Module inputs and outputs

```
module mymodule ( input logic [9:0] SW,
                  input logic [1:0] KEY,
                  output logic [7:0] HEX5,
                  output logic [7:0] HEX4,
                  ...
                  output logic [9:0] LEDR = 10'b0 ); // Initialize with value

endmodule
```

always_comb

- Use =, not <=
- Can have if statements, and/or cases

```
logic [1:0] state;
logic a;
logic b;
...
always_comb begin
    case(state)
        2'b00: begin
            a = 1;
            b = 0;
        end
        2'b01: begin
            a = 0;
            b = 1;
        end
        default: begin
            a = 0;
            b = 0;
        end
    endcase
end
```

- If you have ‘dont care’ values, can use casez with ? for dont care.
- It prioritizes first written statements first, so be careful
- MUST ALWAYS USE DEFAULT or else program retains previous case

```
  always_comb begin
    casez (SW[6:0])
      7'b00?????: LEDR[2:0] = 3'b000; //add
      7'b01?????: LEDR[2:0] = 3'b001; //subtract
      7'b1000000: LEDR[2:0] = 3'b000; //add
      7'b1000001: LEDR[2:0] = 3'b000; //add
      7'b1000010: LEDR[2:0] = 3'b000; //add
      7'b1000011: LEDR[2:0] = 3'b001; //subtract
      7'b10010??: LEDR[2:0] = 3'b101; //slt
      7'b10110??: LEDR[2:0] = 3'b011; //or
      default: LEDR[2:0] = 3'b000;
    endcase
  end
```

always_ff

- use non blocking <= assignments
- use for sequential logic (change state on clock edge)

```
always_ff @(posedge clk) begin
    if (!reset) begin
        ...
    end
    else if (...) begin
    ...
    end
    else begin
    ...
    end
end
```

For loop

```
for (int i = 0; i < 10; i++) begin
    ...
end
```

7 seg display

- active-low, inverted logic so 0 == ON.
- HEX[7] is just the decimal point. Normally can just use HEX[6:0] for numbers
- To assign, count from HEX[7] (decimal), then HEX[6] (middle), then its descending order counter-clockwise. Ends at HEX[0] (top).

```
HEX0 = 8'b11111001 // Displays '1' on HEX0
```

Instances of modules

```
module fsm_synth (input logic clk,
        input logic resetn,
        input logic enter,
        input logic match,
        output logic [1:0] outputs,
        output logic save_pw_out,
        output logic save_at_out );
...
endmodule

//In another module
  fsm_synth safe_fsm (
        .clk(MAX10_CLK1_50),
        .resetn(reset),
        .enter(enter),
        .match(MATCH),
        .outputs(PRESENT_STATE),
        .save_pw_out(savePW),
        .save_at_out(saveAT)
    );
```

FSMs

- An FSM module consists of state definitions, always_comb for outputs, and always_comb for next state behaviour.
- You also need an always_ff for reset behaviour.

```
enum int unsigned { STATE_A = 0, STATE_B = 1, STATE_C = 2, STATE_D = 4}
    present_state; next_state;

always_comb begin
    case(present_state)
        STATE_A: outputs = ...;
        STATE_B: outputs = ...;
        STATE_C: outputs = ...;
        STATE_D: outputs = ...;
    endcase
end

always_comb begin
    next_state = present_state; // By default, stay at same state
    case(present_state)
        STATE_A: begin
            if (...) next_state = STATE_B;
            else if (...) next_state = STATE_C; // else is important
        STATE_B: begin
            ...
    endcase
end

always_ff @(posedge clk) begin
    if (reset) begin
        present_state <= STATE_A; // Or whichever your default state is.
    end
    else begin
        present_state <= next_state;
    end
end
```

Misc notes:

- KEY are 1 when not pressed, so invert the logic.
- 4'b0000 or similar format can assign to an array to corresponding size ([3:0] in this case)
- Assignments:

```
// HEX5-HEX1 are [7:0] size arrays
localparam logic [47:0] OPEN = 48'hFC_C0_8C_86_AB_F7;
    ...
// This assigns them in order. hFC -> HEX5, C0 -> HEX4, etc.
{HEX5, HEX4, HEX3, HEX2, HEX1, HEX0} = OPEN;
```

- Can have cases for HEX which convert from binary automatically. eg. for a 4 bit binary input we can have:

```
always_comb begin
        case (data_in)
            4'h0: segments = 7'b1000000; // 0
            4'h1: segments = 7'b1111001; // 1
            4'h2: segments = 7'b0100100; // 2
            4'h3: segments = 7'b0110000; // 3
            4'h4: segments = 7'b0011001; // 4
            4'h5: segments = 7'b0010010; // 5
            4'h6: segments = 7'b0000010; // 6
            4'h7: segments = 7'b1111000; // 7
            4'h8: segments = 7'b0000000; // 8
            4'h9: segments = 7'b0010000; // 9
            4'hA: segments = 7'b0001000; // A
            4'hB: segments = 7'b0000011; // b
            4'hC: segments = 7'b1000110; // C
            4'hD: segments = 7'b0100001; // d
            4'hE: segments = 7'b0000110; // E
            4'hF: segments = 7'b0001110; // F
            default: segments = 7'b1111111; // Off
        endcase
    end
```

- To only detect button press once, implement logic to check if button was pressed in previous cycle.

```
logic prevPressed;

always_ff @(posedge clk) begin
    prevPressed <= push;
end

// Now we can make assignments later with taking the button input once only.
always_ff @(posedge clk) begin
    if (reset) begin
        led = 4'b0;
    end
    else if (push && !prevPressed) begin
        ...
    end
end
```

- Most things can be done with regular operations instead of logic.
- Think in terms of bit shifts, addition, etc.

```
module fulladder ( input logic [3:0] a,
                   input logic [3:0] b,
                   input logic cin,
                   output logic [3:0] sum,
                   output logic cout );

always_comb begin
    {cout, sum} = a + b + cin;

end

endmodule
```