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 = \( \text{ceil}(\log_2(5)) = 3 \) bits
  – Total possible states with 3 bits = \( 2^3 = 8 \)
  – 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 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
  – Ex: counter \( \text{out} = \text{count}; \)
  – Ex: FSM \( \text{out} = (\text{state} == \text{DONE}) \&\& (x == 8'hF0); \)

• Moore machines have outputs that are a function of the state only

• 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
3 Key Signal Groups

- One of the first steps in the design process is to identify and write down the following independent key signal names and word widths:
  1. inputs
  2. state
     - 1. FSM state
     - 2. counter(s)
  3. outputs
II. General FSMs

Major Components

- FSMs contain two major circuit structures
  1) State register (row of FFs with their clocks tied together)
  2) Next State combinational logic
- 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. For example, write logic as a function of reset, abc, xyz, and/or state; rather than state_c.
- In the example below, the longest path from the input(s) to the output state_c passes through **three** adders and a case selector (mux).

```verbatim
//--- Next State combinational logic
always @(state 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
```

© B. Baas
II. General FSMs

Next State Logic

- In the example below, the longest path from the input(s) to the output \textit{state\_c} passes through \textbf{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)
    3'b000: state_c = d;
    3'b001: state_c = e;
    default: state_c = f;
  endcase

end
```

© B. Baas
II. General FSMs

Next State Logic

- Example Next State combinational circuit with output `freq_c`
  - Always declare default values at beginning of always blocks
  - Use all of the best combinational logic design practices

```verbatim
//--- 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: `state_c = state;`
2) `case (state)`
3) `STATE: begin .......... end` for each state. Partition design by `state`.
   (In contrast, in traditional EEC18-style design, we partition the overall design by bits of the state.)
4) `if (reset == 1'b1)` at the end of the combinational `always` block
5) Instantiate FF register(s) in a separate `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
```

