CSE140L: Components and Design Techniques for Digital Systems Lab

Verilog HDL

Tajana Simunic Rosing

Source: Eric Crabill, Xilinx
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
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

```
Delayed assignment
#\( \Delta t \) variable = expression;

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

```
reg [6:0] sum; reg h, zilch;
zilch = #15 ckz&h; /* ckz&a evaluated now; zilch changed
                   after 15 time units. */
#10 hat = b&c; /* 10 units after zilch 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
```
System Tasks

• The $ sign denotes Verilog system tasks, there are a large number of these, most useful being:
  – $display(“The value of a is %b”, a);
    • Used in procedural blocks for text output.
    • The %b is the value format (binary, in this case…)
  – $finish;
    • Used to finish the simulation.
    • Use when your stimulus and response testing is done.
  – $stop;
    • Similar to $finish, but doesn’t exit simulation.
Driving a simulation through a “testbench”

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);
        #10 $finish;
    end
end

assign x = cnt[1];
assign y = cnt[0];
endmodule
Blocking and 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

• Example: copy vs. swap

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

```verilog
always @(posedge CLK) begin
  A <= B;
  B <= A;
end
```
Blocking vs. nonblocking with delay

**Blocking**

- variable = expression;
- variable = \#\Delta t expression;
- grab inputs now, deliver ans.

later.

- \#\Delta t variable = expression;
- grab inputs later, deliver ans.

**Non-Blocking**

- variable <= expression;
- variable <= \#\Delta t expression;
- \#\Delta t variable <= expression;

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

```verilog
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=5 = b+c at t=5.
end
```

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

```verilog
initial
begin
    #3 b <= a; /* grab a at t=0 Deliver b at t=3.
    #6 x <= b + c; // grab b+c at t=0, wait and assign x at t=6.
    x is unaffected by b’s change. */
end
```

```verilog
reg G[7:0];
always @(posedge clk)
    G <= { G[6:0], G[7]}; // End around rotate 8-bit register.
```
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

---

Trigger the always block
Output changes only 3ns after last input change

---

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>12</th>
<th>15</th>
<th>17</th>
<th>19</th>
<th>21</th>
<th>24</th>
<th>27</th>
<th>29</th>
<th>31</th>
<th>33</th>
<th>36</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ci</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>co</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
module adder_t4 (co, sum, a, b, ci);
    output    cc;
    output [3:0] sum;
    input [3:0] a, b,
    input    ci;

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

Blocking vs. Nonblocking Assignment Guidelines

- Guidelines:
  - use nonblocking assignments for sequential logic
  - use blocking assignments for combinational logic in always blocks
  - when modeling both sequential and combinational logic within the same always block, use nonblocking assignments.
  - do not mix blocking and nonblocking assignments in the same always block
  - Multiple nonblocking assignments to the same variable – the last assignment wins!
CSE140L: Components and Design Techniques for Digital Systems Lab

Flip Flops

Tajana Simunic Rosing
D Flip-Flop

- **Flip-flop**: Bit storage that stores on clock edge, not level
- **Master-slave design:**

![Diagram of D Flip-Flop](image)
Comparison of latches and flip-flops

- Positive edge-triggered flip-flop
- Transparent (level-sensitive) latch
## FF Types

<table>
<thead>
<tr>
<th>FLIP-FLOP NAME</th>
<th>FLIP-FLOP SYMBOL</th>
<th>CHARACTERISTIC TABLE</th>
<th>CHARACTERISTIC EQUATION</th>
<th>EXCITATION TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>![SR Diagram]</td>
<td>![SR Table]</td>
<td>$Q_{\text{next}} = S + R'Q$</td>
<td>![SR Excitation Table]</td>
</tr>
<tr>
<td>JK</td>
<td>![JK Diagram]</td>
<td>![JK Table]</td>
<td>$Q_{\text{next}} = JQ' + K'Q$</td>
<td>![JK Excitation Table]</td>
</tr>
<tr>
<td>D</td>
<td>![D Diagram]</td>
<td>![D Table]</td>
<td>$Q_{\text{next}} = D$</td>
<td>![D Excitation Table]</td>
</tr>
<tr>
<td>T</td>
<td>![T Diagram]</td>
<td>![T Table]</td>
<td>$Q_{\text{next}} = TQ' + T'Q$</td>
<td>![T Excitation Table]</td>
</tr>
</tbody>
</table>
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
  - multiplexer 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'
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
```
What does this code do?

