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($\log_2(5)$) = 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)

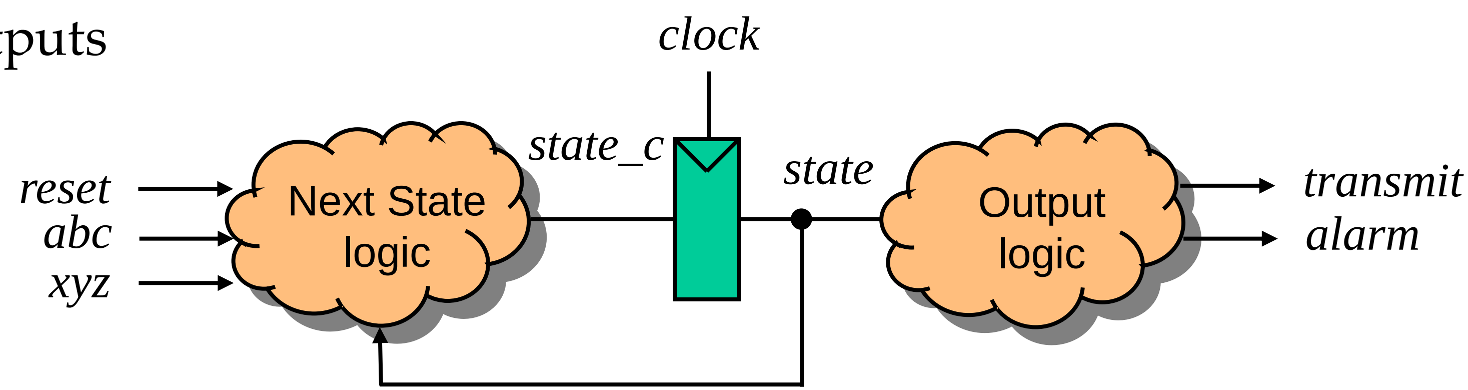
  – Not practical for a very large number of states

II. General FSMs
The I/O of the Machine

- Typically the **outputs** of FSMs are derived from the state (Moore machines) but often not a copy of the state as is often true with counters
  - Ex: counter `out = count;`
  - Ex: FSM `out = (state == 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

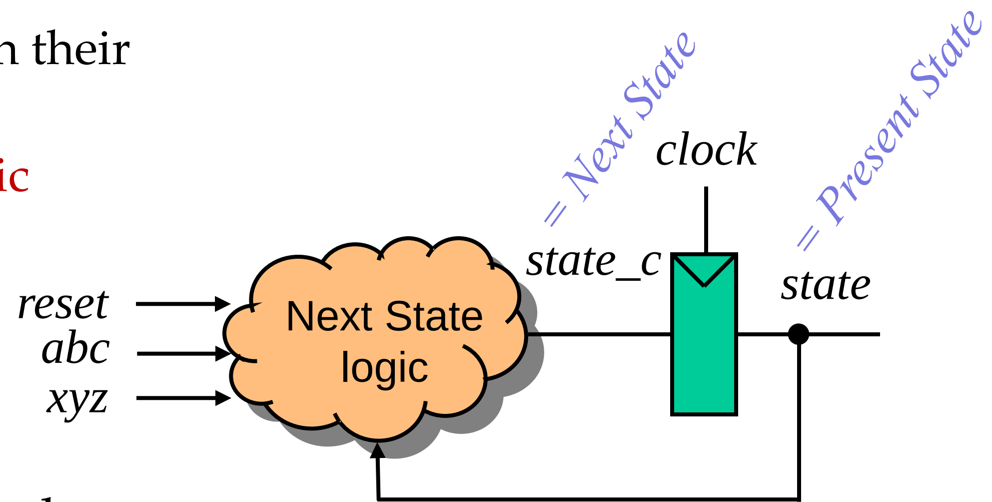
II. General FSMs 3 Key Signal Groups

- 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

- All 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)