```verilog
// instantiate the state register
always @(posedge clock) begin
    state <= #1 state_c;
end
```
1) In the default section, it is probably a good idea to have an auto-increment or auto-decrement statement
   - count_c = count + 8'h01;  // Example increment
   - count_c = count - 8'h01;  // 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
   - count_c = count;  // Example hold counter value
   - count_c = 8’h00;  // Example hold counter at zero
Finite State Design Example

• 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
State Diagram

- State diagram

Diagram:
- IDLE
  - Transition to IDLE with reset
  - Transition to PREP with go
- PREP
  - Transition to IDLE with go
  - Transition to JOB1 with 10 cycles
  - Transition to JOB2 with 5 cycles
- JOB1
  - Transition to IDLE with go
  - Transition to PREP with 5 cycles
- JOB2
  - Transition to IDLE with go
  - Transition to PREP with 20 cycles

Conditions:
- After 10~ if \( x \leq 5 \)
- After 10~ if \( x > 5 \)
• 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 three counters, one for each use – wasteful
  – 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**)
The Circuit diagram in this case is very similar to the detailed block diagram.

- Inputs: reset, go, x[7:0]
- FFs: state[1:0], count[4:0]
- Outputs: not specified, probably driven by state[1:0]
Example Verilog Implementation

**fsm.v**

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

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

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

  // main state machine logic
  case (state)
  IDLE: begin
    if (go == 1'b1) begin
      state_c = PREP;
      count_c = 5'd09;  // constant in decimal
    end else begin
      count_c = 5'd00;  // only for lower power
    end
  end
  PREP: begin
    if (count <= 5'b00000) begin
      if (x <= 8'd005) begin  // assume 8-bit x
        state_c = JOB1;  // goto JOB1
        count_c = 5'd04;
      end else begin  // goto JOB2
        state_c = JOB2;
        count_c = 5'd19;
      end
    end
  endcase
  end
  // reset logic (place last to override other logic)
  if (reset == 1'b1) begin
    state_c = IDLE;
    count_c = 5'b00000;
  end
end // end of always block

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

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

- reset\textsubscript{cold} to reset the clock oscillator
- reset to reset the state machine
- IDLE $\rightarrow$ PREP $\rightarrow$ JOB1 (since $x[7:0] \leq 8d05$)
Design Compiler Synthesis
Complete Gate Netlist

Related to count

DFF_X1 \( \text{count[2]} \) ( .D(\text{count_c[2]}), .CK(\text{clk}), .Q(\text{count[2]}), .QN(\text{n35}) );
DFF_X1 \( \text{count[3]} \) ( .D(\text{count_c[3]}), .CK(\text{clk}), .Q(\text{count[3]});
DFF_X1 \( \text{count[4]} \) ( .D(\text{count_c[4]}), .CK(\text{clk}), .Q(\text{count[4]});
DFF_X1 \( \text{count[1]} \) ( .D(\text{count_c[1]}), .CK(\text{clk}), .Q(\text{count[1]});
DFF_X1 \( \text{count[0]} \) ( .D(\text{count_c[0]}), .CK(\text{clk}), .Q(\text{count[0]}), .QN(\text{N8});
XOR2_X1 U33 ( .A(\text{count[3]}), .B(\text{n33}), .Z(\text{n29}) );
XNOR2_X1 U34 ( .A(\text{count[4]}), .B(\text{n34}), .ZN(\text{n30}) );
OAI22_X1 U36 ( .A1(\text{n21}), .A2(\text{n42}), .B1(\text{n40}), .B2(\text{n18}), .ZN(\text{count_c[2]});
NOR3_X1 U41 ( .A(\text{count[1]}), .A2(\text{count[0]}), .A3(\text{n26}), .ZN(\text{n17}) );
OR3_X1 U42 ( .A(\text{count[4]}), .A2(\text{count[3]}), .A3(\text{count[2]}), .ZN(\text{n26}) );
OAI22_X1 U45 ( .A(\text{reset}), .B1(\text{n21}), .B2(\text{n30}), .ZN(\text{count_c[4]});;
OAI22_X1 U46 ( .A(\text{reset}), .A2(\text{n22}), .B1(\text{n21}), .B2(\text{n29}), .ZN(\text{count_c[3]});;
OAI22_X1 U47 ( .A(\text{reset}), .A2(\text{n20}), .B1(\text{n21}), .B2(\text{n31}), .ZN(\text{count_c[1]});;
NOR2_X1 U53 ( .A(\text{reset}), .A2(\text{n25}), .ZN(\text{count_c[0]});;
NOR2_X1 U56 ( .A(\text{count[1]}), .A2(\text{count[0]}), .ZN(\text{n32}) );
OAI21_X1 U57 ( .B1(\text{count[0]}), .B2(\text{count[1]}), .A(\text{n32}), .ZN(\text{n31}) );
NOR2_X1 U60 ( .A1(\text{count[3]}), .A2(\text{n33}), .ZN(\text{n34}) );

Related to state

DFF_X1 \( \text{state[1]} \) ( .D(\text{n28}), .CK(\text{clk}), .Q(\text{state[1]}), .QN(\text{n4}) );
DFF_X1 \( \text{state[0]} \) ( .D(\text{n27}), .CK(\text{clk}), .Q(\text{state[0]}), .QN(\text{n7}) );
NAND3_X1 U27 ( .A1(\text{n17}), .A2(\text{state[0]}), .A3(\text{n23}), .ZN(\text{n16}) );
NOR2_X1 U43 ( .A1(\text{n7}), .A2(\text{state[1]}), .ZN(\text{n23}) );
OAI22_X1 U48 ( .A1(\text{state[1]}), .A2(\text{n17}), .B1(\text{go}), .B2(\text{n4}), .ZN(\text{n19}) );
NAND3_X1 U28 ( .A1(\text{n43}), .A2(\text{n41}), .A3(\text{n19}), .ZN(\text{n14}) );
NAND3_X1 U29 ( .A1(\text{n17}), .A2(\text{n41}), .A3(\text{n23}), .ZN(\text{n18}) );
NAND2_X1 U30 ( .A1(\text{n24}), .A2(\text{n41}), .ZN(\text{n21}) );
NAND3_X1 U31 ( .A1(\text{n7}), .A2(\text{n4}), .A3(\text{go}), .ZN(\text{n22}) );
NAND3_X1 U32 ( .A1(\text{n17}), .A2(\text{n40}), .A3(\text{n23}), .ZN(\text{n20}) );
INV_X1 U35 ( .A(\text{n20}), .ZN(\text{n39}) );
INV_X1 U37 ( .A(\text{n10}), .ZN(\text{n42}) );
INV_X1 U38 ( .A(\text{n23}), .ZN(\text{n40}) );
INV_X1 U39 ( .A(\text{n23}), .ZN(\text{n43}) );
INV_X1 U40 ( .A(\text{n36}), .ZN(\text{n37}) );
OAI21_X1 U44 ( .B1(\text{n17}), .B2(\text{n43}), .A(\text{n4}), .ZN(\text{n24}) );
OAI21_X1 U49 ( .B1(\text{n7}), .B2(\text{n14}), .A(\text{n15}), .ZN(\text{n27}) );
NAND4_X1 U50 ( .A1(\text{n16}), .A2(\text{n14}), .A3(\text{n41}), .A4(\text{n4}), .ZN(\text{n15}) );
OAI21_X1 U51 ( .B1(\text{n4}), .B2(\text{n14}), .A(\text{n18}), .ZN(\text{n28}) );
INV_X1 U52 ( .A(\text{reset}), .ZN(\text{n41}) );
OAI211_X1 U54 ( .C1(\text{n8}), .C2(\text{n24}), .A(\text{n39}), .B(\text{n38}), .ZN(\text{n25}) );
INV_X1 U55 ( .A(\text{n22}), .ZN(\text{n38}) );
NAND2_X1 U58 ( .A1(\text{n32}), .A2(\text{n35}), .ZN(\text{n33}) );
OAI21_X1 U59 ( .B1(\text{n32}), .B2(\text{n35}), .A(\text{n33}), .ZN(\text{n10}) );
OAI211_X1 U61 ( .C1(\text{x2}), .C2(\text{x1}), .A(\text{x4}), .B(\text{x3}), .ZN(\text{n36}) );
NOR4_X1 U62 ( .A1(\text{n37}), .A2(\text{x5}), .A3(\text{x7}), .A4(\text{x6}), .ZN(\text{n23}) );

Relationship is unclear