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 sum < 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 state
  – Mealy type FSM
    • outputs depend on state and inputs
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
State in FSMs

• 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: Maybe better to have one global state machine instead of two separate ones
• Increase the amount of state if helpful (clarity)
  – Ex: Maybe better to have two separate global state machines instead of a single global one
  – Ex: Instantiate separate counters to count independent events
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

- 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 EEC 281
- It is normally clearer to define states with names (such as IDLE) rather than constants (such as 2'b011)
`define vs. parameter

- There are two methods to simplify state names and other constants by using readable text to represent a number
  - **parameter**
    - Use this for state names in EEC 281
    - Local to a module
    - Usage:
      ```
      parameter   HALT = 4'b0101;
      ...
      if (inst == HALT) begin
      ```
  - **`define** macro
    - Global text macro substitution using a compiler directive
    - Best when helpful to put all definitions in a global file
    - Usage:
      ```
      `define   HALT  4'b0101
      ...
      if (inst == `HALT) begin       // requires “back tick”
      ```
Example State Machine

- State diagram

Diagram:
- States: IDLE, PREP, JOB1, JOB2
- Transitions:
  - IDLE to PREP on 'go'
  - PREP to JOB1 on 'x <= 5'
  - PREP to JOB2 on 'x > 5'
  - JOB1 to IDLE on 'reset'
  - JOB2 to IDLE on 'x <= 5'
  - JOB2 to IDLE on 'x > 5'
- Timing:
  - IDLE to PREP: 5 cycles
  - PREP to JOB1: 10 cycles
  - PREP to JOB2: 20 cycles
  - JOB1 to IDLE: 5 cycles
  - JOB2 to IDLE: 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
  // defaults (place first)
  c_state = state;             // default same state
  c_count = count – 5’b00001;  // default count down

  // 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
          c_state = JOB1;
          c_count = 5’d05;
        end
        else begin // goto JOB2
          c_state = JOB2;
          c_count = 5’d20;
        end
      end
    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
      default: begin // good practice, but not used here
        c_state = 2’bxx;  // better for testing
        c_state = IDLE;  // another option
      endcase
    end

  // reset logic (place last to override other logic)
  if (reset) begin
    c_state = IDLE;
    c_count = 5’b00000;
  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