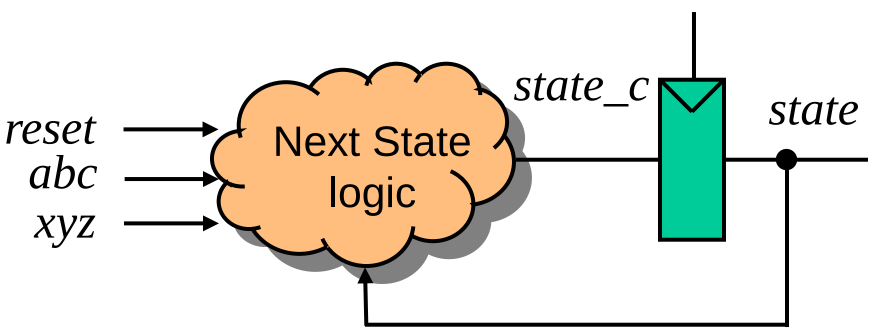


- Both (1) and (2) are usually coded with **`always`** blocks in verilog
- Always keep a clear picture of the output(s) and input(s) of each
- The Next State logic in this example:
  - output(s) *state_c*
  - input(s) *state, reset, abc, xyz*

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)

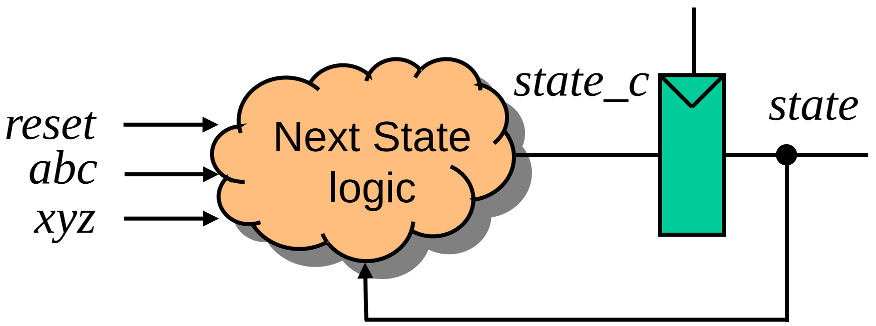
```
//--- Next State combinational logic
always @(state or xyz or abc or reset) begin
   ...

   // 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
```



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

```
//--- 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
```



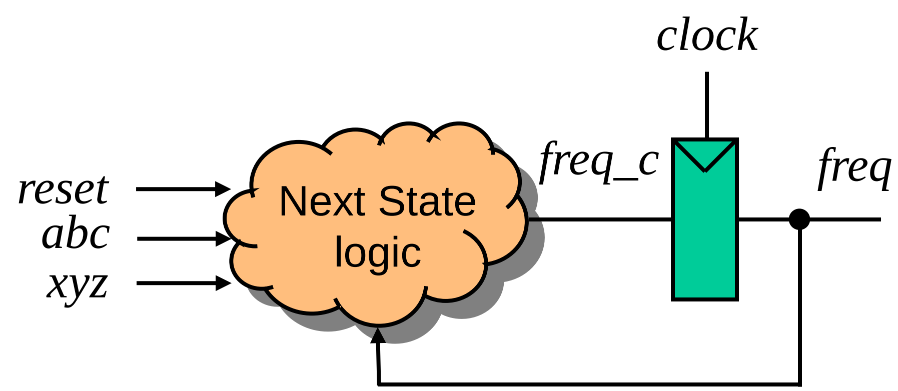
II. General FSMs
Next State Logic

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

higher priority or precedence

gets the "final word"

