CSE140L: Components and Design Techniques for Digital Systems Lab

Verilog HDL (cont.)

Tajana Simunic Rosing

Source: Eric Crabill, Xilinx
Where we are now…

• What we covered thus far:
  – Appendix A,B, Chap. 1-5, part of Chap 6
  – Transistors, delay
  – Verilog HDL for combinational circuits

• Lab #1 graded, Lab #2 due next week

• Lab #3 will be a team project

• What is next:
  – More of Verilog HDL:
    • Examples of combinational implementations
    • Differences between blocking/non blocking code
    • How to model flip flops and latches in verilog
module life (n0, n1, n2, n3, n4, n5, n6, n7, self, out);
  input n0, n1, n2, n3, n4, n5, n6, n7, self;
  output out;
  reg out;
  reg [7:0] neighbors;
  reg [3:0] count;
  reg [3:0] i;

assign neighbors = {n7, n6, n5, n4, n3, n2, n1, n0};

always @(neighbors or self) begin
  count = 0;
  for (i = 0; i < 8; i = i+1) count = count + neighbors[i];
  out = (count == 3);
  out = out | ((self == 1) & (count == 2));
end
endmodule

More complex behavioral model
Delay in Assignment

- Delayed assignment:
  - $\Delta t$ time units pass before the statement is executed and LHS assignment is made
- Intra-assignment delay:
  - RHS is evaluated immediately but there is a delay of $\Delta t$ before the result is placed in LHS
  - If another procedure changes RHS signals during $\Delta t$, it does not affect the output

```plaintext
Delayed assignment
#\Delta t \ variable = expression;

Intra-assignment delay
variable = #\Delta t \ expression;
```

```plaintext
reg [6:0] sum; reg h, ziltch;
ziltch = #15 ckz&h; /* ckz&a evaluated now; ziltch changed after 15 time units. */
#10 hat = b&c; /* 10 units after ziltch changes, b&c is evaluated and hat changes. */
```
Delay Control

• Control the timing of assignments in procedural blocks by:
  – Level triggered timing control.
    • wait (!reset);
    • wait (!reset) a = b;
  – Simple delays.
    • #10;
    • #10 a = b;
  – Edge triggered timing control.
    • @(a or b);
    • @(a or b) c = d;
    • @(posedge clk);
    • @(negedge clk) a = b;
module Compare1 (Equal, Alarger, Blarger, A, B);
    input     A, B;
    output    Equal, Alarger, Blarger;

    assign #5 Equal = (A & B) | (~A & ~B);
    assign #3 Alarger = (A & ~B);
    assign #3 Blarger = (~A & B);
endmodule
Delay Control

- Generation of clock and resets in testbench:

```verilog
reg rst, clk;
initial // this happens once at time zero
begin
    rst = 1'b1; // starts off as asserted at time zero
    #100; // wait for 100 time units
    rst = 1'b0; // deassert the rst signal
end
always // this repeats forever
begin
    clk = 1'b1; // starts off as high at time zero
    #25; // wait for half period
    clk = 1'b0; // clock goes low
    #25; // wait for half period
end
```
Driving a simulation through a “testbench”

```verilog
module testbench (x, y);
    output x, y;
    reg [1:0] cnt;

initial begin
    cnt = 0;
    repeat (4) begin
        #10 cnt = cnt + 1;
        $display (@ time=%d, x=%b, y=%b, cnt=%b,
        $time, x, y, cnt); end
    #10 $finish;
end

assign x = cnt[1];
assign y = cnt[0];
endmodule
```

- **2-bit vector**: `reg [1:0] cnt;
- **Initial block executed only once at start of simulation**: `initial begin
- **Print to a console**: `$display (@ time=%d, x=%b, y=%b, cnt=%b,
    $time, x, y, cnt); end
- **Directive to stop simulation**: `#10 $finish;`
Blocking/Non-Blocking Assignments

• Blocking assignments (X=A)
  – completes the assignment before continuing on to next statement

• Non-blocking assignments (X<=A)
  – completes in zero time and doesn’t change the value of the target until a blocking point (delay/wait) is encountered