module unknwn (clk, d, q);

input  clk, d;
output  q;
reg  q;

always @(clk)
  q = d;

endmodule
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
design_file
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
design_file
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
```
Non-Ideal Flip-Flop Behavior

- Can’t change flip-flop input too close to clock edge
  - Setup time: time that D must be stable before edge
    - Else, stable value not present at internal latch
  - Hold time: time that D must be held stable after edge
    - Else, new value doesn’t have time to loop around and stabilize in internal latch

Setup time violation
Leads to oscillation!
Timing: Definitions

- Setup time
  - Minimum time before the clocking event by which the input must be stable ($T_{su}$)
- Hold time:
  - Minimum time after the clocking event until which the input must remain stable ($T_h$)
- Propagation delay
  - Amount of time for value to propagate from input to output ($T_{pd}$)
Clock skew

D-FF original state: IN = 0, Q0 = 1, Q1 = 1
due to skew, next state becomes: Q0 = 0, Q1 = 0, and not Q0 = 0, Q1 = 1

• The problem
  – correct behavior assumes next state of all storage elements determined by all storage elements at the same time
  – this is difficult in high-performance systems because time for clock to arrive at flip-flop is comparable to delays through logic
  – effect of skew on cascaded flip-flops:
• Violating setup/hold time can lead to a metastable state
  – Metastable state: Any flip-flop state other than a stable 1 or 0
    • Eventually settles to one or other, but we don’t know which
  – Fix: for internal circuits make sure to observe setup time; not possible for external inputs (e.g. button press)

• Partial solution
  – Insert synchronizer flip-flop for asynchronous input
    • flip-flop w very small setup/hold time, but doesn’t completely prevent metastability
Metastability

Probability of flip-flop being metastable is…

- One flip-flop doesn’t completely solve problem
  - Add more synchronizer flip-flops to decrease the probability of metastability
  - Can’t solve completely – just decrease the likelihood of failure
Metastability and asynchronous inputs

- Clocked synchronous circuits
  - inputs, state, and outputs sampled or changed in relation to a common reference signal (called the clock)
  - e.g., master/slave, edge-triggered
- Asynchronous circuits
  - inputs, state, and outputs sampled or changed independently of a common reference signal (glitches/hazards a major concern)
  - e.g., R-S latch
- Asynchronous inputs to synchronous circuits
  - inputs can change at any time, will not meet setup/hold times
  - dangerous, synchronous inputs are greatly preferred
  - cannot be avoided (e.g., reset signal, memory wait, user input)
Synchronization failure

- Occurs when FF input changes close to clock edge
  - the FF may enter a metastable state – neither a logic 0 nor 1 –
  - it may stay in this state an indefinite amount of time
  - this is not likely in practice but has some probability

small, but non-zero probability that the FF output will get stuck in an in-between state

oscilloscope traces demonstrating synchronizer failure and eventual decay to steady state
Dealing with synchronization failure

- Reduce the probability of failure:
  - (1) slow down the system clock
    - This gives the synchronizer more time to decay into a steady state; synchronizer failure becomes a big problem for very high speed systems
  - (2) use fastest possible logic technology in the synchronizer
    - This makes for a very sharp "peak" upon which to balance
  - (3) cascade two synchronizers
    - This effectively synchronizes twice (both would have to fail)
Handling asynchronous inputs

In is asynchronous and fans out to D0 and D1. One FF catches the signal, one does not inconsistent state may be reached!
Glitching

• Glitch: Temporary values on outputs that appear soon after input changes, before stable new output values

• Designer must determine whether glitching outputs may pose a problem
  – If so, may consider adding flip-flops to outputs
    • Delays output by one clock cycle, but may be OK
CSE140L: Components and Design Techniques for Digital Systems Lab

Counters and FSMs

Tajana Simunic Rosing

Source: Eric Crabill, Xilinx
Mobius Counter in Verilog

initial
  begin
    A = 1'b0;
    B = 1'b0;
    C = 1'b0;
    D = 1'b0;
  end

always @(posedge clk)
  begin
    A <= ~D;
    B <= A;
    C <= B;
    D <= C;
  end
Light Game FSM

- Tug of War game
  - 7 LEDs, 2 push buttons (L, R)
module Light_Game (LEDS, LPB, RPB, CLK, RESET);
  input LPB;
  input RPB;
  input CLK;
  input RESET;
  output [6:0] LEDS;
  reg [6:0] position;
  reg left;
  reg right;

  wire L, R;
  assign L = ~left && LPB;
  assign R = ~right && RPB;
  assign LEDS = position;

  always @(posedge CLK)
  begin
    left <= LPB;
    right <= RPB;
    if (RESET) position <= 7'b0001000;
    else if ((position == 7'b0000001) || (position == 7'b1000000)) ;
    else if (L) position <= position << 1;
    else if (R) position <= position >> 1;
  end
endmodule
Finite string pattern recognizer

• Output a 1 when …010… appears in a bit string, and stop when …100 appears
Finite string pattern recognizer in Verilog

• Verilog description including state assignment