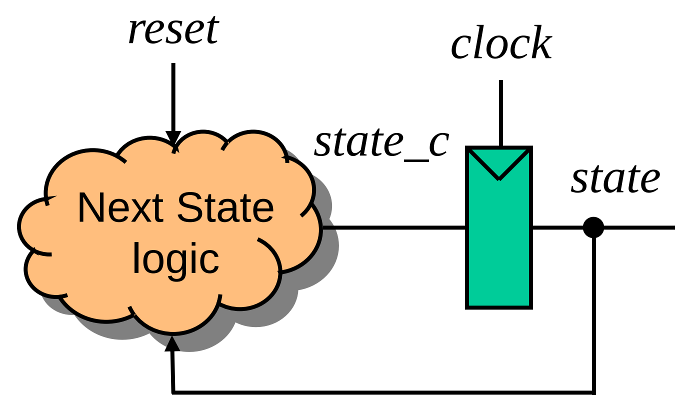
```
//--- Next State combinational logic
always @(freq or xyz or abc or reset) begin
   // defaults
   freq_c = freq;    // an example

   // 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
```



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

```
// 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
```

Characteristics of a Well-Designed Verilog State Machine with Integrated Counters

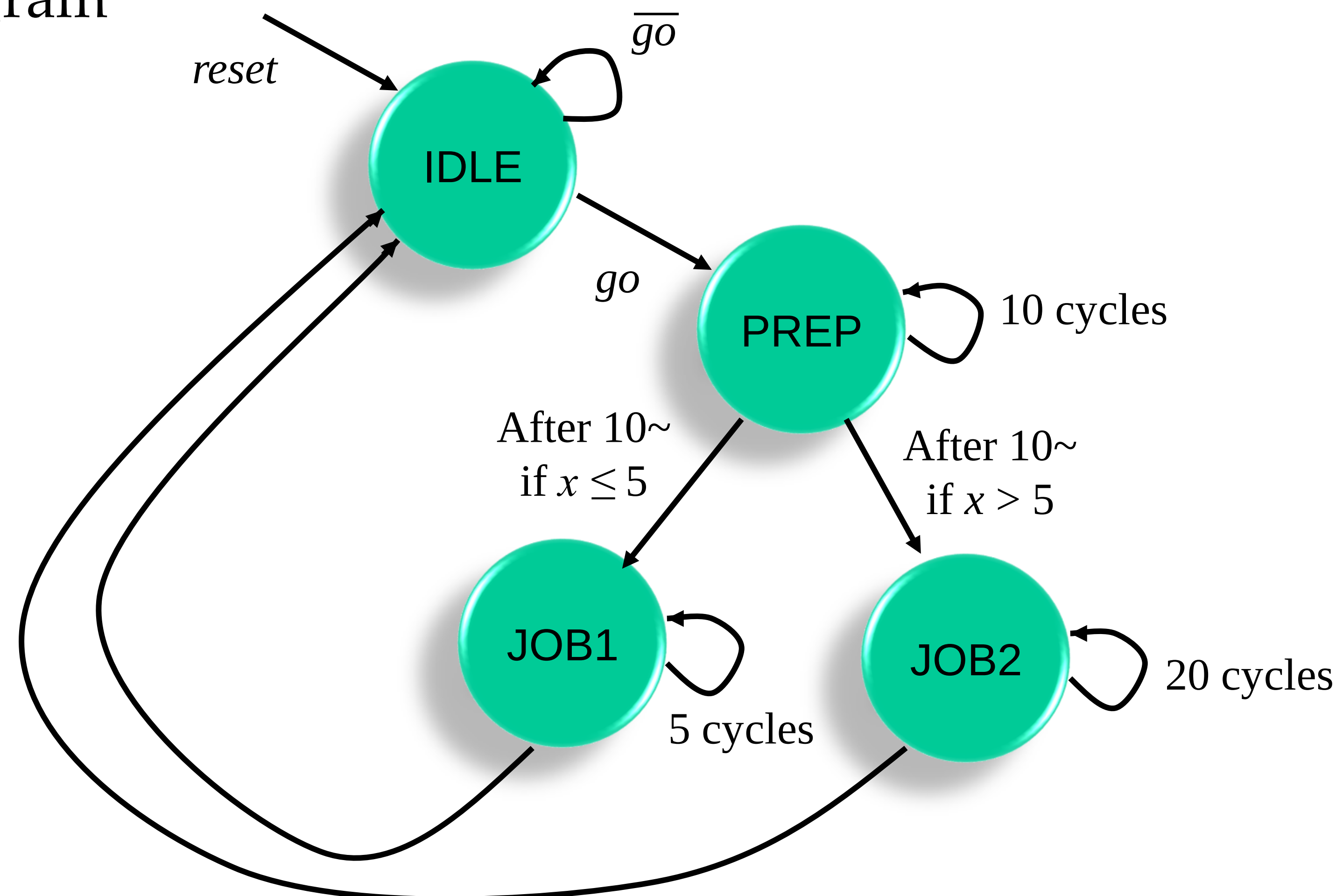
1) In the default section, it is probably a good idea to have an auto-increment or auto-decrement statement
   - `count_c = count + 8’h01;    // Example increment`
   - `count_c = count - 8’h01;    // Example decrement`
2) In the “idle” or “wait” state, it is probably a good idea to hold the counter 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 $x <= 5$
    - 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

