finite-State Machines

The Mealy Form:

Outputs may be a function of both the current state and the inputs.

A mnemonic: Moore machines often have more states.
module moore_tlc(input logic clk, reset,
    input logic advance,
    output logic red, yellow, green);

enum logic [2:0] {R, Y, G} state; // Symbolic state names

always_ff @(posedge clk) // Moore-style next-state logic
    if (reset) state <= R;
    else case (state)
        R: if (advance) state <= G;
        G: if (advance) state <= Y;
        Y: if (advance) state <= R;
        default: state <= R;
    endcase

assign red  = state == R; // Combinational output logic
assign yellow = state == Y; // separated from next-state logic
assign green = state == G;

endmodule
module mealy_tlc(input logic clk, reset,
    input logic advance,
    output logic red, yellow, green);

typedef enum logic [2:0] {R, Y, G} state_t;
state_t state, next_state;

always_ff @(posedge clk) state <= next_state;

always_comb begin // Mealy-style next state and output logic
    {red, yellow, green} = 3'b0; // Default: all off and
    next_state = state; // hold state
    if (reset) next_state = R;
    else case (state)
        R: begin red = 1;
            if (advance) next_state = G; end
        G: begin green = 1;
            if (advance) next_state = Y; end
        Y: begin yellow = 1;
            if (advance) next_state = R; end
        default: if (advance) next_state = R;
    endcase
end
endmodule
### Blocking vs. Nonblocking assignment

**module** nonblock(input clk, 
    input logic a, 
    output logic d);

logic b, c;

always_ff @(posedge clk)
    begin
        b <= a; \(\text{Nonblocking assignment:}\)
        c <= b; \(\text{All run on the}\)
        d <= c; \(\text{clock edge}\)
    end

**endmodule**

**module** blocking(input clk, 
    input logic a, 
    output logic d);

logic b, c;

always_ff @(posedge clk)
    begin
        b = a; \(\text{Blocking assignment:}\)
        c = b; \(\text{Effect felt by}\)
        d = c; \(\text{next statement}\)
    end

**endmodule**
Summary of Modeling Styles
module styles_tlc(input logic clk, reset,
                 input logic advance,
                 output logic red, yellow, green);
enum logic [2:0] {R, Y, G} state;

always_ff @(posedge clk) // Imperative sequential
  if (reset) state <= R; // Non-blocking assignment
  else case (state) // Case
    R: if (advance) state <= G; // If-else
    G: if (advance) state <= Y;
    Y: if (advance) state <= R;
    default: state <= R;
  endcase

always_comb begin // Imperative combinational
  {red, yellow} = 2'b 0; // Blocking assignment
  if (state == R) red = 1; // If-else
  case (state) // Case
    Y: yellow = 1;
  default: ;
  endcase;
end

assign green = state == G; // Cont. assign. (comb)
endmodule
Example: Bresenham’s Line Algorithm
Bresenham’s Line Algorithm

Objective: Draw a line...
Bresenham’s Line Algorithm

...with well-approximating pixels...
Bresenham’s Line Algorithm

...by maintaining error information.

Error = \(-\frac{3}{7}\)

Error = \frac{1}{7}
Bresenham’s Line Algorithm

...encoded using integers

Error = -3/7

Error = 1/7
Approach

1. Understand the algorithm
   I went to Wikipedia; doesn’t everybody?

2. Code and test the algorithm in software
   I used C and the SDL library for graphics

3. Define the interface for the hardware module
   A communication protocol: consider the whole system

4. Schedule the operations
   Draw a timing diagram! In hardware, you must know in which cycle each thing happens.

5. Code in RTL
   Always envision the hardware you are asking for

6. Test in simulation
   Create a testbench: code that mimicks the environment (e.g., generates clocks, inputs).

7. Test on the FPGA
   Simulating correctly is necessary but not sufficient.
The Pseudocode from Wikipedia

function line(x0, y0, x1, y1)
  dx := abs(x1-x0)
  dy := abs(y1-y0)
  if x0 < x1 then sx := 1 else sx := -1
  if y0 < y1 then sy := 1 else sy := -1
  err := dx-dy

loop
  setPixel(x0,y0)
  if x0 = x1 and y0 = y1 exit loop
  e2 := 2*err
  if e2 > -dy then
    err := err - dy
    x0 := x0 + sx
  end if
  if e2 < dx then
    err := err + dx
    y0 := y0 + sy
  end if
end loop
void line(Uint16 x0, Uint16 y0, Uint16 x1, Uint16 y1) {
    Sint16 dx, dy; // Width and height of bounding box
    Uint16 x, y;   // Current point
    Sint16 err;    // Loop-carried value
    Sint16 e2;     // Temporary variable
    int right, down; // Boolean

    dx = x1 - x0; right = dx > 0; if (!right) dx = -dx;
    dy = y1 - y0; down = dy > 0; if (down) dy = -dy;
    err = dx + dy; x = x0; y = y0;
    for (;;) {
        plot(x, y);
        if (x == x1 && y == y1) break; // Reached the end
        e2 = err << 1; // err * 2
        if (e2 > dy) { err += dy; if (right) x++; else x--;
        }
        if (e2 < dx) { err += dx; if (down) y++; else y--;
        }
    }
}
Module Interface

```verilog
module bresenham(input logicclk, reset,
                 input logictstart,
                 input logic [10:0] x0, y0, x1, y1,
                 output logictplot,
                 output logic [10:0] tx, ty,
                 output logictdone);
```

