CONTROL CIRCUITS
Control in Digital Systems

- Three primary components of digital systems
  - Datapath (does the work)
  - Control (manager)
  - Memory (storage)
Control in Digital Systems

- Control blocks in chips
  - Typically small amount of HW
  - Typically substantial verilog code
  - Therefore:
    - We typically do not care about small circuit area
    - We typically do not care about fast circuits
      - A possible exception is when arithmetic is required to make control decisions e.g., "change states if \( \text{sum} < \text{phase} \)"
    - Verilog code can be complex
      - Many opportunities for bugs
      - May be impossible to test all states and transitions
      - Often the focus of testing by Verification Engineers
Sequential Logic

- *Combinational circuits’ outputs* are a function of the circuit’s inputs and a time delay.
- *Sequential circuits’ outputs* are a function of the circuit’s inputs, previous circuit state, and a time delay.
Control with Finite State Machines

• Of course we can design all standard finite state machines we learned in basic logic design classes
  – Moore type FSM
    • outputs depend only on the present state (state below)
  – Mealy type FSM
    • outputs depend on the present state and the inputs

![Diagram of input, state, logic, and output connections]
Writing Verilog for State Machines

• Design process
  – Think
  – Draw state diagrams if helpful
  – Draw block diagrams if helpful
  – Draw timing diagrams
  – Pick descriptive signal names
  – Think

  – Then…
    • Write verilog
    • Test it
#1 Design Goal for Controllers: Clarity

- **Clear code** $\rightarrow$ bugs will be less likely
- It is even more important to use good signal naming conventions in control logic than with other digital circuits
  - Ex: $state_c \rightarrow state$
- **Reduce the amount of state if possible** (clarity)
  - Ex: It may be better to have one global state machine instead of two separate ones
- **Increase the amount of state if helpful** (clarity)
  - Ex: It may be better to have two separate global state machines instead of a single global one
  - Ex: Instantiate separate counters to count independent events
Good FSM Coding Style

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

```vhdl
// this "always" block instantiates the combinational logic
always @(freq or xyz or abc) begin
  // defaults
  c_freq = freq;   // hold previous value in this case
  // main logic block
  if (xyz==4'b0010) begin
    c_freq = abc;
  end

  case (freq) begin
    3'b000:  c_freq = abc;
    3'b001:  c_freq = abc+3'b001;
    default: c_freq = 3'bxxx;   // error case
  endcase

  // reset logic often last in always block for highest priority
  if (reset == 1'b1) begin
    c_freq = 3'b000;
  end
end
```

```vhdl
// this "always" block instantiates the register
always @(posedge clock) begin
  freq <= #1 c_freq;
end
```
Example Controller Specification

• Four states
  - **IDLE**
    • Go to PREP when go is asserted
  - **PREP**
    • Do something for 10 cycles
    • Go to JOB1 if \(x \leq 5\)
    • 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 control to IDLE
Control Block Example Solution
(There are many solutions!)

- State registers
  - Choose two bits (obviously the minimum) since there are 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 comparator for all uses compares with zero)
- Safest to 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**)

(There are many solutions!)
Example State Machine

- State diagram

```
  IDLE
     \----------------------\
        | reset            |
     \----------------------\
       \                   \\
        | go                |
     \----------------------\
            \              \\  x \leq 5
         \   \          x \geq 5
        \     X          \    \\  5 cycles
            \          \    \\
            \          10 cycles
                \----------------------\
                    | go            |
                \----------------------\
                      \           \\  20 cycles
```
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
    c_state = state; // default same state
    c_count = count – 5’h00001; // default count down
end

// main state machine logic
case (state) begin
    IDLE: begin
        if (go) begin
            c_state = PREP;
            c_count = 5’d10; // constant in decimal
        end
        else begin
            c_count = 0;
        end
    end
    PREP: begin
        if (count == 5’b00000) begin
            if (x <= 5) begin
                goto JOB1
            end
            else begin
                goto JOB2
            end
        end
        c_state = JOB1;
        c_count = 5’d05;
    end
end

// reset logic (place last to override other logic)
if (reset) begin
    c_state = IDLE;
    c_count = 5’h00000;
end

end // end of always block

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

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