Outline for Today's Lecture

- Sequential logic
- Flip Flops & Latches
- Behavior statements
- Blocking/Non blocking assignments
Sequential Logic

• Verilog uses **idioms** to describe latches, flip-flops and FSMs

• Other coding styles may simulate correctly but produce incorrect hardware

**idiom**
/ˈɪdɪəm/

*noun*
plural noun: idioms

a group of words established by usage as having a meaning not deducible from those of the individual words
Always Statement

General Structure:

always @(sensitivity list)
  statement;

Whenever the event in sensitivity list occurs, statement is executed
module flop(input clk,
            input [3:0] d,
            output [3:0] q);

    always @(posedge clk)
        q <= d;             // pronounced “q gets d”

endmodule
Resettable D Flip-Flop

module flopr(input clk,
               input reset,
               input [3:0] d,
               output [3:0] q);

  // synchronous reset
  always @(posedge clk)
      if (reset) q <= 4'b0;
      else       q <= d;

endmodule
module flopr(input clk,
    input reset,
    input [3:0] d,
    output [3:0] q);

// asynchronous reset
always @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
else       q <= d;
endmodule
module flopren(input clk, 
    input reset, 
    input en, 
    input [3:0] d, 
    output [3:0] q);

// asynchronous reset and enable
always @(posedge clk, posedge reset) 
    if      (reset) q <= 4'b0; 
    else if (en)    q <= d;

endmodule
module sync (input clk,
    input d,
    output reg q);
reg n1;       // internal signal
always @(posedge clk)
    begin
        n1 <= d;
        q <= n1;
    end
endmodule

Begin/end construct is necessary because multiple statements appear in the always statement. Begin/end not needed when using if/else because if/else counts as a single statement.
module latch(input clk,
             input [3:0] d,
             output reg [3:0] q); //declare q as register

always * (clk, d)
  if (clk) q <= d;

endmodule

The always statement evaluates any time clk or d changes. If clk is HIGH, d flows through to q. q must be declared to be a reg because it appears on the left hand side of in an always statement.

**Warning:** We typically don’t use latches in this text. But you might write code that inadvertently implies a latch. Check synthesized hardware – if it has latches in it, there’s an error.
Other Behavioral Statements

- Statements that must be inside `always` statements:
  - `if/else`
  - `case, casez`
Combinational Logic using always

// combinational logic using an always statement
module gates(input [3:0] a, b,
    output [3:0] y1, y2, y3, y4, y5);

always @ (*) // need begin/end because there is
    begin // more than one statement in always
        y1 = a & b; // AND
        y2 = a | b; // OR
        y3 = a ^ b; // XOR
        y4 = ~(a & b); // NAND
        y5 = ~(a | b); // NOR
    end
endmodule

This hardware could be described with assign statements using fewer lines of code, so it’s better to use assign statements in this case.
Combinational Logic using case

module sevenseg(input [3:0] data,
    output reg [6:0] segments);

    always @ (*)
    case (data)
        //
        0: segments = 7'b111_1110;
        1: segments = 7'b011_0000;
        2: segments = 7'b110_1101;
        3: segments = 7'b111_1001;
        4: segments = 7'b011_0011;
        5: segments = 7'b101_1011;
        6: segments = 7'b101_1111;
        7: segments = 7'b111_0000;
        8: segments = 7'b111_1111;
        9: segments = 7'b111_0011;
        default: segments = 7'b000_0000; // required
    endcase
    endmodule
Combinational Logic using case

- case statement implies combinational logic only if all possible input combinations described
- Remember to use default statement
module priority_casez(input [3:0] a,
                       output [3:0] y);

    always @ (*)
        casez(a)
            4'b1????: y = 4'b1000; // ? = don’t care
            4'b01???: y = 4'b0100;
            4'b001?: y = 4'b0010;
            4'b0001: y = 4'b0001;
            default: y = 4'b0000;
        endcase
    endmodule
Blocking vs. Nonblocking Assignment

- $\leq$ is **nonblocking** assignment
  - Occurs simultaneously with others
- $=$ is **blocking** assignment
  - Occurs in order it appears in file

```vhdl
// Good synchronizer using
// nonblocking assignments
module syncgood(input clk, input d, output q);

reg n1;
always @(posedge clk)
begin
  n1 <= d; // nonblocking
  q <= n1; // nonblocking
end
endmodule
```

```vhdl
// Bad synchronizer using
// blocking assignments
module syncbad(input clk, input d, output q);

reg n1;
always @(posedge clk)
begin
  n1 = d; // blocking
  q = n1; // blocking
end
endmodule
```
Rules for Signal Assignment

• **Synchronous sequential logic:** use `always @(posedge clk)` and nonblocking assignments (`<=`)

  ```
  always @(posedge clk)
  q <= d; // nonblocking
  ```

• **Simple combinational logic:** use continuous assignments (`assign...`)

  ```
  assign y = a & b;
  ```

• **More complicated combinational logic:** use `always @ (*)` and blocking assignments (==)

• Assign a signal in **only one** `always` statement or continuous assignment statement.