FINITE STATE MACHINES
II. General FSMs

The State of the Machine

• The state can be encoded with a compact binary representation
  – Ex: 5 states
  – Minimum number of state bits = ceil(log₂(5)) = 3 bits
  – Total possible states with 3 bits = 2³ = 8
  – There is probably limited value in optimizing which 5 of the 8 possible states you choose despite the fact you may have spent time looking at this in EEC 180A / EEC 18. You could try in a critical situation.
II. General FSMs
The State of the Machine

- The *state* can be encoded with a “one-hot” representation
  - Ex: 5 states
    - Number of state bits = Number of states = 5 bits
  - No min, no max, no optimizing
  - 00001
  - 00010
  - 00100
  - 01000
  - 10000
  + Zero-time state decode logic
  + Very fast state increment (shift not addition)
- Not practical for a very large number of states
II. General FSMs
The I/O of the Machine

• Typically the outputs of FSMs are derived from the state but often not a copy of the state as is often true with counters

• Mealy machines have outputs that are functions of the inputs which creates a purely-combinational path through the FSM from input to output which could limit the maximum clock frequency
II. General FSMs

Major Components

- FSMs contain two major circuit structures
  1) Next State combinational logic
  2) State register (row of FFs with their clocks tied together)
- We could also include a third—output logic which is a function of state (Moore) or state and inputs (Mealy)
- Both (1) and (2) are usually coded with `always` blocks in verilog
- Always keep a clear picture of the output(s) and input(s) of each
- The Next State logic in this example:
  - output(s)  `state_c`
  - input(s)  `state, reset, abc, xyz`
II. General FSMs

Next State Logic

• Give thought to the logic you are creating
• In many cases, it will be best to write statements that directly use inputs of the Next State logic rather than other internal combinational logic variables
• In the example below, the longest path from the input(s) to the output \textit{state}_c passes through \textbf{three} adders and a case selector (mux)

```vhdl
//--- Next State combinational logic
always @(freq or xyz or abc or reset) begin
...
  // logic
  d = abc + 8'h01;  // no issues
  e = d + 8'h01;   // creates state + 2
  f = e + 8'h01;   // creates state + 3
  case (state) begin
    3'b000: state_c = d;
    3'b001: state_c = e;
    default: state_c = f;
  endcase
end
...
II. General FSMs

**Next State Logic**

- In the example below, the longest path from the input(s) to the output `state_c` passes through one adder and a case selector (mux).
- Even though the critical path is far shorter than the critical path of the previous example, the function is exactly the same.

```verilog
//--- Next State combinational logic
always @(freq or xyz or abc or reset) begin
    // logic
    d = abc + 8'h01; // no issues
    e = abc + 8'h02; // creates state + 2
    f = abc + 8'h03; // creates state + 3
    case (state) begin
        3'b000: state_c = d;
        3'b001: state_c = e;
        default: state_c = f;
    endcase
    ...
end
```
II. General FSMs

Next State Logic

- Example Next State combinational circuit with output \( \text{freq}_c \)
  - Always declare default values at beginning of always blocks
  - Use all of the best combinational logic design practices

```vhdl
//--- Next State combinational logic
always @ (freq or xyz or abc or reset) begin
  // defaults
  freq_c = freq;  // an example
  // logic
  case (freq)
  ... endcase
  // logic
  if (xyz==4'b0010) begin
    ... end
  // reset logic is usually last for highest priority
  if (reset == 1'b1) begin
    freq_c = 3'b000;
  end
end
```

higher priority or precedence
gets the “final word”
Five Things in Virtually Every Well-Designed Verilog State Machine

1) Default: \texttt{state\_c = state;}
2) \texttt{case (state)}
3) \texttt{STATE: begin \ldots \ldots \ldots \ end} for each state. Partition design by \texttt{state}.
   (In traditional 180A-style design, we partition the overall design by \textit{bits of the state}.)
4) \texttt{if (reset == 1'b1)} at the end of the combinational \texttt{always} block
5) Instantiate FF register(s) in a separate \texttt{always} block

```verilog
// Next State combinational logic
always @(state or reset or ...) begin
  // defaults
  state\_c = state;  // hold previous state
  case (state) begin
    INIT: begin
      ...
    end
    // Add a case target for each state
    default: c\_freq = 3'bxxx;  // error case
  endcase
  // reset logic often last for highest priority
  if (reset == 1'b1) begin
    state\_c = 3'b000;
  end
end

