

## EE 231L

### Using Verilog to Design Sequential Circuits

In order to design a sequential circuit, you need to use a logic element with memory – a flip-flop or a latch. Verilog has several types of such elements – a latch, a D flip-flop, a JK flip-flop, an SR flip-flop and a toggle flip-flop. Here we will discuss two of these elements — the latch and the D flip-flop.

**Latch** A latch has the inputs D and Clk and the output Q. The if clause defines that the Q output must take the value of D when Clk=1. Since no else clause is given, a latch will be synthesized to maintain the value of Q when Clk=0. The sensitivity list includes Clk and D because both of these signals can cause a change in the value of the Q output.

```
module Altera_Seq_Verilog1 (D, Clk, Q);
    input D, Clk;
    output reg Q;

    always @(D, Clk)
        if(Clk)
            Q = D;

endmodule
```

**D flip-flop** The flip-flop is a bit different from the latch in the sense that it could be positive or negative edge-triggered flip-flop. The sensitivity list contains only the clock signal because it is the only signal that can cause a change in the Q input. The key word **posedge** specifies that a change may occur only on the positive edge of Clock. At this time the output Q is set to the value of the input D.

```
module Altera_Seq_Verilog1 (D, Clock, Q);
    input D, Clock;
    output reg Q;

    always @(posedge Clock)
        Q = D;

endmodule
```

A D flip-flop could also have an asynchronous or synchronous active-low reset (clear) input. When a reset input is equal to 0, the flip-flop's Q output is set to 0. In an asynchronous reset input, the sensitivity list needs to have the reset signal **Resetn**, along with the positive edge of the clock. The keyword **negedge** cannot be omitted because the sensitivity list cannot have both edge-triggered and level-sensitive signals. Note also that **non-blocking** (<=) versus **blocking** (=) assignments need to be used. All non-blocking assignment statements in an always block are evaluated using the values that the variables have when the always block was entered.

```

module Altera_Seq_Verilog1 (D, Clock, Resetn, Q);
  input D, Clock, Resetn;
  output reg Q;

  always @(negedge Resetn, posedge Clock)
    if(!Resetn)
      Q <= 0;
    else
      Q <= D;

endmodule

```

In an asynchronous reset, the signal is acted upon only when a positive clock edge arrives.

```

module Altera_Seq_Verilog1 (D, Clock, Resetn, Q);
  input D, Clock, Resetn;
  output reg Q;

  always @(posedge Clock)
    if(!Resetn)
      Q <= 0;
    else
      Q <= D;

endmodule

```

Let's build a simple 3-bit up counter: it will count 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, ... Here is the state transition table:

Present State			Next State			Count
y2	y1	y0	Y2	Y1	Y0	
0	0	0	0	0	1	0
0	0	1	0	1	0	1
0	1	0	0	1	1	2
0	1	1	1	0	0	3
1	0	0	1	0	1	4
1	0	1	1	1	0	5
1	1	0	1	1	1	6
1	1	1	0	0	0	7

You can design the counter by specifying the transition table and let Verilog determine the state transition equations using a case statement:

```

module Altera_Seq_Verilog3 (Clock, Resetn, Y);
input Clock, Resetn;
output reg [2:0] Y;
reg [2:0] y;

    always @(y)
    begin
        case(y)
            3'b000: Y=3'b001;
            3'b001: Y=3'b010;
            3'b010: Y=3'b011;
            3'b011: Y=3'b100;
            3'b100: Y=3'b101;
            3'b101: Y=3'b110;
            3'b110: Y=3'b111;
            3'b111: Y=3'b000;
        endcase
    end

    always @(negedge Resetn, posedge Clock)
        if (Resetn == 0) y <= 3'b000;
        else y <= Y;

endmodule

```

However, there is a much easier way to design counters. The inputs to D flip-flops are the outputs of the D flip-flops plus one:

```

module Altera_Seq_Verilog3 (Clock, Resetn, Y);
input Clock, Resetn;
output [2:0] Y;
reg [2:0] tmp;

    always @(negedge Resetn, posedge Clock)
        if (Resetn == 0) tmp = 3'b000;
        else tmp = tmp + 1'b1;
    assign Y = tmp;

endmodule

```

This gives you the ability to design very large counters which would be hard to do using other techniques. A 16-bit counter has  $2^{16}$  or 65,536 states. It is difficult to develop the Boolean equations, and impractical to enter a transition table with 65,536 lines.

```
module Altera_Seq_Verilog3 (Clock, Resetn, Y);
input Clock, Resetn;
output [15:0] Y;
reg [15:0] tmp;

    always @(negedge Resetn, posedge Clock)
        if (Resetn == 0) tmp = 16'b0;
        else tmp = tmp + 16'b1;
    assign Y = tmp;

endmodule
```

Another, more powerful, way to design sequential circuits is with state machines. We will discuss how to do this in Lab 4.

When you simulate a sequential circuit, you need to do a *Timing Analysis*. Before you can do a timing analysis, you have to tell Quartus which input is your clock signal. To do this, go to **Assignments — Timing Analysis Settings**. Click on **Individual Clocks**, and select **New**. Give the clock input a name (perhaps *clock*), and select the node which is used for the clock. The Required *f<sub>max</sub>* setting is the maximum speed for the clock signal (for now, just choose **1 MHz**). The timing analysis will tell you if your circuit can work at that frequency.

After setting up your clock signal, recompile your design. Quartus should no longer give the warning "Found pins functioning as undefined clocks and/or memory enables". Go to **Processing — Simulator Tool**, and make sure the Simulation mode is set for **Timing**. You can now create a **Vector Waveform File** to simulate your design.

When you design with sequential circuits, one (or more) of your inputs will function as a clock.