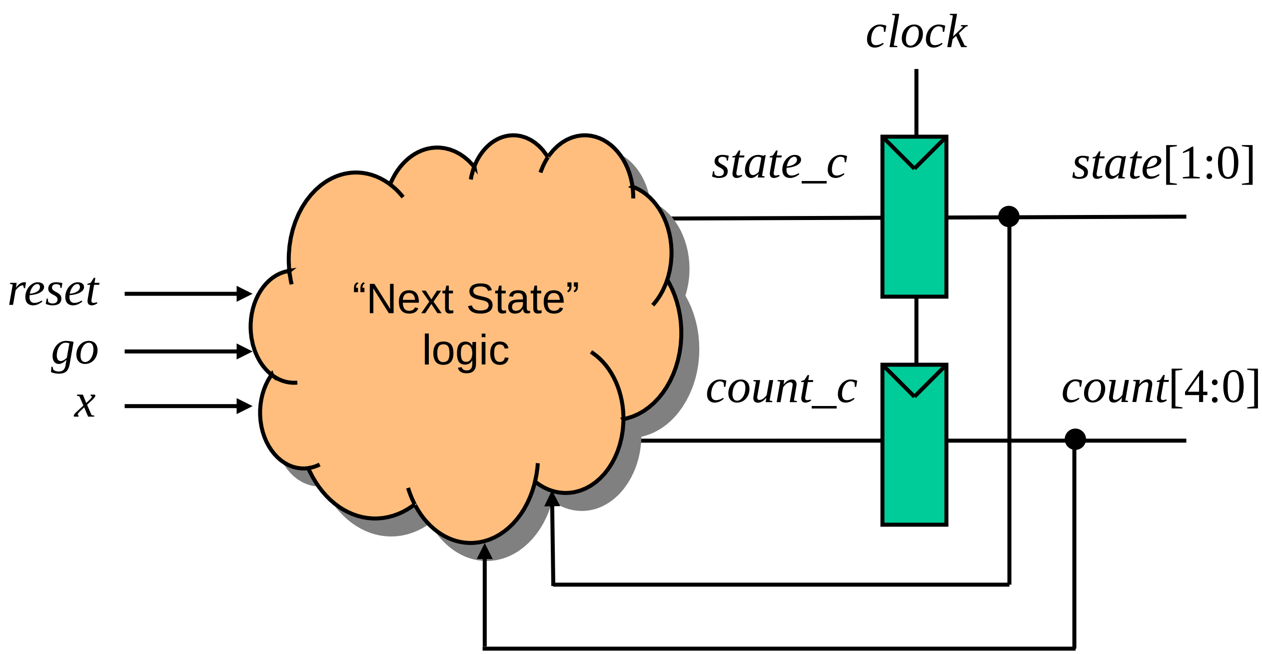


Control Block Example Solution (There are many solutions!)