`start` indicates `(x0, y0) and (x1, y1) are valid`  

`plot` indicates `(x, y)` is a point to plot

`done` indicates we are ready for the next `start`
Scheduling: Timing Diagram

- **clk**
- **(x0,y0)**: (0,0) to (5,3)
- **(x1,y1)**: (7,4) to (6,4)
- **start**
- **state**: IDLE, RUN, IDLE
- **done**
- **plot**
- **x**: 0, 1, 2, 3, 4, 5, 6, 7, 5, 6
- **y**: 0, 1, 2, 3, 4, 3, 4
- **err**: 3, 6, 2, 5, 1, 4, 0, 3, 0
- **dx, dy**: 7, -4, 1, -1
RTL: The IDLE state

```c
/* C code */
Sint16 dx;
Sint16 dy;
Uint16 x, y;
Sint16 err;
Sint16 e2;
int right;
int down;
dx = x1 - x0;
right = dx > 0;
if (!right) dx = -dx;
dy = y1 - y0;
down = dy > 0;
if (down) dy = -dy;
err = dx + dy;
x = x0;
y = y0;
for (;;) {
    plot(x, y);
}
```

```vhdl
logic signed [11:0] dx, dy, err, e2;
logic right, down;

typedef enum logic {IDLE, RUN} state_t;
state_t state;

always_ff @(posedge clk) begin
    done <= 0;
    plot <= 0;
    if (reset) state <= IDLE;
    else case (state)
        IDLE:
            if (start) begin
                dx = x1 - x0; // Blocking!
                right = dx >= 0;
                if (~right) dx = -dx;
                dy = y1 - y0;
                down = dy >= 0;
                if (down) dy = -dy;
                err = dx + dy;
                x <= x0;
                y <= y0;
                plot <= 1;
                state <= RUN;
            end
end
```
RTL: The RUN state

/* C Code */

for (;;) {
    plot(x, y);
    if (x == x1 && y == y1)
        break;
    e2 = err << 1;
    if (e2 > dy) {
        err += dy;
        if (right) x++;
        else x--;
    }
    if (e2 < dx) {
        err += dx;
        if (down) y++;
        else y--;
    }
}

RUN:
    if (x == x1 && y == y1) begin
        done <= 1;
        state <= IDLE;
    end else begin
        plot <= 1;
        e2 = err << 1;
        if (e2 > dy) begin
            err += dy;
            if (right) x <= x + 10'd 1;
            else x <= x - 10'd 1;
        end
        if (e2 < dx) begin
            err += dx;
            if (down) y <= y + 10'd 1;
            else y <= y - 10'd 1;
        end
    end
    default:
        state <= IDLE;
endcase
end
Datapath for $dx$, $dy$, $right$, and $down$

I: if (start)
   $dx = x_1 - x_0$;
   $right = dx >= 0$;
   if (~right) $dx = -dx$;
   $dy = y_1 - y_0$;
   $down = dy >= 0$;
   if (down) $dy = -dy$;
   $err = dx + dy$;
   $x <= x_0$;
   $y <= y_0$;
   plot <= 1;
   state <= RUN;
R: if (x == x_1 && y == y_1)
   done <= 1;
   state <= IDLE;
else
   plot <= 1;
   e2 = err << 1;
   if (e2 > dy)
      err += dy;
      if (right) x <= x + 10'd 1;
      else x <= x - 10'd 1;
   if (e2 < dx)
      err += dx;
      if (down) y <= y + 10'd 1;
      else y <= y - 10'd 1;
Datapath for err

I: if (start)
    dx = x1 - x0;
    right = dx >= 0;
    if (~right) dx = -dx;
    dy = y1 - y0;
    down = dy >= 0;
    if (down) dy = -dy;
    err = dx + dy;
    x <= x0;
    y <= y0;
    plot <= 1;
    state <= RUN;
R: if (x == x1 && y == y1)
    done <= 1;
    state <= IDLE;
else
    plot <= 1;
    e2 = err << 1;
    if (e2 > dy)
        err += dy;
        if (right) x <= x + 10’d 1;
        else x <= x - 10’d 1;
    if (e2 < dx)
        err += dx;
        if (down) y <= y + 10’d 1;
        else y <= y - 10’d 1;
Datapath for x and y