• Watch out for assignments with delay!
  – Examples follow…. 
Blocking Delay – Unintended Behavior

module adder_t1 (co, sum, a, b, ci);
    output    co;
    output [3:0] sum;
    input [3:0] a, b;
    input     ci;
    reg       co;
    reg [3:0] sum;

always @(a or b or ci)
    #12 {co, sum} = a + b + ci;
endmodule
module adder_t4 (co, sum, a, b, ci);
    output co;
    output [3:0] sum;
    input [3:0] a, b;
    input ci;

    assign #12 {co, sum} = a + b + ci;
endmodule

Trigger the assign statement

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>ci</th>
<th>sum</th>
<th>co</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>A</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
```

0 12 15 17 19 21 24 27 29 31 33 36
Nonblocking delay – expected behavior

module adder_t3 (co, sum, a, b, ci);
    output co;
    output [3:0] sum;
    input [3:0] a, b;
    input ci;
    reg co;
    reg [3:0] sum;

    always @(a or b or ci)
        {co, sum} <= #12 a + b + ci;
endmodule
CSE140L: Components and Design Techniques for Digital Systems Lab

Flip Flops & Verilog HDL

Tajana Simunic Rosing

Source: Eric Crabill, Xilinx
• **Flip-flop**: Bit storage that stores on clock edge, not level
• Master-slave design:
Comparison of latches and flip-flops

- **positive edge-triggered flip-flop**
- **transparent (level-sensitive) latch**
Flip-flop in Verilog

- Use always block's sensitivity list to wait for clock edge

```verilog
module dff (clk, d, q);

    input  clk, d;
    output q;
    reg    q;

    always @(posedge clk)
        q = d;

endmodule
```
module unknown (clk, d, q);

    input  clk, d;
    output q;
    reg    q;

    always @(clk)
        q = d;

endmodule
Blocking/Non-Blocking Assignments & FFs

• Blocking assignments (X=A)
  – completes the assignment before continuing on to next statement

• Non-blocking assignments (X<=A)
  – completes in zero time and doesn’t change the value of the target until a blocking point (delay/wait) is encountered

• Example: copy vs. swap

```verilog
always @(posedge CLK) begin
    A = B;
    B = A;
end

always @(posedge CLK) begin
    A <= B;
    B <= A;
end
```
Flip-flop features

- Reset (set state to 0) – R
  - synchronous: \( D_{\text{new}} = R' \cdot D_{\text{old}} \) (when next clock edge arrives)
  - asynchronous: doesn't wait for clock

- Preset or set (set state to 1) – S (or sometimes P)
  - synchronous: \( D_{\text{new}} = D_{\text{old}} + S \) (when next clock edge arrives)
  - asynchronous: doesn't wait for clock

- Both reset and preset (set and reset dominant)
  - \( D_{\text{new}} = R' \cdot D_{\text{old}} + S \) (set-dominant)
  - \( D_{\text{new}} = R' \cdot D_{\text{old}} + R'S \) (reset-dominant)

- Selective input capability (input enable or load) – LD or EN
  - multiplexor at input: \( D_{\text{new}} = LD' \cdot Q + LD \cdot D_{\text{old}} \)
  - load may or may not override reset/set (usually R/S have priority)

- Complementary outputs – Q and Q'
More Flip-flops

- Synchronous/asynchronous reset/set
  - single thread that waits for the clock
  - three parallel threads – only one of which waits for the clock

### Synchronous

```verilog
module dff (clk, s, r, d, q);
  input  clk, s, r, d;
  output q;
  reg    q;
  always @(posedge clk)
    if (r)      q = 1'b0;
    else if (s) q = 1'b1;
    else        q = d;
endmodule
```

### Asynchronous

```verilog
module dff (clk, s, r, d, q);
  input  clk, s, r, d;
  output q;
  reg    q;
  always @(posedge r)
    q = 1'b0;
  always @(posedge s)
    q = 1'b1;
  always @(posedge clk)
    q = d;
endmodule
```
-blocking/non-blocking with delay in FFs

always @(posedge clk)
begin
\[ Z=Y; \ Y=X; // shift register \]
\[ y=x; \ z=y; // parallel ff. \]
end

initial
begin
\[ a=1; \ b=2; \ c=3; \]
\[ #5 \ a = b + c; // wait for 5 units, and execute a = b + c = 5. \]
\[ d = a; // Time continues from last line, d = b + c \text{ at } t=5. \]
end

always @(posedge clk)
begin
\[ Z<=Y; \ Y<=X; // shift register \]
\[ y<=x; \ z<=y; // also a shift register. \]
end

initial
begin
\[ #3 \ b <= a; // * grab a \text{ at } t=0 \text{ Deliver } b \text{ at } t=3. \]
\[ #6 \ x <= b + c; // grab b+c \text{ at } t=0, \text{ wait and assign } x \text{ at } t=6. \]
\[ x \text{ is unaffected by b's change. */} \]
Blocking vs. Nonblocking Assignment Guidelines

- Guidelines:
  - when modeling both sequential and combinational logic within the same always block, use non-blocking assignments.
  - do not mix blocking and non-blocking assignments in the same always block
  - with multiple non-blocking assignments to the same variable the last assignment wins!