- 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 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`**)

Circuit Diagram

- 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]`



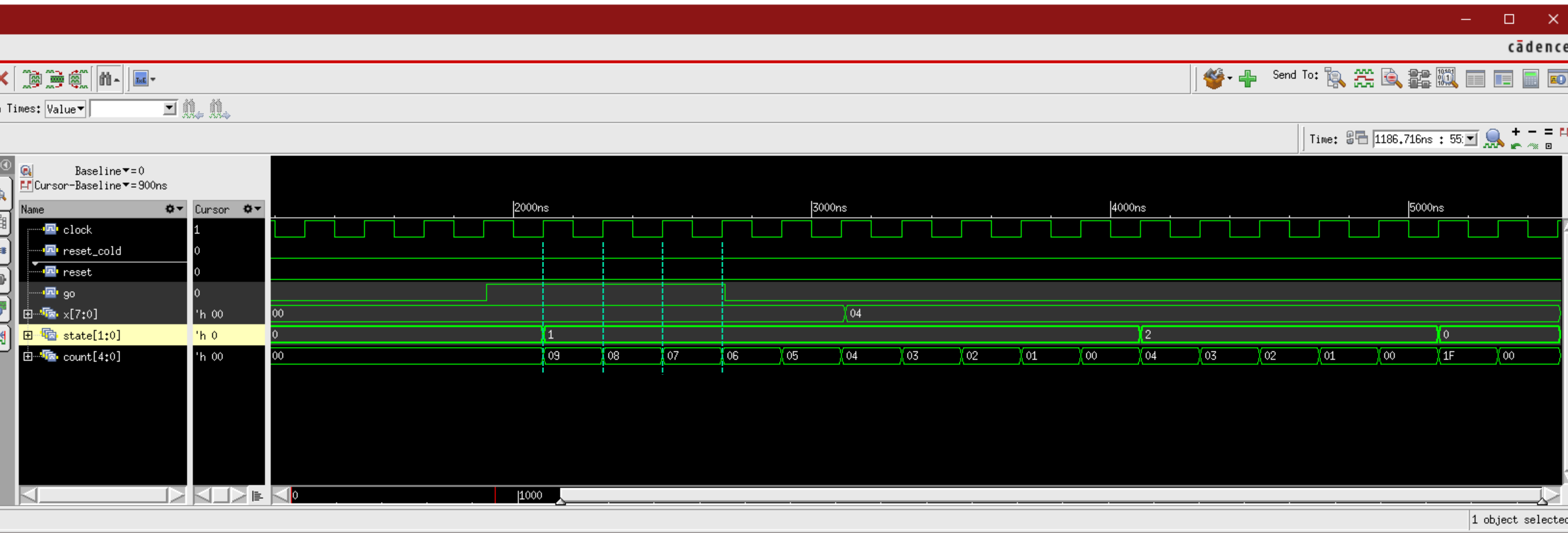
Example Verilog Implementation **fsm.v**

```
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
       end

      JOB1: begin
         if (count == 5'b00000) begin
            state_c = IDLE;
            // count will underflow to -1 for 1~, no prob
         end
       end

      JOB2: begin
         if (count == 5'b00000) begin
            state_c = IDLE;
            // count will underflow to -1 for 1~, no prob
         end
       end

      default: begin     // good practice, but not used here
         state_c = 2'bxx;   // better for testing
         state_c = IDLE;    // another option
      end
   endcase

   // 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


// Instantiates 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_cold* to reset the clock oscillator
- *reset* to reset the state machine
- IDLE → PREP → JOB1 (since $x[7:0] \le 8'd05$)



Design Compiler Synthesis Complete Gate Netlist

Related to *count*

DFF_X1 \count_reg[2] (.D(count_c[2]), .CK(clk), .Q(count[2]), .QN(n35));
DFF_X1 \count_reg[3] (.D(count_c[3]), .CK(clk), .Q(count[3]));
DFF_X1 \count_reg[4] (.D(count_c[4]), .CK(clk), .Q(count[4]));
DFF_X1 \count_reg[1] (.D(count_c[1]), .CK(clk), .Q(count[1]));
DFF_X1 \count_reg[0] (.D(count_c[0]), .CK(clk), .Q(count[0]), .QN(N8));
XOR2_X1 U33 (.A(count[3]), .B(n33), .Z(n29));
XNOR2_X1 U34 (.A(count[4]), .B(n34), .ZN(n30));
OAI22_X1 U36 (.A1(n21), .A2(n42), .B1(n40), .B2(n18), .ZN(count_c[2]));
NOR3_X1 U41 (.A1(count[1]), .A2(count[0]), .A3(n26), .ZN(n17));
OR3_X1 U42 (.A1(count[4]), .A2(count[3]), .A3(count[2]), .ZN(n26));