```verilog
module string (clk, X, rst, Q0, Q1, Q2, Z);
input clk, X, rst;
output Q0, Q1, Q2, Z;

parameter S0 = [0,0,0]; //reset state
parameter S1 = [0,0,1]; //strings ending in ...0
parameter S2 = [0,1,0]; //strings ending in ...01
parameter S3 = [0,1,1]; //strings ending in ...010
parameter S4 = [1,0,0]; //strings ending in ...1
parameter S5 = [1,0,1]; //strings ending in ...10
parameter S6 = [1,1,0]; //strings ending in ...100

reg state[0:2];
assign Q0 = state[0];
assign Q1 = state[1];
assign Q2 = state[2];
assign Z = (state == S3);

always @(posedge clk) begin
    if (rst) state = S0;
    else case (state)
        S0: if (X) state = S4 else state = S1;
        S1: if (X) state = S2 else state = S1;
        S2: if (X) state = S4 else state = S3;
        S3: if (X) state = S2 else state = S6;
        S4: if (X) state = S4 else state = S5;
        S5: if (X) state = S2 else state = S6;
        S6: state = S6;
        default: begin
            $display ("invalid state reached");
            state = 3'bxxx;
        end
    endcase
end
endmodule
```
A synchronous 3-bit counter has a mode control M
- when M = 0, the counter counts up in the binary sequence
- when M = 1, the counter advances through the Gray code sequence

Binary: 000, 001, 010, 011, 100, 101, 110, 111
Gray: 000, 001, 011, 010, 110, 111, 101, 100
module string (clk, M, rst, Z0, Z1, Z2);
input clk, X, rst;
output Z0, Z1, Z2;

parameter S0 = [0,0,0];
parameter S1 = [0,0,1];
parameter S2 = [0,1,0];
parameter S3 = [0,1,1];
parameter S4 = [1,0,0];
parameter S5 = [1,0,1];
parameter S6 = [1,1,0];
parameter S7 = [1,1,1];

reg state[0:2];

assign Z0 = state[0];
assign Z1 = state[1];
assign Z2 = state[2];

always @(posedge clk) begin
  if rst state = S0;
  else
    case (state)
      S0: state = S1;
      S1: if (M) state = S3 else state = S2;
      S2: if (M) state = S6 else state = S3;
      S3: if (M) state = S2 else state = S4;
      S4: if (M) state = S0 else state = S5;
      S5: if (M) state = S4 else state = S6;
      S6: if (M) state = S7 else state = S7;
      S7: if (M) state = S5 else state = S0;
    endcase
  endcase
end

endmodule
HDLs vs. PLs

- **Program structure**
  - instantiation of multiple components of the same type
  - specify interconnections between modules via schematic
  - hierarchy of modules
- **Assignment**
  - continuous assignment (logic always computes)
  - propagation delay (computation takes time)
  - timing of signals is important (when does computation have its effect)
- **Data structures**
  - size explicitly spelled out - no dynamic structures
  - no pointers
- **Parallelism**
  - hardware is naturally parallel (must support multiple threads)
  - assignments can occur in parallel (not just sequentially)