// instantiate the state register
always @(posedge clock) begin
  state <= #1 state\_c;
end
```

© B. Baas
Characteristics of a Well-Designed Verilog State Machine with Integrated Counters

1) In the default section, it is probably a good idea to have an auto-increment or auto-decrement statement
   - \texttt{count_c = count + 8'h01;} \hspace{1em} // Example increment
   - \texttt{count_c = count - 8'h01;} \hspace{1em} // Example decrement

2) In the “idle” or “wait” state, it is probably a good idea to hold the counter value to eliminate unnecessary toggling to reduce power dissipation
   - \texttt{count_c = count;} \hspace{1em} // Example hold counter value
   - \texttt{count_c = 8'h00;} \hspace{1em} // Example hold counter at zero
Example Controller Specification

- There are four states
  - **IDLE**
    - Go to PREP when go is asserted
  - **PREP**
    - Do something for 10 cycles
    - Then go to JOB1 if $x \leq 5$
    - Then go to JOB2 if $x > 5$
  - **JOB1**
    - Do something for 5 cycles
    - Then go to IDLE
  - **JOB2**
    - Do something for 20 cycles
    - Then go to IDLE
- *reset* at any time returns controller to IDLE state
Example State Diagram

- State diagram

```
Example State Diagram

• State diagram

reset → IDLE

IDLE → PREP, go

PREP → JOB2, 10 cycles

PREP → JOB1, go

Job1 → IDLE, After 10~ if x ≤ 5

Job1 → JOB2, 5 cycles

Job2 → IDLE, After 10~ if x > 5

20 cycles
```
Control Block Example Solution
(There are many solutions!)

• What are the registers (the values that need to be remembered)?
  1. state for the main controlling state machine
  2. count to count number of cycles in some of the states

• State registers
  – Choose two bits (obviously the minimum) for the four states

• Counter(s)
  – Choose one five bit counter since states are independent and counter
    can be shared between different states
  – Counting down may be slightly better (simpler shared comparator
    for all cases that compare with zero when done)

• Keep registers (flip-flops) separate from state machine logic
  – Always do this for this class

• It is normally clearer to define states with names (such as IDLE) rather than constants (such as 2'b01)
Example Circuit Diagram

- The Circuit diagram in this case is essentially the same as a detailed block diagram
Example State Machine
(untested, could have bugs—especially syntax or off-by-one-cycle bugs)

parameter IDLE = 2’h0; // constants in hex notation
parameter PREP = 2’h1;
parameter JOB1 = 2’h2;
parameter JOB2 = 2’h3;

reg [1:0] state, c_state; // declare both FF regs
reg [4:0] count, c_count; // and comb. logic regs

// Combinational logic for state machine
always @(state or count or go or x or reset) begin
  // defaults (place first)
  c_state = state; // default same state
  c_count = count - 5'b00001; // default count down

  // main state machine logic
  case (state)
    IDLE: begin
      if (go == 1'b1) begin
        c_state = PREP;
        c_count = 5’d10; // constant in decimal
      end
      else begin
        c_count = 0; // only for lower power
      end
    end
    PREP: begin
      if (count == 5’b00000) begin // count will underflow to -1 but it is ok
        c_state = IDLE;
      end
    end
    default: begin // good practice, but not used here
      c_state = 2’bx0; // better for testing
      c_state = IDLE; // another option
    endcase

    // reset logic (place last to override other logic)
    if (reset == 1’b1) begin
      c_state = IDLE;
      c_count = 5’b00000;
    end // end of always block

    // Instantiate registers (flip-flops)
    always @(posedge clk) begin
      state <= #1 c_state;
      count <= #1 c_count;
    end

  JOB1: begin
    if (count == 5’b00000) begin
      c_state = IDLE;
      // count will underflow to -1 but it is ok
    end
  end
  JOB2: begin
    if (count == 5’b00000) begin
      c_state = IDLE;
      // count will underflow to -1 but it is ok
    end
  end

  default: begin // good practice, but not used here
    c_state = 2’bx0; // better for testing
    c_state = IDLE; // another option
  endcase

I think it is better to put reset logic inside the control logic rather than with the FF declaration