OAI22_X1 U45 (.A1(reset), .A2(n20), .B1(n21), .B2(n30), .ZN(count_c[4]));
OAI22_X1 U46 (.A1(reset), .A2(n22), .B1(n21), .B2(n29), .ZN(count_c[3]));
OAI22_X1 U47 (.A1(reset), .A2(n20), .B1(n21), .B2(n31), .ZN(count_c[1]));
NOR2_X1 U53 (.A1(reset), .A2(n25), .ZN(count_c[0]));
NOR2_X1 U56 (.A1(count[1]), .A2(count[0]), .ZN(n32));
AOI21_X1 U57 (.B1(count[0]), .B2(count[1]), .A(n32), .ZN(n31));
NOR2_X1 U60 (.A1(count[3]), .A2(n33), .ZN(n34));

Related to *state*

DFF_X1 \state_reg[1] (.D(n28), .CK(clk), .Q(state[1]), .QN(n4));
DFF_X1 \state_reg[0] (.D(n27), .CK(clk), .Q(state[0]), .QN(n7));
NAND3_X1 U27 (.A1(n17), .A2(state[0]), .A3(N23), .ZN(n16));
NOR2_X1 U43 (.A1(n7), .A2(state[1]), .ZN(n23));
AOI22_X1 U48 (.A1(state[1]), .A2(n17), .B1(go), .B2(n4), .ZN(n19));

NAND3_X1 U28 (.A1(n43), .A2(n41), .A3(n19), .ZN(n14));
NAND3_X1 U29 (.A1(n17), .A2(n41), .A3(n23), .ZN(n18));
NAND2_X1 U30 (.A1(n24), .A2(n41), .ZN(n21));
NAND3_X1 U31 (.A1(n7), .A2(n4), .A3(go), .ZN(n22));
NAND3_X1 U32 (.A1(n17), .A2(n40), .A3(n23), .ZN(n20));
INV_X1 U35 (.A(n20), .ZN(n39));
INV_X1 U37 (.A(N10), .ZN(n42));
INV_X1 U38 (.A(N23), .ZN(n40));
INV_X1 U39 (.A(n23), .ZN(n43));
INV_X1 U40 (.A(n36), .ZN(n37));
OAI21_X1 U44 (.B1(n17), .B2(n43), .A(n4), .ZN(n24));
OAI21_X1 U49 (.B1(n7), .B2(n14), .A(n15), .ZN(n27));
NAND4_X1 U50 (.A1(n16), .A2(n14), .A3(n41), .A4(n4), .ZN(n15));
OAI21_X1 U51 (.B1(n4), .B2(n14), .A(n18), .ZN(n28));
INV_X1 U52 (.A(reset), .ZN(n41));
AOI211_X1 U54 (.C1(N8), .C2(n24), .A(n39), .B(n38), .ZN(n25));
INV_X1 U55 (.A(n22), .ZN(n38));
NAND2_X1 U58 (.A1(n32), .A2(n35), .ZN(n33));
OAI21_X1 U59 (.B1(n32), .B2(n35), .A(n33), .ZN(N10));
AOI211_X1 U61 (.C1(x[2]), .C2(x[1]), .A(x[4]), .B(x[3]), .ZN(n36));
NOR4_X1 U62 (.A1(n37), .A2(x[5]), .A3(x[7]), .A4(x[6]), .ZN(N23));

Relationship is unclear