I: if (start)
    dx = x1 - x0;
    right = dx >= 0;
    if (~right) dx = -dx;
    dy = y1 - y0;
    down = dy >= 0;
    if (down) dy = -dy;
    err = dx + dy;
    x <= x0;
    y <= y0;
    plot <= 1;
    state <= RUN;
R: if (x == x1 && y == y1)
    done <= 1;
    state <= IDLE;
else
    plot <= 1;
    e2 = err << 1;
    if (e2 > dy)
        err += dy;
        if (right) x <= x + 10'd 1;
        else x <= x - 10'd 1;
    if (e2 < dx)
        err += dx;
        if (down) y <= y + 10'd 1;
        else y <= y - 10'd 1;
module VGA_framebuffer(
  input logic clk50, reset,
  input logic [10:0] x, y, // Pixel coordinates
  input logic pixel_color, pixel_write,
  output logic [7:0] VGA_R, VGA_G, VGA_B,
  output logic VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

parameter HACTIVE = 11'd 1280,
HFRONT_PORCH = 11'd 32,
HSYNC = 11'd 192,
HBACK_PORCH = 11'd 96,
HTOTAL =
  HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH; //1600

parameter VACTIVE = 10'd 480,
VFRONT_PORCH = 10'd 10,
VSYNC = 10'd 2,
VBACK_PORCH = 10'd 33,
VTOTAL =
  VACTIVE + VFRONT_PORCH + VSYNC + VBACK_PORCH; //525
// Horizontal counter
logic [10:0] hcount;
logic endOfLine;

always_ff @(posedge clk50 or posedge reset)
  if (reset) hcount <= 0;
  else if (endOfLine) hcount <= 0;
  else hcount <= hcount + 11’d 1;

assign endOfLine = hcount == HTOTAL - 1;

// Vertical counter
logic [9:0] vcount;
logic endOfField;

always_ff @(posedge clk50 or posedge reset)
  if (reset) vcount <= 0;
  else if (endOfLine)
    if (endOfField) vcount <= 0;
    else vcount <= vcount + 10’d 1;
  else vcount <= vcount + 10’d 1;

assign endOfField = vcount == VTOTAL - 1;

assign VGA_HS = !( (hcount[10:7] == 4’b1010) &
  (hcount[6] | hcount[5]) );
assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);
assign VGA_SYNC_n = 1; // Sync on R, G, and B. Unused for VGA.

logic blank;
( vcount[9] | (vcount[8:5] == 4'b1111) ); // 480

logic framebuffer [307199:0]; // 640 * 480
logic [18:0] read_address, write_address;

assign write_address = x + (y << 9) + (y << 7); // x + y * 640
assign read_address =
(hcount >> 1) + (vcount << 9) + (vcount << 7);

logic pixel_read;
always_ff @(posedge clk50) begin
if (pixel_write) framebuffer[write_address] <= pixel_color;
if (hcount[0]) begin
pixel_read <= framebuffer[read_address];
VGA_BLANK_n <= ~blank; // Sync blank with read pixel data
end
end

assign VGA_CLK = hcount[0]; // 25 MHz clock
assign {VGA_R, VGA_G, VGA_B} = pixel_read ? 24’hFF_FF_FF_FF : 24’h0;
endmodule
The “Hallway” Line Generator

```verilog
define module hallway(input logic clk, reset,
                         input logic VGA_VS,
                         input logic done,
                         output logic [10:0] x0, y0, x1, y1,
                         output logic start, pixel_color);

// ...

// Typical state:
S_TOP:
    if (done) begin
        start <= 1;
        if (x0 < 620)
            x0 <= x0 + 10'd 10;
        else begin
            state <= S_RIGHT;
            x0 <= 639;
            y0 <= 0;
        end
    end
end
```

// SoCKit_Top.sv

logic [10:0] x, y, x0, y0, x1, y1;
logic pixel_color;
logic pixel_write;
logic done, start;

VGA_framebuffer fb(.clk50(OSC_50_B3B),
    .reset(~RESET_n),
    .*);

bresenham liner(.clk(OSC_50_B3B),
    .reset(~RESET_n),
    .plot(pixel_write),
    .*);

hallway hall(.clk(OSC_50_B3B),
    .reset(~RESET_n),
    .*);
Testbenches
Testbenches

A model of the environment; exercises the module.

```
// Module to test:  
// Three-bit  
// binary counter

module count3(
    input logic clk,
    reset,
    output logic [2:0] count);

always_ff
    @(posedge clk)
    if (reset)
        count <= 3'd 0;
    else
        count <=
            count + 3'd 1;

endmodule

module count3_tb;
    logic clk, reset;
    logic [2:0] count;

    count3 dut(.*);

    initial begin
        clk = 0;
        forever
            forever
                #20ns clk = ~clk;
    end

    initial begin  // Reset
        reset = 0;
        repeat (2)
            @(posedge clk);
        reset = 1;
        repeat (2)
            @(posedge clk);
        reset = 0;
    end

endmodule
```
Running this in ModelSim