Today: Verilog and Sequential Logic

- Flip-flops
  - representation of clocks - timing of state changes
  - asynchronous vs. synchronous
- Counters (State Machines)
  - structural view (FFs separate from combinational logic)
  - behavioral view (synthesis of sequencers)
- More Advanced Verilog Features
  - $display() and $time() statements
  - non-blocking (RTL) assignment

Incorrect Flip-flop in Verilog

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

```
module dff (CLK, d, q);

    input  CLK, d;
    output q;
    reg    q;

    always @(CLK)
        q = d;

endmodule
```

Not correct! Q will change whenever the clock changes, not just on the edge.

Correct Flip-flop in Verilog

- Use always block's sensitivity list AND the posedge keyword to wait for clock <u>edge</u>

```
module dff (CLK, d, q);

    input  CLK, d;
    output q;
    reg    q;

    always @(posedge 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

```
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

```
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
```

Verilog Implementation of a Counter (State Machine)

- General view of a counter or state machine in verilog

```
module Ctr (CLK, in, out);
    input       CLK;
    input       in;
    output      out;
    reg         out;

    // state variable
    reg [1:0]   state;

    // local variable
    reg [1:0]   next_state;

    always @(posedge CLK) // registers
        state = next_state;

    always @(state or in)
      // Compute next_state[1:0] logic (D inputs) whenever state/inputs change.
      // Make sure state is always assigned to in every execution path!

    // assign to the outputs

endmodule
```

Verilog BCD Counter Example

```
module BCDCount (CLK, clear, load, a0, a1);
  input CLK, reset, in;
  output a0, a1;
  reg a0, a1;
  reg [1:0] state;       // state variables
  reg [1:0] next_state;

  always @(posedge CLK) begin
    state = next_state;
  end

  always @(state or clear or load) begin
    case (state)
      2'b00: next_state = 2'b01;
      2'b01: next_state = 2'b10;
      2'b10: next_state = 2'b11;
      2'b11: next_state = 2'b00;
    endcase
    if (clear) next_state = 2'b00;
    … // handle load
  end

  assign a0 = state[0];
  assign a0 = state[1];
endmodule
```

$display and $time statements

- Documentation in the online manual (p. 56)
- Doesn't synthesize to anything!
- Formats similar to printf() in C
  - %h Hex, %d Decimal, %o Octal, %b Binary, %% Display a “%” sign
- Examples of $display()
  - $display(”output %d is %h", i, vec[i]);
  - $display("%d%% completed", (count * 100) / max_count);
- The $time function returns system simulation time as a 32-bit integer
  - $display("Got an event at time %d", $time);

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: swap

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

RTL Assignment

- Non-blocking assignment is also known as an RTL assignment
  - if used in an always block triggered by a clock edge
  - mimic register-transfer-level semantics – all flip-flops change together

```
// B, C, and D all get the value of A
always @(posedge clk)
   begin
      B = A;
      C = B;
      D = C;
   end
```

```
// this implements a shift operation
always @(posedge clk)
   begin
      {D, C, B} = {C, B, A};
   end
```

```
// implements a shift operation too
always @(posedge clk)
   begin
      B <= A;
      C <= B;
      D <= C;
   end
```